Simply-typed lambda calculus

CS 152 (Spring 2022)

Harvard University

Tuesday, March 1, 2022

Today, we will learn about

- Simply-typed lambda calculus
- Type soundness
- Normalization

Types

- 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.

Types

Type systems are a lightweight formal method for reasoning about behavior of a program.

Uses of type systems

- Naming and organizing useful concepts
- Providing information (to the compiler or programmer) about data manipulated by a program
- Ensuring that the run-time behavior of programs meet certain criteria.

Simply-typed lambda calculus

We will 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*.

Simply-typed lambda calculus

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.

Simply-typed lambda calculus: Syntax

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**.

Simply-typed lambda calculus: Syntax

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

Simply-typed lambda calculus: CBV small step operational semantics

The operational semantics of the simply-typed lambda calculus are the same as the untyped lambda calculus.

Simply-typed lambda calculus: CBV small step operational semantics

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

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

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

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

The typing relation

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

The typing relation

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*.

The typing relation

A term e is stuck if e is not a value and there is no term e' such that $e \longrightarrow e'$.

The typing relation

$$42 + \lambda x.\, x$$

The typing relation

() 47

Typing judgment

- 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.

Typing judgment

- 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.
- We write $\vdash e : \tau$ to mean that the closed term e has type τ under the empty context.

Well-typed expression

- 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*.

Inductive definition of $\Gamma \vdash e : \tau$

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

$$\textsc{T-Unit}\ \frac{}{\Gamma \vdash () : \mathbf{unit}}$$

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

$$\textsc{T-App}\ \frac{\Gamma \vdash e_1 : \tau \to \tau' \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\ e_2 : \tau'}$$

Inductive definition of $\Gamma \vdash e : \tau$

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**.

Inductive definition of $\Gamma \vdash e : \tau$

- Variable x has whatever type the context associates with x.

- 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.

- 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 type τ.

Type-checking an expression

Consider the program $(\lambda x\!:\!\mathbf{int}.\, x + 40)\ 2$.

Type-checking an expression

The following is a proof that $(\lambda x : \mathbf{int}.\, x + 40)\ 2$ is well-typed.

Type-checking an expression

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

Theorem (Type soundness)

If $\vdash e : \tau$ and $e \longrightarrow^* e'$ then either e' is a value, or there exists e'' such that $e' \longrightarrow e''$.

Theorem (Type soundness)

To prove this, we use two lemmas: *preservation* and *progress*.

Theorem (Type soundness)

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.

Theorem (Type soundness)

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.

Lemma (Preservation)

If $\vdash e : \tau$ and $e \longrightarrow e'$ then $\vdash e' : \tau$.

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

To prove this, 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

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

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

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

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

$\textsc{Add}$
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 $\textsc{T-Int}$, we have $\vdash e' : \textbf{int}$ as required.

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

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.

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

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 $\vdash E[e_2] : \tau$ as required.

If $\vdash e : \tau$ and $e \longrightarrow e'$ then $\vdash e' : \tau$

This proves the lemma.

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* $\vdash E[e_1] : \tau$.

Lemma (Progress)

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

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

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

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

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

$\textsc{T-Var}$ This case is impossible, since a variable is not well-typed in the empty environment.

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

T-Unit, T-Int, T-Abs
Trivial, since e must be a value.

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

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.

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

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 $e_1\ e_2 \longrightarrow e'\{e_2/x\}$.

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

This proves the Progress lemma.

Expressive power of the simply-typed lambda calculus

Are there programs that do not get stuck that are not well-typed?

Expressive power of the simply-typed lambda calculus

Unfortunately, the answer is yes.
Consider the identity function $\lambda x.\, x$.
We must provide a type for the argument. If we specify $\lambda x : \mathbf{int}.\, x$, then the program $(\lambda x : \mathbf{int}.\, x)\ ()$ is not well-typed, even though it does not get stuck.

Expressive power of the simply-typed lambda calculus: Recursion

We can no longer write recursive functions. Consider $\Omega = (\lambda x.\, x\ x)\ (\lambda x.\, x\ x)$. Let's suppose that the type of $\lambda x.\, x\ x$ is $\tau \to \tau'$. Then τ must be equal to $\tau \to \tau'$. There is no such type for which this equality holds.

Theorem (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.