Advanced Topics in Programming Languages	Spring Semester, 2012

Lecture 9: Typed Lambda Calculus

May 8, 2012

Lecturer: Mooly Sagiv *Scribe: Guy Golan Gueta and Shachar Itzhaky*

1 Defining a Type System for Lambda-Calculus

A type checker, as part of the language compiler, provides many benefits, the most important of which being detection of compile-time errors to prevent a situation where undefined semantics is encountered at run-time. In other words, we wish to define a type system such that if a program is well-typed, then it is guaranteed to have fully defined semantics.

There is no single type system for a given language syntax and operational semantics — instead, choosing the set of types and typing rules is an important design decision that has to be made by the language designer. Failure to define a coherent type system may result in a broken or buggy implementation, or in confusing language semantics. In addition, we would prefer a simple type system, because we would later like to formally prove some properties of well-typed programs; cluttering the type space will make such proofs tedious and cumbersome.

Let us re-examine the compact language of (untyped) lambda-calculus.

$$t \quad ::= \quad x \quad | \quad \lambda x.\ t \quad | \quad t\ t$$

Recall that we defined a *value* to be an abstraction lambda term, that is, $v ::= \lambda x.\ t$. The distinction of value terms is crucial because they determine when a program is allowed to "terminate". A value term is not expected to have any further derivations — it is then considered as the outcome of the program. If non-value term, on the other hand, does not have a valid derivation according to the operational semantics, it is regarded as a "stuck" state. In the context of type systems, we want to avoid such states by making appropriate type checks beforehand.

It is easy to show stuck states in lambda-calculus: the SOS only defines application of a λ-term; the semantics gets stuck when it is a variable instead. Here are some examples:

$$x\ y \qquad x\ \lambda x.\ x \qquad z\ ((\lambda y.\ y)\ z)$$

Note We are assuming call-by-value semantics. in these variant, when we have an application term $t_1\ t_2$, derivation of the argument t_2 can proceed only when the applicant t_1 is a value type. Therefore the third term is also stuck.

Still, this case is a little degenerate, since programs containing free variables are not all that interesting. Indeed, *closed terms* (terms with no free variables, that is, all the variables are bound by λ), **never get stuck** in call-by-value semantics of untyped lambda-calculus. We make things a little more interesting (and complex) by also introducing *primitives* into the lambda-calculus: remember that we can represent boolean values and natural numbers directly using the lambda-calculus syntax:

$$\texttt{tru} = \lambda t.\ \lambda f.\ t \qquad \texttt{fls} = \lambda t.\ \lambda f.\ f$$
$$'0' = \lambda s.\lambda z.\ z \quad '1' = \lambda s.\lambda z.\ s\ z \quad '2' = \lambda s.\lambda z.\ s\ (s\ z) \quad '3' = \lambda s.\lambda z.\ s\ (s\ (s\ z))\ \ldots$$

Similarly encoding control structures such as $\texttt{ite}(cond, t_1, t_2)$ ($\texttt{if}\ \ cond\ \texttt{then}\ t_1\ \texttt{else}\ t_2$) and arithmetic operations such as `succ`, `plus`, `mult`. We can see one shortcoming of this approach by considering the function:

$$g = \lambda b.\ \lambda x.\ \lambda y.\ \texttt{ite}\ b\ \ (x+y)\ \ (x \cdot y)$$

Clearly, $g\ \texttt{tru}\ '3'\ '5' = '8'$ and $g\ \texttt{fls}\ '9'\ '6' = '54'$. But what is $g\ '7'\ \texttt{fls}\ \texttt{tru}$? Since this is a closed term it must hold a meaning in lambda-calculus, but a programmer will find it very hard to trace. Breaking the abstraction we defined for booleans and numerals renders this program's behavior unpredictable. The correct answer (calculated mechanically) is in fact

$$\lambda x.\ \lambda i_0.\ \lambda i_1.\ \lambda i_2.\ \lambda i_3.\ \lambda i_4.\ \lambda i_5.\ \texttt{fls}$$

This small example illustrates the need for type abstraction in programming. So we choose an alternative to using the Church numerals, which is to define opaque constant symbols `true` and `false`, as well as $0, 1, 2, \ldots$, and extend the language syntax with new constructs that express operations on these symbols. When defining a new language construct, one has two options for defining its semantics:

1. By reducing it to an existing construct. These are called *derived forms* (or, in some contexts, *syntactic sugar*).
2. By introducing new operational semantics derivation rules for it.

For example, we can introduce a syntax for booleans:

$$t ::= \quad \ldots \quad | \quad \texttt{true} \quad | \quad \texttt{false} \quad | \quad \texttt{if}\ t\ \texttt{then}\ t\ \texttt{else}\ t$$

and then use the second method to define their meaning:

$$\texttt{if true then}\ t_1\ \texttt{else}\ t_2 \to t_1 \qquad \text{(E-IF-THEN)}$$

$$\texttt{if false then}\ t_1\ \texttt{else}\ t_2 \to t_2 \qquad \text{(E-IF-ELSE)}$$

$$\frac{t_1 \to t_1'}{\texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3 \to \texttt{if}\ t_1'\ \texttt{then}\ t_2\ \texttt{else}\ t_3} \qquad \text{(E-IF-COND)}$$

To distinguish the valid runs, we also have to extend the set of values, otherwise we will consider even the simple program `true` as undefined behavior.

$$v \quad ::= \quad \ldots \quad | \quad \texttt{true} \quad | \quad \texttt{false}$$

Note that using this setting, the previous $g\ 7\ \texttt{false}\ \texttt{true}$ will reach a stuck state:

$$g\ 7\ \texttt{false}\ \texttt{true} \to \texttt{if}\ \ 7\ \ \texttt{then}\ \texttt{false} + \texttt{true}\ \texttt{else}\ \texttt{false} \cdot \texttt{true}$$

Because 7 is neither `true` nor `false` and cannot be reduced to anything. To prevent these stuck states we want the type checker to reject this program.

1.1 Simple Type System with 2 Types

In an attempt to locally fix this stuck condition, we define one type "Bool" that will be assigned to boolean values, and one type "$\to$" that will be assigned to functions, $\lambda x.\ t$.

$$T \quad ::= \quad \text{Bool} \quad | \quad \to$$

With the typing rules:

$$\texttt{true} : \text{Bool} \qquad \text{(T-TRUE)}$$

$$\texttt{false} : \text{Bool} \qquad \text{(T-FALSE)}$$

$$\lambda x.\ t : \rightarrow \qquad \text{(T-ABS)}$$

$$\frac{t_1 : \text{Bool} \quad t_2 : T \quad t_3 : T}{\texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3 : T} \qquad \text{(T-IF)}$$

But now we encounter a problem when trying to state the typing rule for application $(t\ t)$. The terms $f_1 = \lambda x.\ \texttt{true}$ and $f_2 = \lambda x.\ x$ both receive the type $\rightarrow$ according to T-ABS. However, $f_1\ f_2 \rightarrow \texttt{true}$ (type Bool) while $f_2\ f_1 \rightarrow \lambda x.\ \texttt{true}$ (type $\rightarrow$). This example shows that we cannot precisely type a term using the types of the sub-terms. In other words, we cannot define the typing rule for an application *compositionally* in this type system. This is not a good property, hinting that we should improve our selection of types.

1.2 Complicating Things: an Unbounded Type System

Our major setback during the previous attempt was that all functions are mapped to the same type, regardless of the value they operate on and the value they produce. Have we had somehow "remember" the fact that f_1 returns a value of type Bool, while f_2 returns a value of the same type as its argument, we would have been able to correctly distinguish the type of the application term.

We therefore define the following set of types, using a BNF grammar.

$$T \quad ::= \quad \text{Bool} \quad \big| \quad T \rightarrow T$$

Notice, now instead of T being a set of two types, it contains **infinitely many** types, because we can always apply the operator $\rightarrow$ to existing types to receive a new type. This will not stand in our way, however, because we will only deal with finite programs, and such programs will have only a finite number of types out of T.

One problem still remains — typing value terms, where a λ-abstraction term is not applied to any value. How can we know the type of the argument? Consider $f_1 = \lambda x.\ \texttt{true}$. This function can be successfully applied to an argument of type "Bool" or "Bool $\rightarrow$ Bool" (and in fact any other type as well), so shall we say $f_1 : \text{Bool} \rightarrow \text{Bool}$ or $f_1 : (\text{Bool} \rightarrow \text{Bool}) \rightarrow \text{Bool}$? Similarly, we have equally the same justification for $f_2 : \text{Bool} \rightarrow \text{Bool}$ as we have for $f_2 : (\text{Bool} \rightarrow \text{Bool}) \rightarrow (\text{Bool} \rightarrow \text{Bool})$ (recall that $f_2 = \lambda x.\ x$).

To circumvent this issue, we choose to **change the language** itself for the sake of making typing easier. In this case, we require the programmer to *annotate* function parameter types in abstraction terms. So, instead of $\lambda x.\ t$ we shall have $\lambda x : T.\ t$

Before we formally define the association of types to terms, here are several examples of terms with the type they should have, intuitively:

t	τ
$\lambda x : \text{Bool}.\ \texttt{true}$	$\text{Bool} \rightarrow \text{Bool}$
$\lambda x : \text{Bool} \rightarrow \text{Bool}.\ \texttt{true}$	$(\text{Bool} \rightarrow \text{Bool}) \rightarrow \text{Bool}$
$\lambda x : \text{Bool} \rightarrow \text{Bool}.\ x$	$(\text{Bool} \rightarrow \text{Bool}) \rightarrow (\text{Bool} \rightarrow \text{Bool})$
$\lambda x : \text{Bool}.\ \lambda y : \text{Bool}.$ $\texttt{if}\ x\ \texttt{then}\ y\ \texttt{else false}$	$\text{Bool} \rightarrow (\text{Bool} \rightarrow \text{Bool})$

Note The $\rightarrow$ operator is **not** associative, $(\text{Bool} \rightarrow \text{Bool}) \rightarrow \text{Bool}$ has a strictly different meaning from $\text{Bool} \rightarrow (\text{Bool} \rightarrow \text{Bool})$. In order to reduce the use parenthesis, for readability, we say that $\rightarrow$ *associates to the right*, that is $T_1 \rightarrow T_2 \rightarrow T_3 \equiv T_1 \rightarrow (T_2 \rightarrow T_3)$. This allows us to express the type of the last term simply as $\text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool}$ (similar to Haskell and ML).

2 Formulating the Typing Relation

To preserve the desired property of compositionality, we have to mind function bodies (the t in $\lambda x.\ t$). Clearly, the type of the body depends on the type of the argument; on the other hand, the type of the whole function term depends on both the type of the argument **and** the type of the body. This cross-dependency means that type information has to be propagated from the term down to its sub-terms, as well as from the sub-terms back to the root.

To this end, we define the *context* Γ of the typing as a finite mapping of variables to types. Intuitively, Γ will be used to assign types to free variables in sub-terms. We formally define a context using the BNF grammar:

$$\Gamma \quad ::= \quad \varnothing \quad \big| \quad x : T$$

but you can thing of it as just another form to represent a function from some finite domain of variable symbols to the set of types; this observation is denoted as $\mathrm{dom}(\Gamma) \to T$.

Note We ignore the corner case where a single variable x occurs more than once in Γ. We are not going to use such contexts anyway. So when we write $\Gamma, x : T$, it is implied that $x \notin \mathrm{dom}(\Gamma)$.

The use of contexts then becomes clear — contexts will provide the missing type information for function arguments which have not yet been bound. That is, if $t_1 = \lambda x : T.\ t_2$, then x is free in t_2 (but bound in t_1). Its type is then determined from context. This leads to the type of non-closed terms depending on the context, making the typing relation a **ternary** one, instead of binary: it is now a relation between a context Γ, term t, and type τ. The accepted notation for the typing relation is:

$$\Gamma \vdash t : \tau$$

Formal definitions will come in a minute, but before that here are some more examples to give the intuition:

Γ	t	τ
$x : \mathrm{Bool}$	x	Bool
$x : \mathrm{Bool}$	$\lambda x : \mathrm{Bool} \to \mathrm{Bool}.\ \mathtt{true}$	$(\mathrm{Bool} \to \mathrm{Bool}) \to \mathrm{Bool}$
$x : \mathrm{Bool} \to \mathrm{Bool}$, $y : \mathrm{Bool}$	$x\ y$	Bool
$y : \mathrm{Bool} \to \mathrm{Bool}$	$\lambda x : \mathrm{Bool}.\ y\ (y\ x)$	$\mathrm{Bool} \to \mathrm{Bool}$

2.1 Type Rules

The *typing relation* $\Gamma \vdash t : T$ is defined by the following rules:

$$\Gamma \vdash \mathtt{true} : \mathrm{Bool} \qquad \text{(T-TRUE)}$$

$$\Gamma \vdash \mathtt{false} : \mathrm{Bool} \qquad \text{(T-FALSE)}$$

$$\begin{array}{l} \text{if } x : T \in \Gamma, \\ \quad \Gamma \vdash x : T \end{array} \qquad \text{(T-VAR)}$$

$$\frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash \lambda x.\ t :\ T_1 \to T_2} \qquad \text{(T-ABS)}$$

$$\frac{\Gamma \vdash t_1 : \mathrm{Bool} \quad \Gamma \vdash t_2 : T \quad \Gamma \vdash t_3 : T}{\Gamma \ \vdash\ \mathtt{if}\ t_1\ \mathtt{then}\ t_2\ \mathtt{else}\ t_3 \ :\ T} \qquad \text{(T-IF)}$$

$$\frac{\Gamma \vdash t_1 : T_1 \to T_2 \quad \Gamma \vdash t_2 : T_1}{\Gamma \ \vdash\ t_1\ t_2 \ :\ T_2} \qquad \text{(T-APP)}$$

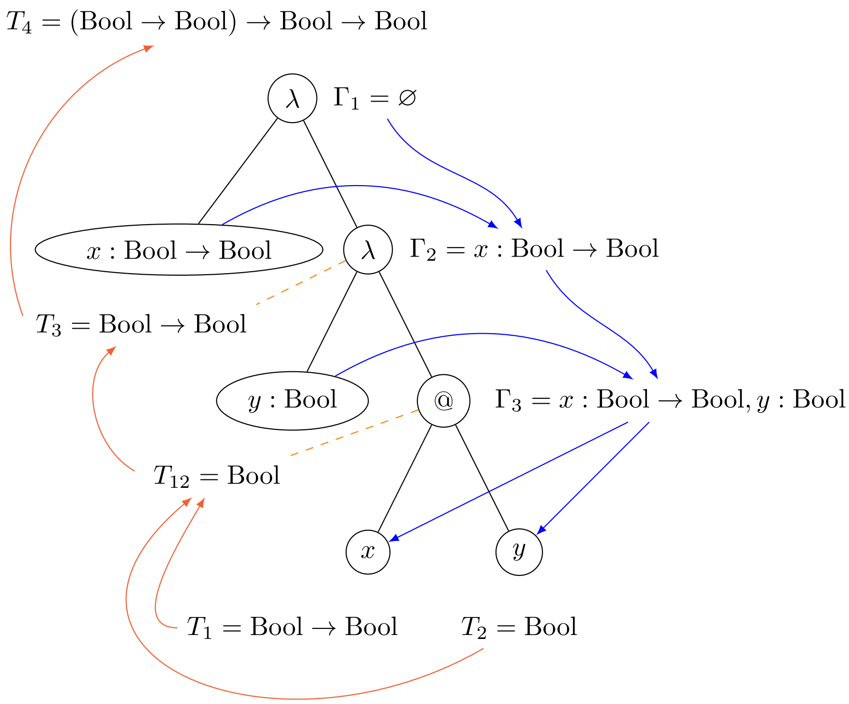

Figure 1: Type information flowing between sub-terms of $\lambda x : \text{Bool} \to \text{Bool}.\ \lambda y : \text{Bool}.\ x\ y$

Here is how we prove that $\lambda x : \text{Bool} \to \text{Bool}.\ \lambda y : \text{Bool}.\ x\ y\ :\ (\text{Bool} \to \text{Bool}) \to \text{Bool} \to \text{Bool}$ using the rules above.

$$\dfrac{\dfrac{\dfrac{\overset{\text{(T-VAR)}}{x : \text{Bool} \to \text{Bool}, y : \text{Bool} \vdash x : \text{Bool} \to \text{Bool}} \qquad \overset{\text{(T-VAR)}}{x : \text{Bool} \to \text{Bool}, y : \text{Bool} \vdash y : \text{Bool}}}{x : \text{Bool} \to \text{Bool}, y : \text{Bool} \vdash x\ y : \text{Bool}}\ \text{(T-APP)}}{x : \text{Bool} \to \text{Bool} \vdash (\lambda y : \text{Bool}.\ x\ y)\ :\ \text{Bool} \to \text{Bool}}\ \text{(T-ABS)}}{\vdash (\lambda x : \text{Bool} \to \text{Bool}.\ \lambda y : \text{Bool}.\ x\ y)\ :\ (\text{Bool} \to \text{Bool}) \to \text{Bool} \to \text{Bool}}\ \text{(T-ABS)}$$

Explanation From the types of the arguments we see that we have to prepare the context:

$$\Gamma_3 = x : \text{Bool} \to \text{Bool}, y : \text{Bool}$$

From the application rule T-APP we conclude that $\Gamma_3 \vdash x\ y$. T-APP never changes the context, but T-ABS does; when we compute the type of an abstraction, we remove the parameter from the context to receive the function type. Parameters are removed from the inside out: first we remove y, and get a new context $\Gamma_2 = x : \text{Bool} \to \text{Bool}$. By T-ABS, $\Gamma_2 \vdash (\lambda y : \text{Bool}.\ x\ y)\ :\ \text{Bool} \to \text{Bool}$. A second application of T-ABS eliminates x from context and the resulting type is $(\text{Bool} \to \text{Bool}) \to \text{Bool} \to \text{Bool}$. Note that the context at the end of the proof is empty, and that the term contains no free variables.

Figure 2.1 gives the intuition behind the proof by demonstrating how information passes between nodes of the expression tree. The blue arrows show the contexts being built, and the orange arrows denote use of the rules T-ABS and T-APP.

3 Types and Operational Semantics

The formal definition of structural operational semantics for untyped lambda- calculus no longer applies to the typed variant: the two do not even share the same language anymore, due to the introduction of a new term scheme and the abolishing of another ($\lambda x.\ t$ is replaced with $\lambda x : T.\ t$). Fortunately, this change is quite simple to overcome: in order to preserve the original semantics, just ignore the type at evaluation time.

Untyped	Typed
$\dfrac{t_1 \to \boldsymbol{t_1'}}{t_1\ t_2 \to t_1'\ t_2}$ (E-APP1)	$\dfrac{t_1 \to \boldsymbol{t_1'}}{t_1\ t_2 \to t_1'\ t_2}$ (E-APP1)
$\dfrac{t_1 \to \boldsymbol{t_1'}}{t_1\ t_2 \to t_1'\ t_2}$ (E-APP2)	$\dfrac{t_2 \to \boldsymbol{t_2'}}{v_1\ t_2 \to v_1'\ t_2}$ (E-APP2)
$(\lambda x.\ t)\ v \to [x \mapsto v]t$ (E-APPABS)	$(\lambda x : \mathbf{T}.\ t)\ v \to [x \mapsto v]t$ (E-APPABS)

To this should be added the derivation rules for the Bool-typed expressions, and in fact any other base type should we wish to support it.

Type Bool	
$\texttt{if true then } t_1 \texttt{ else } t_2 \to t_1$	(E-IF-THEN)
$\texttt{if false then } t_1 \texttt{ else } t_2 \to t_2$	(E-IF-ELSE)
$\dfrac{t_1 \to t_1'}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \to \texttt{if } t_1' \texttt{ then } t_2 \texttt{ else } t_3}$	(E-IF-COND)

The property that the operational semantics completely ignore types gives rise to an interesting concept involving programs:

Definition 3.1. Given a typed lambda-calculus term t, the *erasure* of t is the untyped lambda-calculus term obtained from t by removing all type annotations, that is, replacing every $\lambda x : T$ by λx.

Formally:

$$\begin{aligned}&\text{erase}(v) = v\\ &\text{erase}(t_1\ t_2) = \text{erase}(t_1)\ \text{erase}(t_2)\\ &\text{erase}(\lambda x : T.\ t) = \lambda x.\ \text{erase}(t)\end{aligned}$$

Of course $\text{erase}(t)$ is not even defined for terms containing extensions to the calculus (true/false/if, in the case of booleans); but when it is defined, we would expect it to mean the same thing as the original (typed) term.

Theorem 3.2 (Evaluation commutes with erasure)**.** *(see Figure 3).*

- *If $t \to t'$ (under the typed evaluation relation), then $erase(t) \to erase(t')$ (under the untyped evaluation relation).*
- *If $erase(t) = m$ and $m \to m'$ (under the untyped evaluation relation), then there is a simply typed term t' such that $t \to t'$ (under the typed evaluation relation) and $erase(t') = m'$.*

$$
\begin{tikzcd}
t \arrow[r, "\rightarrow"] \arrow[d, "\text{erase}"'] & t' \arrow[d, "\text{erase}"] \\
m \arrow[r, "\rightarrow"'] & m'
\end{tikzcd}
$$

Figure 2: Evaluation-erasure commutativity

Proof. Done by induction on the derivation tree for the evaluation relation corresponding to either $t \to t'$ or $m \to m'$. The interesting induction step is when a derivation utilizes the derivation rule E-APPABS, which is different between the typed and untyped variants. In this case, because evaluation completely ignores types, the untyped version of E-APPABS is applicable in exactly the same cases where the typed one is, so transforming a derivation in the typed calculus to a derivition in the untyped calculus is straightforward — just apply erase() to all the terms involved. $\square$

Furthermore, if you also substitute true/false/if-then-else with their untyped derived counterparts tru/fls/ite, you still preserve the semantics — assuming this time that the original term is well-typed. We will not prove this property but we will give an example.

Example 3.3.

$$\text{(untyped)} \quad maj \quad = \quad \lambda p.\ \lambda q.\ \lambda r.\ \mathtt{ite}\ p\ (\mathtt{ite}\ q\ \mathtt{tru}\ r)\ (\mathtt{ite}\ q\ r\ \mathtt{fls})$$

$$\text{(typed)} \quad \begin{aligned} maj^{(T)} \quad = \quad & \lambda p : \text{Bool}.\ \lambda q : \text{Bool}.\ \lambda r : \text{Bool}. \\ & \qquad \mathtt{if}\ p\ \mathtt{then}\ (\mathtt{if}\ q\ \mathtt{then}\ \mathtt{true}\ \mathtt{else}\ r) \\ & \qquad\quad \mathtt{else}\ (\mathtt{if}\ q\ \mathtt{then}\ r\ \mathtt{else}\ \mathtt{false}) \end{aligned}$$

The term *maj* represents a function that computes the majority of three boolean values. By typing rules, $maj^{(T)}$: Bool $\to$ Bool $\to$ Bool $\to$ Bool, so for every three **boolean values** b_1, b_2, b_3 we expect $maj\ b_1\ b_2\ b_3$ and $maj^{(T)}\ b_1\ b_2\ b_3$ to be consistent. Let's try with $b_1 = \mathtt{true}, b_2 = \mathtt{true}, b_3 = \mathtt{false}$.

In the untyped setting we will carry out the entire derivation with E-ABSAPP.

$$
\begin{aligned}
& maj\ \mathtt{tru}\ \mathtt{tru}\ \mathtt{fls} \\
& \quad \to (\lambda q.\ \lambda r.\ \mathtt{ite}\ \mathtt{tru}\ (\mathtt{ite}\ q\ \mathtt{tru}\ r)\ (\mathtt{ite}\ q\ r\ \mathtt{fls}))\ \mathtt{tru}\ \mathtt{fls} \\
& \quad \to (\lambda r.\ \mathtt{ite}\ \mathtt{tru}\ (\mathtt{ite}\ \mathtt{tru}\ \mathtt{tru}\ r)\ (\mathtt{ite}\ \mathtt{tru}\ r\ \mathtt{fls}))\ \mathtt{fls} \\
& \quad \to \mathtt{ite}\ \mathtt{tru}\ (\mathtt{ite}\ \mathtt{tru}\ \mathtt{tru}\ \mathtt{fls})\ (\mathtt{ite}\ \mathtt{tru}\ \mathtt{fls}\ \mathtt{fls}) \\
& \quad = (\lambda c.\ \lambda t.\ \lambda e.\ c\ t\ e)\ \mathtt{tru}\ (\mathtt{ite}\ \mathtt{tru}\ \mathtt{tru}\ \mathtt{fls})\ (\mathtt{ite}\ \mathtt{tru}\ \mathtt{fls}\ \mathtt{fls}) \\
& \quad \to (\lambda t.\ \lambda e.\ \mathtt{tru}\ t\ e)\ (\mathtt{ite}\ \mathtt{tru}\ \mathtt{tru}\ \mathtt{fls})\ (\mathtt{ite}\ \mathtt{tru}\ \mathtt{fls}\ \mathtt{fls}) \\
& \quad = (\lambda t.\ \lambda e.\ \mathtt{tru}\ t\ e)\ ((\lambda c.\ \lambda t.\ \lambda e.\ c\ t\ e)\ \mathtt{tru}\ \mathtt{tru}\ \mathtt{fls})\ (\mathtt{ite}\ \mathtt{tru}\ \mathtt{fls}\ \mathtt{fls}) \\
& \quad \to (\lambda t.\ \lambda e.\ \mathtt{tru}\ t\ e)\ ((\lambda t.\ \lambda e.\ \mathtt{tru}\ t\ e)\ \mathtt{tru}\ \mathtt{fls})\ (\mathtt{ite}\ \mathtt{tru}\ \mathtt{fls}\ \mathtt{fls}) \\
& \quad \to (\lambda t.\ \lambda e.\ \mathtt{tru}\ t\ e)\ ((\lambda e.\ \mathtt{tru}\ \mathtt{tru}\ e)\ \mathtt{fls})\ (\mathtt{ite}\ \mathtt{tru}\ \mathtt{fls}\ \mathtt{fls}) \\
& \quad \to (\lambda t.\ \lambda e.\ \mathtt{tru}\ t\ e)\ (\mathtt{tru}\ \mathtt{tru}\ \mathtt{fls})\ (\mathtt{ite}\ \mathtt{tru}\ \mathtt{fls}\ \mathtt{fls}) \\
& \quad = (\lambda t.\ \lambda e.\ \mathtt{tru}\ t\ e)\ ((\lambda t.\ \lambda f.\ t)\ \mathtt{tru}\ \mathtt{fls})\ (\mathtt{ite}\ \mathtt{tru}\ \mathtt{fls}\ \mathtt{fls}) \\
& \quad \to (\lambda t.\ \lambda e.\ \mathtt{tru}\ t\ e)\ ((\lambda f.\ \mathtt{tru})\ \mathtt{fls})\ (\mathtt{ite}\ \mathtt{tru}\ \mathtt{fls}\ \mathtt{fls}) \\
& \quad \to (\lambda t.\ \lambda e.\ \mathtt{tru}\ t\ e)\ \mathtt{tru}\ (\mathtt{ite}\ \mathtt{tru}\ \mathtt{fls}\ \mathtt{fls}) \\
& \quad \to \cdots \to (\lambda t.\ \lambda e.\ \mathtt{tru}\ t\ e)\ \mathtt{tru}\ \mathtt{fls} \to \cdots \to \mathtt{tru}
\end{aligned}
$$

In the typed frontier, we have more powerful rules, making the derivation somewhat shorter.

$$
\begin{aligned}
&maj^{(T)}\ \texttt{tru}\,\texttt{tru}\,\texttt{fls}\\
&\quad = \Big(\lambda p : \text{Bool}.\ \lambda q : \text{Bool}.\ \lambda r : \text{Bool}.\\
&\qquad\quad \texttt{if}\ p\ \texttt{then}\ (\texttt{if}\ q\ \texttt{then}\ \texttt{true}\ \texttt{else}\ r)\\
&\qquad\qquad\quad \texttt{else}\ (\texttt{if}\ q\ \texttt{then}\ r\ \texttt{else}\ \texttt{false})\Big)\ \texttt{true}\,\texttt{true}\,\texttt{false}\\
&\quad \to \Big(\lambda q : \text{Bool}.\ \lambda r : \text{Bool}.\\
&\qquad\quad \texttt{if}\ \texttt{true}\ \texttt{then}\ (\texttt{if}\ q\ \texttt{then}\ \texttt{true}\ \texttt{else}\ r)\\
&\qquad\qquad\quad \texttt{else}\ (\texttt{if}\ q\ \texttt{then}\ r\ \texttt{else}\ \texttt{false})\Big)\ \texttt{true}\,\texttt{false}\\
&\quad \to \Big(\lambda r : \text{Bool}.\\
&\qquad\quad \texttt{if}\ \texttt{true}\ \texttt{then}\ (\texttt{if}\ \texttt{true}\ \texttt{then}\ \texttt{true}\ \texttt{else}\ r)\\
&\qquad\qquad\quad \texttt{else}\ (\texttt{if}\ \texttt{true}\ \texttt{then}\ r\ \texttt{else}\ \texttt{false})\Big)\ \texttt{false}\\
&\quad \to \texttt{if}\ \texttt{true}\ \texttt{then}\ (\texttt{if}\ \texttt{true}\ \texttt{then}\ \texttt{true}\ \texttt{else}\ \texttt{false})\\
&\qquad\qquad\quad \texttt{else}\ (\texttt{if}\ \texttt{true}\ \texttt{then}\ \texttt{false}\ \texttt{else}\ \texttt{false})\\
&\quad \to \texttt{if}\ \texttt{true}\ \texttt{then}\ \texttt{true}\\
&\qquad\qquad\quad \texttt{else}\ (\texttt{if}\ \texttt{true}\ \texttt{then}\ \texttt{false}\ \texttt{else}\ \texttt{false})\\
&\quad \to \texttt{if}\ \texttt{true}\ \texttt{then}\ \texttt{true}\ \texttt{else}\ \texttt{false}\quad \to\quad \texttt{true}
\end{aligned}
$$

4 Properties of Typing

In this section we describe several properties of the typed lambda-calculus presented in previous section.

Inversion Lemma The inversion lemma describes several simple structure properties of typable terms. For each syntactic form, the lemma tells us: "if a term of this form is typable, then its subterms must have types of these forms . . .".

Lemma 4.1 (Inversion Lemma)**.** *1. If $\Gamma \vdash x : R$, then $x : R \in \Gamma$.*

2. If $\Gamma \vdash \lambda x : T_1.t_2 : R$, then $R = T_1 \to R_2$ for some R_2 with $\Gamma, x : T_1 \vdash t_2 : R_2$.

3. If $\Gamma \vdash t_1 t_2 : R$, then there is some type T_{11} such that $\Gamma \vdash t_1 : T_{11} \to R$ and $\Gamma \vdash t_2 : T_{11}$.

4. If $\Gamma \vdash \texttt{true} : R$, then $R = \texttt{Bool}$.

5. If $\Gamma \vdash \texttt{false} : R$, then $R = \texttt{Bool}$.

6. If $\Gamma \vdash \texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3 : R$, then $\Gamma \vdash t_1 : \texttt{Bool}$ and $\Gamma \vdash t2, t3 : R$.

Proof. Immediate from definitions. □

Example 4.2. *Consider the term $t = \texttt{if true then}\ (\lambda x : \texttt{Bool}.\ x)\ \texttt{else}\ (\lambda y : \texttt{Bool}.\ \texttt{false})$. The term t has the type $\texttt{Bool} \to \texttt{Bool}$. According to the inversion lemma (6) we know that the subterms $\lambda x : \texttt{Bool}.\ x$ and $\lambda y : \texttt{Bool}.\ \texttt{false}$ have the same type as t (i.e., their type is also $\texttt{Bool} \to \texttt{Bool}$).*

Lemma 4.1 is fundamental because it instructs a compiler designer how to build the type checker (see example implementation in Figure 4).

```haskell
import qualified Data.Map as M

-- Language Grammar

type Variable = String

data LType = Bool | Func LType LType deriving (Eq,Show)

data LExpr =
      Var Variable | Abs Variable LType LExpr | App LExpr LExpr
    | LTrue | LFalse | If LExpr LExpr LExpr        deriving Show

-- Type Checking Rules

typecheck ctx (Var j) =
    case M.lookup j ctx of
        Nothing -> undefined
        Just tp -> tp

typecheck ctx (Abs x tp t) = Func tp (typecheck (M.insert x tp ctx) t)

typecheck ctx (App t1 t2) =
    case (typecheck ctx t1, typecheck ctx t2) of
        (Func tp11 tp12, tp2) -> if tp11 == tp2 then tp12 else undefined
        _ -> undefined

typecheck ctx LTrue = Bool
typecheck ctx LFalse = Bool
typecheck ctx (If tcond tthen telse) =
    let tc = typecheck ctx
    case (tc tcond, tc tthen, tc telse) of
        (Bool, tp1, tp2) -> if tp1 == tp2 then tp1 else undefined
        _ -> undefined
```

Figure 3: Type checker for typed lambda-calculus implemented in Haskell

Uniqueness of Types In the presented typed lambda-calculus, every typable term has a unique type. Moreover, the typing derivation of a term t is uniquely defined by t (and vice versa). This is provided by the following theorem:

Theorem 4.3. *In a given typing context* Γ*, a term* t *has at most one type. If* t *is typable (in* Γ*), then there is just one derivation of this typing built from the inference rules that generate the typing relation.*

Proof. Let Γ be a typing context, and let t be a term. There is at most one typing rule that can be applied on t in Γ (immediate from the syntax and the typing rules). $\square$

Note that, this property does not hold for all type systems (for example, for some type systems with subtyping).

Safety The most basic property of type systems is *safety* — this property ensures that every typable term will never go wrong. Formally, a typable term can never reach (during evaluation) a term t_e such that: (i) t_e is not designated as a legal final value; (ii) there is no term t' such that $t_e \rightarrow t'$. This property is shown by the progress and preservation theorems.

- **Progress**: A typable term is either a value or it can take a step according to the evaluation rules.
- **Preservation**: If a typable term takes a step of evaluation, then the resulting term is also typable.

The following lemma is used to prove the progress theorem.

Lemma 4.4 (Canonical Forms)**.**

1. *If* v *is a value of type* `Bool`*, then* v *is either* `true` *or* `false`*.*
2. *If* v *is a value of type* $T_1 \rightarrow T_2$*, then* $v = \lambda x : T_1.t_2$*.*

Proof. Immediate from definitions. $\square$

The progress property does not hold for the shown type system, since it may fail on terms with free variables. For example, the term x *true* is a tyable term (in a context in which x is a function from `Bool`). However, this failure is not a problem since complete programs which are the terms we actually care about evaluating are always closed (i.e., have no free variables).

Theorem 4.5 (Progress)**.**
Suppose t *is a closed, typable term (that is,* $\vdash t : T$ *for some* T*). Then either* t *is a value or else there is some* t' *with* $t \rightarrow t'$*.*

Proof. Induction on typing derivations. If t is not a value then there are two cases to consider: (1) t is a variable, (2) $t = t_1 t_2$. Case (1) cannot occur because t is closed. In case (2), $t = t_1 t_2$ with $\vdash t_1 : T_{11} \rightarrow T_{12}$ and $\vdash t_2 : T_{11}$. By the induction hypothesis, either t_1 is a value or else it can make a step of evaluation, and likewise t_2. If t_1 can take a step, then rule E-App1 applies to t. If t_1 is a value and t_2 can take a step, then rule E-App2 applies. Finally, if both t_1 and t_2 are values, then the canonical forms lemma tells us that t_1 has to form $\lambda x : T_{11}.t_{12}$, and so rule E-AppAbs applies to t. $\square$

The following immediate lemmas are used to prove the preservation theorem:

Lemma 4.6 (Permutation)**.**
If $\Gamma \vdash t : T$ *and* Δ *is a permutation of* Γ*, then* $\Delta \vdash t : T$*. Moreover, the latter derivation has the same depth as the former.*

Lemma 4.7 (Weakening)**.**
If $\Gamma \vdash t : T$ *and* $x \notin dom(\Gamma)$*, then* $\Gamma, x : S \vdash t : T$*. Moreover, the latter derivation has the same depth as the former.*

These lemmas are used to prove an important property of the type system: a typable term remains typable when its variables are substituted with terms of appropriate types.

Lemma 4.8 (Preservation of types under substitution)**.**
If $\Gamma, x : S \vdash t : T$ *and* $\Gamma \vdash s : S$*, then* $\Gamma \vdash [x \mapsto s]t : T$*.*

Proof. By induction on the depth of a derivation of the statement $\Gamma, x : S \vdash t : T$ (using the depth of derivation, enables us to use the previous lemmas).
The only non-trivial case is the case in which t is an abstraction. In this case, we can assume:
$t = \lambda y : T_2.t_1$
$T = T_2 \to T_1$
$\Gamma, x : S, y : T_2 \vdash t_1 : T_1$
By convention we may assume that $x \neq y$ and $y \notin FV(s)$. Using permutation on the given subderivation, we obtain $\Gamma, y : T_2, x : S \vdash t_1 : T_1$. Using weakening on the other given derivation ($\Gamma \vdash s : S$), we obtain $\Gamma, y : T_2 \vdash s : S$. Now, by the induction hypothesis, $\Gamma, y : T_2 \vdash [x \mapsto s]t_1 : T_1$. By T-Abs, $\Gamma \vdash \lambda y : T_2.[x \mapsto s]t_1 : T_2 \to T_1$. this is precisely the needed result, since, by the definition of substitution, $[x \mapsto s]t = \lambda y : T_2.[x \mapsto s]t_1$. ☐

Using the substitution lemma, we can prove the preservation theorem.

Theorem 4.9 (Preservation)**.**
If $\Gamma \vdash t : T$ *and* $t \to t'$*, then* $\Gamma \vdash t' : T$*.*

Proof. By induction on a derivation of $\Gamma \vdash t : T$. The only non-trivial case is the case in which t is an application of abstraction — this case is implied from the previous lemma. ☐

Note that this theorem enables us to verify that all terms (that can be evaluated from t) have the same type as t. So, if the type of t is one of the designated (legal) types, then the type of the resulting value is one of the designated (legal) types.

Note also that, we only need a strong version of this theorem, since we only interested in evaluating closed terms.

5 Curry-Style vs. Church-Style

We have seen two different styles in which the semantics of the simply typed lambda-calculus can be formulated: as an evaluation relation defined directly on the syntax of the simply typed calculus, or as a compilation to an untyped calculus plus an evaluation relation on untyped terms. An important commonality of the two styles is that, in both, it makes sense to talk about the behavior of a term t, whether or not t is actually typable. This form of language definition is often called *Curry-style*. We first define the terms, then define a semantics showing how they behave, then give a type system that rejects some terms whose behaviors we do not like. **Semantics is prior to typing**.

A rather different way of organizing a language definition is to define terms, then identify the typable terms, then give semantics just to these. In these so-called *Church-style* systems, **typing is prior to semantics**: we only define the semantics of typable terms.

6 Extensions

The typed lambda calculus has several possible extensions. In class, we have mentioned the following extensions: Base Types, The Unit Type, Ascription, Let bindings, Pairs, Tuples, Records, Sums, Variants, General recursion, and Lists. The first three are described in this section.

Base Types Programming languages provide variety of base types — we saw typed lambda calculus with a type for booleans (`Bool`). Other types (like integers) can be added in the same way the `Bool` type has been added . For example, we can add a type for integers by using the following type grammar: $T ::= \ T \to T \mid \texttt{Bool} \mid \texttt{Int}$.

The Unit Type The `Unit` is a type with a single possible value — if $t : \texttt{Unit}$ then the value of t is the constant `unit` (written with a small u). This type is similar the `void` type in languages like C and Java. Its main application is for terms with side effects (e.g., assignments) for which we do not care about their return value.

It is often useful to evaluate two expressions in sequence without using the first expression's result — this is represented by the notation $t_1; t_2$.

There are actually two different ways to formalize sequencing. One is to follow the same pattern we have used for other syntactic forms: add $t_1; t_2$ as a new alternative in the syntax of terms, and then add two evaluation rules

$$\frac{t_1 \to t_1'}{t_1; t_2 \to t_1'; t_2} \qquad \text{(E-Seq)}$$

$$\texttt{unit}; t_2 \to t_2 \qquad \text{(E-SeqNext)}$$

and a typing rule

$$\frac{\Gamma \vdash t_1 : \texttt{Unit} \quad \Gamma \vdash t_2 : T_2}{\Gamma \vdash t_1; t_2 : T_2} \qquad \text{(T-Seq)}$$

capturing the intended behavior of ;.

An alternative way of formalizing sequencing is simply to regard $t_1; t_2$ as an abbreviation for the term $(\lambda x : \texttt{Unit}.t_2)t_1$, where the variable x is different from all the free variables of t_2. The later approach is called "derived form" or "syntactic sugar". In a sense, such approach is better becuase it does require changing the type system.

Ascription Another extension enables to explicitly ascribe a particular type to a given term. We write t `as` T to ascribe type T to the term t, and the type system verifies that T is the type of t. The evaluation of t `as` T is not affected from the ascription (i.e., "`as` T" is ignored).

Ascription is used for documentation and to introduce abbreviations for long or complex type expressions. For example, the declaration

$$\texttt{UU} = \texttt{Unit} \to \texttt{Unit};$$

makes `UU` an abbreviation for $\texttt{Unit} \to \texttt{Unit}$ in what follows. For example, the following term

$$(\lambda f : \texttt{UU}.f \ \texttt{unit})(\lambda x : \texttt{Unit}.x);$$

is equivalant (from the point of view of the type system) to the term

$$(\lambda f : \texttt{Unit} \to \texttt{Unit}.f \ \texttt{unit})(\lambda x : \texttt{Unit}.x);$$

Ascription can be added by adding the syntactic form $t\texttt{as}T$, and the new evaluation rules:

$$v_1 \ \texttt{as} \ T \to v_1 \qquad \text{(E-Ascribe)}$$

$$\frac{t_1 \to t_1'}{t_1 \ \texttt{as} \ T \to t_1' \ \texttt{as} \ T} \qquad \text{(E-Ascribe1)}$$

and the new typing rule:

$$\frac{\Gamma \vdash t_1 : T}{\Gamma \vdash t_1 \ \texttt{as} \ T : T} \qquad \text{(T-Ascribe)}$$