Harvard School of Engineering and Applied Sciences — CS 152: Programming Languages

Notes on Typed Lambda Calculi

Spring 2017

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 that 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. We will include 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.

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

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

$$\beta\text{-}\textsc{reduction}\ \frac{}{(\lambda x.\, e)\ v \longrightarrow e\{v/x\}}$$

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

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 $42 + \lambda x : \textbf{int}.\, x$ 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 $\varnothing$, 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.

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

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

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 Γ must contain an associating for x in order to the judgment $\Gamma \vdash x : \tau$ to hold, that is, $x \in \mathrm{dom}(\Gamma)$. 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.

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

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 $e \longrightarrow^* e'$ then either e' is a value, or there exists e'' such that $e' \longrightarrow e''$.*

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 $e \longrightarrow e'$ then $\vdash e' : \tau$.*

Proof. We proceed by induction on $e \longrightarrow e'$. That is, we will prove for all e and e' such that $e \longrightarrow e'$, that $P(e \longrightarrow e')$ holds, where

$$P(e \longrightarrow e') = \forall \tau.\ \text{if } \vdash e : \tau \text{ then } \vdash e' : \tau.$$

Consider each of the inference rules for the small step relation.

- Add

  Assume $\vdash e : \tau$.

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

- β-reduction

  Assume $\vdash e : \tau$.

  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

  Assume $\vdash e : \tau$.

  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 $e_1 \longrightarrow e_2$. The inductive hypothesis is that $P(e_1 \longrightarrow e_2)$.

  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. 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 $\vdash E[e_0] : \tau$ and $\vdash e_0 : \tau'$ and $\vdash e_1 : \tau'$ then $\vdash E[e_1] : \tau$.*

1.2.2 Progress

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

Proof. We proceed by induction on the derivation of $\vdash e : \tau$. That is, we will show for all e and τ such that $\vdash e : \tau$, we have $P(\vdash e : \tau)$, where

$$P(\vdash e : \tau) = \text{either } e \text{ is a value or } \exists e' \text{ such that } e \longrightarrow e'.$$

- 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 $\vdash e_i : \textbf{int}$ 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 $e_i \longrightarrow e_i'$.

  If e_1 is not a value, then by Context, $e_1 + e_2 \longrightarrow e_1' + e_2$. If e_1 is a value and e_2 is not a value, then by Context, $e_1 + e_2 \longrightarrow e_1 + e_2'$. 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 $e_1 + e_2 \longrightarrow n$ 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 $\vdash e_2 : \tau'$. By the inductive hypothesis, for $i \in \{1, 2\}$, either e_i is a value or there is an e_i' such that $e_i \longrightarrow e_i'$.

  If e_1 is not a value, then by Context, $e_1\ e_2 \longrightarrow e_1'\ e_2$. If e_1 is a value and e_2 is not a value, then by Context, $e_1\ e_2 \longrightarrow e_1\ e_2'$. 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 β-reduction, we have $e_1\ e_2 \longrightarrow e'\{e_2/x\}$.

□

1.3 Expressive power of the simply-typed lambda calculus

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.

First, since 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 $\lambda x : \mathbf{int}.\, x$, then this function can only accept integers, and the program $(\lambda x : \mathbf{int}.\, x)\ ()$ 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.

Second, we can no longer write recursive functions. Consider the nonterminating expression $\Omega = (\lambda x.\, x\ x)\ (\lambda x.\, x\ x)$. What type does it have? Let's suppose that the type of $\lambda x.\, x\ x$ is $\tau \to \tau'$. But $\lambda x.\, x\ x$ is applied to itself! So that means that the type of $\lambda x.\, x\ x$ is the argument type τ. So we have that τ must be equal to $\tau \to \tau'$. There is no such type for which this equality holds. (At least, not in this type system...)

This means that every well-typed program in the simply-typed lambda calculus terminates. More formally:

Theorem 1 (Normalization)**.** *If $\vdash e : \tau$ then there exists a value v such that $e \longrightarrow^* v$.*

This is known as *normalization* since it means that given any well-typed expression, we can reduce it to a *normal form*, which, in our case, is a value.

2 Products and Sums

We have previously seen *products*, which are pairs of expressions. Products were constructed using the expression (e_1, e_2), and destructed using projection $\#1\ e$ and $\#2\ e$.

$$e ::= \cdots \mid (e_1, e_2) \mid \#1\ e \mid \#2\ e$$
$$v ::= \cdots \mid (v_1, v_2)$$

Again, there are structural rules to determine the order of evaluation. In a CBV lambda calculus, the evaluation contexts are extended as follows.

$$E ::= \cdots \mid (E, e) \mid (v, E) \mid \#1\ E \mid \#2\ E$$

In addition to the structural rules, there are two operational semantics rules that show how the destructors and constructor interact.

$$\frac{}{\#1\ (v_1, v_2) \longrightarrow v_1} \qquad \frac{}{\#2\ (v_1, v_2) \longrightarrow v_2}$$

The type of a product expression (or a *product type*) is a pair of types, written $\tau_1 \times \tau_2$. The typing rules for the product constructors and destructors are the following.

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \#1\ e : \tau_1} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \#2\ e : \tau_2}$$

We introduce *sums*, which are dual to products. Intuitively, a product holds two values, one of type τ_1, and one of type τ_2. By contrast, a sum holds a single value that is either of type τ_1 or of type τ_2. The type of a sum is written $\tau_1 + \tau_2$. There are two constructors for a sum, corresponding to whether we are constructing a sum with a value of τ_1 or a value of τ_2.

$$\begin{aligned} e &::= \cdots \mid \mathsf{inl}_{\tau_1+\tau_2}\ e \mid \mathsf{inr}_{\tau_1+\tau_2}\ e \mid \mathsf{case}\ e_1\ \mathsf{of}\ e_2 \mid e_3 \\ v &::= \cdots \mid \mathsf{inl}_{\tau_1+\tau_2}\ v \mid \mathsf{inr}_{\tau_1+\tau_2}\ v \end{aligned}$$

Again, there are structural rules to determine the order of evaluation. In a CBV lambda calculus, the evaluation contexts are extended as follows.

$$E ::= \cdots \mid \mathsf{inl}_{\tau_1+\tau_2}\ E \mid \mathsf{inr}_{\tau_1+\tau_2}\ E \mid \mathsf{case}\ E\ \mathsf{of}\ e_2 \mid e_3$$

In addition to the structural rules, there are two operational semantics rules that show how the destructors and constructors interact.

$$\frac{}{\mathsf{case}\ \mathsf{inl}_{\tau_1+\tau_2}\ v\ \mathsf{of}\ e_2 \mid e_3 \longrightarrow e_2\ v} \qquad \frac{}{\mathsf{case}\ \mathsf{inr}_{\tau_1+\tau_2}\ v\ \mathsf{of}\ e_2 \mid e_3 \longrightarrow e_3\ v}$$

The type of a sum expression (or a *sum type*) is written $\tau_1 + \tau_2$. The typing rules for the sum constructors and destructor are the following.

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \mathsf{inl}_{\tau_1+\tau_2}\ e : \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \mathsf{inr}_{\tau_1+\tau_2}\ e : \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash e : \tau_1 + \tau_2 \quad \Gamma \vdash e_1 : \tau_1 \to \tau \quad \Gamma \vdash e_2 : \tau_2 \to \tau}{\Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ e_1 \mid e_2 : \tau}$$

Let's see an example of a program that uses sum types.

$$\begin{aligned} &\mathsf{let}\ f : (\mathbf{int} + (\mathbf{int} \to \mathbf{int})) \to \mathbf{int} = \\ &\qquad \lambda a : \mathbf{int} + (\mathbf{int} \to \mathbf{int}).\ \mathsf{case}\ a\ \mathsf{of}\ \lambda y.\ y + 1 \mid \lambda g.\ g\ 35\ \mathsf{in} \\ &\mathsf{let}\ h : \mathbf{int} \to \mathbf{int} = \lambda x : \mathbf{int}.\ x + 7\ \mathsf{in} \\ &f\ (\mathsf{inr}_{\mathbf{int}+(\mathbf{int}\to\mathbf{int})}\ h) \end{aligned}$$

Here, the function f takes argument a, which is a sum. That is, the actual argument for a will either be a value of type **int** or a value of type **int** → **int**. We destroy the sum value with a case statement, which must be prepared to take either of the two kinds of values that the sum may contain. We end up applying f to a value of type **int** → **int** (i.e., a value injected into the right type of the sum). The entire program ends up evaluating to 42.

3 Recursion

We saw in last lecture that we could not type recursive functions or fixed-point combinators in the simply-typed lambda calculus. So instead of trying (and failing) to define a fixed-point combinator in the simply-typed lambda calculus, we add a new primitive $\mu x : \tau.\ e$ to the language. The evaluation rules for the new primitive will mimic the behavior of fixed-point combinators.

We extend the syntax with the new primitive operator. Intuitively, $\mu x : \tau.\ e$ is the fixed-point of the function $\lambda x : \tau.\, e$. Note that $\mu x : \tau.\ e$ is *not* a value, regardless of whether e is a value or not.

$$e ::= \cdots \mid \mu x : \tau.\ e$$

We extend the operational semantics for the new operator. There is a new axiom, but no new evaluation contexts.

$$\frac{}{\mu x{:}\tau.\ e \longrightarrow e\{(\mu x{:}\tau.\ e)/x\}}$$

Note that we can define the $\mathsf{letrec}\ x{:}\tau = e_1\ \mathsf{in}\ e_2$ construct in terms of this new expression.

$$\mathsf{letrec}\ x{:}\tau = e_1\ \mathsf{in}\ e_2 \triangleq \mathsf{let}\ x{:}\tau = \mu x{:}\tau.\ e_1\ \mathsf{in}\ e_2$$

We add a new typing rule for the new language construct.

$$\frac{\Gamma[x \mapsto \tau] \vdash e{:}\tau}{\Gamma \vdash \mu x{:}\tau.\ e{:}\tau}$$

Returning to our trusty factorial example, the following program implements the factorial function using the $\mu x{:}\tau.\ e$ expression.

$$FACT \triangleq \mu f{:}\mathbf{int} \to \mathbf{int}.\ \lambda n{:}\mathbf{int}.\ \mathsf{if}\ n = 0\ \mathsf{then}\ 1\ \mathsf{else}\ n \times (f\ (n-1))$$

Or using our convenient letrec notation, we could define a variable *fact* as follows.

$$\begin{array}{l}\mathsf{letrec}\ \mathit{fact}{:}\mathbf{int} \to \mathbf{int} = \lambda n{:}\mathbf{int}.\ \mathsf{if}\ n = 0\ \mathsf{then}\ 1\ \mathsf{else}\ n \times (\mathit{fact}\ (n-1)) \\ \mathsf{in}\ \ldots\end{array}$$

We can write non-terminating computations for any type: the expression $\mu x{:}\tau.\ x$ has type τ, and does not terminate.

Although the $\mu x{:}\tau.\ e$ expression is normally used to define recursive functions, it can be used to find fixed points of any type. For example, consider the following expression.

$$\begin{array}{r}\mu x{:}(\mathbf{int} \to \mathbf{bool}) \times (\mathbf{int} \to \mathbf{bool}).\ (\lambda n{:}\mathbf{int}.\ \mathsf{if}\ n = 0\ \mathsf{then}\ \mathsf{true}\ \mathsf{else}\ ((\#2\ x)\ (n-1)), \\ \lambda n{:}\mathbf{int}.\ \mathsf{if}\ n = 0\ \mathsf{then}\ \mathsf{false}\ \mathsf{else}\ ((\#1\ x)\ (n-1)))\end{array}$$

This expression has type $(\mathbf{int} \to \mathbf{bool}) \times (\mathbf{int} \to \mathbf{bool})$—it is a pair of mutually recursive functions; the first function returns true only if its argument is even; the second function returns true only if its argument is odd.