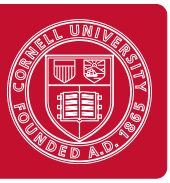
**CS 4110 – Programming Languages and Logics**
**Lecture #19: Simply Typed $\lambda$-calculus**



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** $\to$ **int** represents all functions from integers to integers. The Pascal subrange type `[1..100]` represents all integers between 1 and 100.

Types can be thought of as describing computations succinctly and approximately: types are a *static* approximation to the run-time 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*.

## 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 $\tau$ 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**.

| | |
|---|---|
| expressions | $e ::= x \mid \lambda x\!:\!\tau.\, e \mid e_1\ e_2 \mid n \mid e_1 + e_2 \mid ()$ |
| values | $v ::= \lambda x\!:\!\tau.\, e \mid n \mid ()$ |
| types | $\tau ::= \textbf{int} \mid \textbf{unit} \mid \tau_1 \to \tau_2$ |

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.

$$E ::= [\cdot] \mid E\ e \mid v\ E \mid E + e \mid v + E$$

$$\textsc{Context}\ \frac{e \to e'}{E[e] \to E[e']}$$

$$\beta\textsc{-reduction}\ \frac{}{(\lambda x\!:\!\tau.\, e)\ v \to e\{v/x\}}$$

$$\textsc{Add}\ \frac{}{n_1 + n_2 \to n}\ n = n_1 + n_2$$

### 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 $e \to e'$. 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*) $\Gamma$, expressions $e$, and types $\tau$. The judgment

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

is read as "$e$ has type $\tau$ in context $\Gamma$".

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 $\Gamma$ 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 $\Gamma$ by associating variable $x$ with with type $\tau$. 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 $\tau$ under the empty context.

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

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

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

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

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 $\Gamma$ must contain an associating for $x$ in order to 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 $\tau'$ under the assumption that $x$ has type $\tau$. Finally, an application $e_1\ e_2$ has type $\tau'$ 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 $\tau$.

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

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

## 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* $e \to^* e'$ *and* $e' \not\to$ *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.

### 1.2.1 Preservation

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

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

Consider the last rule used in the derivation of $e \to e'$.

- ADD

  Here $e \equiv n_1 + n_2$, and $e' = n$ where $n = n_1 + n_2$, and $\tau = \textsf{int}$. By the typing rule T-INT, we have $\vdash e' : \textsf{int}$ as required.

- $\beta$-REDUCTION

  Here, $e \equiv (\lambda x : \tau'.\, e_1)\ v$ and $e' \equiv e_1\{v/x\}$. Since $e$ is well-typed, we have derivations showing $\vdash \lambda x : \tau'.\, e_1 : \tau' \to \tau$ 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 the substitution lemma (see below), 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 $e_1 \to e_2$. Since $e$ is well-typed, we can show by induction on the structure of $E$ that $\vdash e_1 : \tau_1$ for some $\tau_1$. By the inductive hypothesis, we thus have $\vdash e_2 : \tau_1$. By the context lemma (see below) we have $\vdash E[e'] : \tau$ as required.

□

Additional lemmas we used in the proof above.

**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* $\vdash E[e'] : \tau$.

### 1.2.2 Progress

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

*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

  Trivial, since $e$ must be a value.

- T-Add

  Here $e \equiv e_1 + e_2$ and $\vdash e_i : \textsf{int}$ for $i \in \{1, 2\}$. 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 Context, $e_1 + e_2 \to e_1' + e_2$. If $e_1$ is a value and $e_2$ 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, it must be the case that they are both integer literals, and so, by Add, we have $e_1 + e_2 \to n$ where $n$ equals $e_1$ plus $e_2$.

- T-App

  Here $e \equiv e_1\ e_2$ and $\vdash e_1 : \tau' \to \tau$ and $\vdash e_1 : \tau'$. 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 Context, $e_1\ e_2 \to e_1'\ e_2$. If $e_1$ is a value and $e_2$ 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, it must be the case that $e_1$ is an abstraction $\lambda x : \tau'.\ e'$, and so, by $\beta$-reduction, we have $e_1\ e_2 \to e'\{e_2/x\}$.

□

Clearly, not all expressions in the untyped lambda calculus are well-typed. Indeed, 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? 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 $\lambda x : \textsf{int}.\ x$, then this function can only accept integers, and the program $(\lambda x : \textsf{int}.\ x)\ ()$ 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.