CS 4110 – Programming Languages and Logics
Lecture #25: Simply-Typed Lambda Calculus

A *type* is a collection of computational entities that share some common property. For example, the type **int** represents all expressions that evaluate to an integer, and the type **int** $\rightarrow$ **int** represents all functions from integers to integers. The Pascal subrange type `[1..100]` represents all integers between 1 and 100.

You can see types as a *static* approximation of the *dynamic* behaviors of terms and programs. Type *systems* are a lightweight formal method for reasoning about behavior of a program. Uses of type systems include: naming and organizing useful concepts; providing information (to the compiler or programmer) about data manipulated by a program; and ensuring that the run-time behavior of programs meet certain criteria.

In this lecture, we'll consider a type system for the lambda calculus that ensures that values are used correctly; for example, that a program never tries to add an integer to a function. The resulting language (lambda calculus plus the type system) is called the *simply-typed lambda calculus* (STLC).

1 Simply-typed lambda calculus

The syntax of the simply-typed lambda calculus is similar to that of untyped lambda calculus, with the exception of abstractions. Since abstractions define functions tht take an argument, in the simply-typed lambda calculus, we explicitly state what the type of the argument is. That is, in an abstraction $\lambda x\!:\!\tau.\, e$, the τ is the expected type of the argument.

The syntax of the simply-typed lambda calculus is as follows. It includes integer literals n, addition $e_1 + e_2$, and the *unit value* $()$. The unit value is the only value of type **unit**.

$$
\begin{array}{ll}
\text{expressions} & e ::= x \mid \lambda x\!:\!\tau.\, e \mid e_1\ e_2 \mid n \mid e_1 + e_2 \mid () \\
\text{values} & v ::= \lambda x\!:\!\tau.\, e \mid n \mid () \\
\text{types} & \tau ::= \textbf{int} \mid \textbf{unit} \mid \tau_1 \rightarrow \tau_2
\end{array}
$$

The operational semantics of the simply-typed lambda calculus are the same as the untyped lambda calculus. For completeness, we present the CBV small step operational semantics here.

$$E ::= [\cdot] \mid E\ e \mid v\ E \mid E + e \mid v + E \qquad \textsc{Context}\ \frac{e \longrightarrow e'}{E[e] \longrightarrow E[e']}$$

$$\beta\textsc{-reduction}\ \frac{}{(\lambda x\!:\!\tau.\, e)\ v \longrightarrow e\{v/x\}} \qquad \textsc{Add}\ \frac{}{n_1 + n_2 \longrightarrow n}\ n = n_1 + n_2$$

1.1 The typing relation

The presence of types does not alter the evaluation of an expression at all. So what use are types?

We will use types to restrict what expressions we will evaluate. Specifically, the type system for the simply-typed lambda calculus will ensure that any *well-typed* program will not get *stuck*. A term e is stuck if e is not a value and there is no term e' such that $e \longrightarrow e'$. For example, the expression $42 + \lambda x.\, x$ is stuck: it attempts to add an integer and a function; it is not a value, and there is no operational rule that allows us to reduce this expression. Another stuck expression is $()\ 47$, which attempts to apply the unit value to an integer.

We introduce a relation (or *judgment*) over *typing contexts* (or *type environments*) Γ, expressions e, and types τ. The judgment

$$\Gamma \vdash e : \tau$$

is read as "e has type τ in context Γ".

A typing context is a sequence of variables and their types. In the typing judgment $\Gamma \vdash e : \tau$, we will ensure that if x is a free variable of e, then Γ associates x with a type. We can view a typing context as a partial function from variables to types. We will write $\Gamma, x : \tau$ or $\Gamma[x \mapsto \tau]$ to indicate the typing context that extends Γ by associating variable x with with type τ. The empty context is sometimes written $\emptyset$, or often just not written at all. For example, we write $\vdash e : \tau$ to mean that the closed term e has type τ under the empty context.

Given a typing environment Γ and expression e, if there is some τ such that $\Gamma \vdash e : \tau$, we say that *e is well-typed under context* Γ; if Γ is the empty context, we say *e is well-typed*.

We define the judgment $\Gamma \vdash e : \tau$ inductively.

$$\textsc{T-Int}\ \frac{}{\Gamma \vdash n : \mathbf{int}} \qquad \textsc{T-Add}\ \frac{\Gamma \vdash e_1 : \mathbf{int} \quad \Gamma \vdash e_2 : \mathbf{int}}{\Gamma \vdash e_1 + e_2 : \mathbf{int}} \qquad \textsc{T-Unit}\ \frac{}{\Gamma \vdash () : \mathbf{unit}}$$

$$\textsc{T-Var}\ \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \textsc{T-Abs}\ \frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x : \tau.\, e : \tau \to \tau'} \qquad \textsc{T-App}\ \frac{\Gamma \vdash e_1 : \tau \to \tau' \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\ e_2 : \tau'}$$

An integer n always has type **int**. Expression $e_1 + e_2$ has type **int** if both e_1 and e_2 have type **int**. The unit value () always has type **unit**.

Variable x has whatever type the context associates with x. Note that Γ must contain an association for x in order for the judgment $\Gamma \vdash x : \tau$ to hold, that is, $x \in dom(\Gamma)$. The abstraction $\lambda x : \tau.\, e$ has the function type $\tau \to \tau'$ if the function body e has type τ' under the assumption that x has type τ. Finally, an application $e_1\ e_2$ has type τ' provided that e_1 is a function of type $\tau \to \tau'$, and e_2 is an argument of the expected type, i.e., of type τ.

To type check an expression e, we attempt to construct a derivation of the judgment $\vdash e : \tau$, for some type τ. For example, consider the program $(\lambda x : \mathbf{int}.\, x + 40)\ 2$. The following is a proof that $(\lambda x : \mathbf{int}.\, x + 40)\ 2$ is well-typed.

$$\textsc{T-App}\ \dfrac{\textsc{T-Abs}\ \dfrac{\textsc{T-Add}\ \dfrac{\textsc{T-Var}\ \dfrac{}{x : \mathbf{int} \vdash x : \mathbf{int}} \quad \textsc{T-Int}\ \dfrac{}{x : \mathbf{int} \vdash 40 : \mathbf{int}}}{x : \mathbf{int} \vdash x + 40 : \mathbf{int}}}{\vdash \lambda x : \mathbf{int}.\, x + 40 : \mathbf{int} \to \mathbf{int}} \quad \textsc{T-Int}\ \dfrac{}{\vdash 2 : \mathbf{int}}}{\vdash (\lambda x : \mathbf{int}.\, x + 40)\ 2 : \mathbf{int}}$$

1.2 Type soundness

We mentioned above that the type system ensures that any well-typed program does not get stuck. We can state this property formally.

Theorem (Type soundness)**.** *If* $\vdash e : \tau$ *and* $e \longrightarrow^* e'$ *and* $e' \not\longrightarrow$ *then* e' *is a value and* $\vdash e' : \tau$*.*

We will prove this theorem using two lemmas: *preservation* and *progress*. Intuitively, preservation says that if an expression e is well-typed, and e can take a step to e', then e' is well-typed. That is, evaluation preserves well-typedness. Progress says that if an expression e is well-typed, then either e is a value, or there is an e' such that e can take a step to e'. That is, well-typedness means that the expression cannot get stuck.

Together, these two lemmas suffice to prove type soundness. Given the preservation lemma, a trivial induction on the number of steps taken to reach e' from e establishes that $\vdash e' : \tau$. Then the progress lemma ensures that, if e' cannot take a step, then it must be a value.

1.3 Preservation

To prove preservation, we will need some extra tiny lemmas.

Lemma (Substitution)**.** *If* $x : \tau' \vdash e : \tau$ *and* $\vdash v : \tau'$ *then* $\vdash e\{v/x\} : \tau$*.*

Lemma (Context)**.** *If* $\vdash E[e] : \tau$ *and* $\vdash e : \tau'$ *and* $\vdash e' : \tau'$ *then* $\vdash E[e'] : \tau$*.*

We'll assume these without proof. (They're not difficult, but the proof of substitution can get rather long.) Equipped with these little lemmas, we're ready to move on to the main proof of preservation.

A quick note on proof strategy: to prove preservation, it's possible to induct either on the typing relation or on the small-step relation. Both have their advantages and disadvantages; we'll use the small-step relation here.

Lemma (Preservation)**.** *If* $\vdash e : \tau$ *and* $e \longrightarrow e'$ *then* $\vdash e' : \tau$*.*

Proof. Assume $\vdash e : \tau$ and $e \longrightarrow e'$. We need to show $\vdash e' : \tau$. We will do this by induction on the derivation of $e \longrightarrow e'$.

- Add

  Here, $e \equiv n_1 + n_2$, and $e' = n$ where $n = n_1 + n_2$.

  There is only one typing rule that applies to addition expressions, T-Add, from which we know $\tau = \textbf{int}$.

  By the typing rule T-Int, we have $\vdash e' : \textbf{int}$ as required.

- β-reduction

  Here, $e \equiv (\lambda x : \tau'.\, e_1)\; v$ and $e' \equiv e_1\{v/x\}$.

  Since e is well-typed by assumption, we have derivations showing $\vdash \lambda x : \tau'.\, e_1 : \tau' \rightarrow \tau$ and $\vdash v : \tau'$. There is only one typing rule for abstractions, T-Abs, from which we know $x : \tau' \vdash e_1 : \tau$.

  By our substitution lemma above, we have $\vdash e_1\{v/x\} : \tau$ as required.

- CONTEXT

  Here, we have some context E such that $e = E[e_1]$ and $e' = E[e_2]$ for some e_1 and e_2 such that $e_1 \longrightarrow e_2$.

  Since e is well-typed, we can show by induction on the structure of E that $\vdash e_1 : \tau_1$ for some τ_1. (This simple sub-induction is left as an exercise.)

  By the induction hypothesis and because we know $e_1 \longrightarrow e_2$, we have $\vdash e_2 : \tau_1$. (Put intuitively, e_2 has the same type as the one we just established for e_1.)

  By our context lemma above, we have $\vdash E[e_2] : \tau$ as required.

$\square$

1.4 Progress

To prove our progress lemma, we'll need one extra lemma that gives us the syntax forms for closed terms.

Lemma (Canonical Forms)**.** *If* $\vdash v : \tau$*, then*

1. *If* τ *is* ***int****, then* v *is a constant, i.e., some* c*.*
2. *If* τ *is* $\tau_1 \longrightarrow \tau_2$*, then* v *is an abstraction, i.e.,* $\lambda x : \tau_1 .\, e$ *for some* x *and* e*.*

Proof. The proof is by inspection of the typing rules.

i If τ is int, then the only rule which lets us give a value this type is T-INT.

ii If τ is $\tau_1 \longrightarrow \tau_2$, then the only rule which lets us give a value this type is T-ABS.

$\square$

Now we're ready to prove progress.

Lemma (Progress)**.** *If* $\vdash e : \tau$ *then either* e *is a value or there exists an* e' *such that* $e \longrightarrow e'$*.*

Proof. We proceed by induction on the derivation of $\vdash e : \tau$.

- T-VAR

  This case is impossible, since a variable is not well-typed in the empty environment.

- T-UNIT, T-INT, T-ABS

  In all of these cases, e is a value.

- T-ADD

  Here $e \equiv e_1 + e_2$ and $\vdash e_1 : \textbf{int}$ and $\vdash e_2 : \textbf{int}$. By the inductive hypothesis, for $i \in \{1, 2\}$ (i.e., for both e_1 and e_2), either e_i is a value or there is an e_i' such that $e_i \longrightarrow e_i'$.

  If e_1 is not a value, we have from above that $e_1 \longrightarrow e_1'$. Therefore, by the CONTEXT rule, $e_1 + e_2 \longrightarrow e_1' + e_2$.

  Otherwise, e_1 is a value. If e_2 is not a value, then by CONTEXT again, $e_1 + e_2 \longrightarrow e_1 + e_2'$.

  Otherwise, both e_1 and e_2 are values. By our canonical forms lemma, $e_1 = n_1$ and $e_2 = n_2$ are both integer literals. By the ADD rule, we have $e_1 + e_2 \longrightarrow n$ where $n = n_1 + n_2$.

- T-App

  Here $e \equiv e_1\ e_2$ and $\vdash e_1 : \tau' \to \tau$ and $\vdash e_2 : \tau'$. By the inductive hypothesis, for $i \in \{1, 2\}$, either e_i is a value or there is an e_i' such that $e_i \to e_i'$.

  If e_1 is not a value, then by the above and by applying the Context rule, $e_1\ e_2 \to e_1'\ e_2$.

  Otherwise, e_1 is a value. If e_2 is not a value, then by Context, $e_1\ e_2 \to e_1\ e_2'$.

  If e_1 and e_2 are values, then, by our canonical forms lemma, e_1 is an abstraction $\lambda x : \tau'.\, e'$. Therefore, by β-reduction, we have $e_1\ e_2 \to e'\{e_2/x\}$.

□

2 Type Completeness?

Not all expressions in the untyped lambda calculus are well-typed. Type soundness implies that any lambda calculus program that gets stuck is not well-typed.

But are there programs that *do not* get stuck that are not well-typed? In other words, does our type system unjustly rule out legal programs?

Unfortunately, the answer is yes. In particular, because the simply-typed lambda calculus requires us to specify a type for function arguments, any given function can only take arguments of one type. Consider, for example, the identity function $\lambda x.\, x$. This function may be applied to any argument, and it will not get stuck. However, we must provide a type for the argument. If we specify $\lambda x : \textbf{int}.\, x$, then this function can only accept integers, and the program $(\lambda x : \textbf{int}.\, x)\ ()$ is not well-typed, even though it does not get stuck. Indeed, in the simply-typed lambda calculus, there is a different identity function for each type.