

CS 4110 – Programming Languages and Logics

Lecture #20: 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 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. 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	$\tau ::= \mathbf{int} \mid \mathbf{unit} \mid \tau_1 \rightarrow \tau_2$

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.

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

$$\text{CONTEXT} \frac{e \rightarrow e'}{E[e] \rightarrow E[e']}$$

$$\beta\text{-REDUCTION} \frac{}{(\lambda x:\tau. e) v \rightarrow e\{v/x\}}$$

$$\text{ADD} \frac{}{n_1 + n_2 \rightarrow n} n = n_1 + n_2$$

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 $e \rightarrow e'$. 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 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.

$$\begin{array}{c} \text{T-INT} \frac{}{\Gamma \vdash n : \mathbf{int}} \quad \text{T-ADD} \frac{\Gamma \vdash e_1 : \mathbf{int} \quad \Gamma \vdash e_2 : \mathbf{int}}{\Gamma \vdash e_1 + e_2 : \mathbf{int}} \quad \text{T-UNIT} \frac{}{\Gamma \vdash () : \mathbf{unit}} \\ \\ \text{T-VAR} \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \quad \text{T-ABS} \frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x : \tau. e : \tau \rightarrow \tau'} \quad \text{T-APP} \frac{\Gamma \vdash e_1 : \tau \rightarrow \tau' \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 e_2 : \tau'} \end{array}$$

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, $x \in \text{dom}(\Gamma)$. The abstraction $\lambda x : \tau. e$ has the function type $\tau \rightarrow \tau'$ 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 $\tau \rightarrow \tau'$, 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.

$$\text{T-APP} \frac{\text{T-ABS} \frac{\text{T-ADD} \frac{\text{T-VAR} \frac{}{x : \mathbf{int}} \vdash x : \mathbf{int}} \quad \text{T-INT} \frac{}{x : \mathbf{int}} \vdash 40 : \mathbf{int}}{\vdash x + 40 : \mathbf{int}}}{\vdash \lambda x : \mathbf{int}. x + 40 : \mathbf{int} \rightarrow \mathbf{int}} \quad \text{T-INT} \frac{}{\vdash 2 : \mathbf{int}}}{\vdash (\lambda x : \mathbf{int}. x + 40) 2 : \mathbf{int}}$$

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 $e \rightarrow^* e'$ and $e' \not\rightarrow$ then e' is a value and $\vdash e' : \tau$.*

We'll prove this theorem in the next lecture.