9. Simply Typed Lambda-Calculus (Types and Programming Languages)

Kwanghoon Choi

Software Languages and Systems Laboratory
Chonnam National University

Week 6

Overview

This chapter introduces the most elementary member of the family of typed languages: the simply typed lambda-calculus of Church (1940) and Curry (1958).

9.1 Function Types

Recall that in the type system for the arithmetic PL with two types:

- Bool: classifying terms whose evaluation yields a boolean
- Nat: classifying terms whose evaluation yields a number

The "ill-typed" terms not belonging to either of these types include all the terms that reach stuck states during evaluation

- e.g., if 0 then 1 else 2

as well as some terms that actually behave fine during evaluation but for which our static classification is too conservative

- e.g., if true then 0 else false.

To extend the type system to include functions, we need to add a type classifying terms whose evaluation results in a function.

9.1 Function Types (Cont.)

To extend the type system to include functions, we need to add a type classifying terms whose evaluation results in a function.

The first trial: to add a typing rule "λx.t : $\rightarrow$" giving every lambda-abstraction the type $\rightarrow$.

- Good to classify lambda-abstractions from booleans and numbers, but
- Too conservative: functions like λx.true and λx.λy.y are in the same type $\rightarrow$,

We need to know what type the function returns and what type of arguments it expects: T1 $\rightarrow$ T2

9.1 Function Types (Cont.)

The syntax of types in the lambda calculus

$$T \quad ::= \quad Nat \quad | \quad Bool \quad | \quad T \rightarrow T$$

- Bool $\rightarrow$ Bool is the type of functions mapping boolean arguments to boolean results.
- "T1 $\rightarrow$ T2 $\rightarrow$ T3" denotes T1 $\rightarrow$ (T2 $\rightarrow$ T3), not (T1 $\rightarrow$ T2) $\rightarrow$ T3.
- (Bool $\rightarrow$ Bool) $\rightarrow$ (Bool $\rightarrow$ Bool)–or, equivalently, (Bool $\rightarrow$ Bool) $\rightarrow$ Bool $\rightarrow$ Bool
  : the type of functions that take boolean-to-boolean functions as arguments and return them as results.

9.2 The Typing Relation

In order to assign a type to an abstraction λx.t, we need to know what type of arugments to expect.

The explicitly typed approach with λx:T. t annotates the type to the argument of the abstraction.

The other approach infers the type (T) of the argument (x) based on how the argument is used in the body of the abstraction (t).

The type of the function's result is the type of the body t2, where occurrences of x in t2 are assumed to denote terms of type T1:

$$\frac{\texttt{x:T1} \vdash \texttt{t2 : T2}}{\vdash \lambda\texttt{x:T1.t2 : T1}\rightarrow\texttt{T2}}$$

"$\Gamma \vdash$ t : T" instead of "t : T" where $\Gamma = \{$ x1:T1, ... , xk:Tk $\}$.

9.2 The Typing Relation (Cont.)

Pure simply typed lambda-calculus ($\lambda_{\rightarrow}$)

$$\begin{array}{llllllll}
\text{Types} & \mathsf{T} & ::= & \mathsf{Nat} & | & \mathsf{Bool} & | & \mathsf{T} \rightarrow \mathsf{T} \\
\text{Terms} & \mathsf{t} & ::= & \mathsf{x} & | & \lambda\, \mathsf{x}{:}\mathsf{T}.\mathsf{t} & | & \mathsf{t}\ \ \mathsf{t} \\
\text{Values} & \mathsf{v} & ::= & \lambda\, \mathsf{x}{:}\mathsf{T}.\mathsf{t} & & & &
\end{array}$$

Typing rules

$$\frac{\Gamma(\mathtt{x})=\mathtt{T}}{\Gamma \vdash \mathtt{x} : \mathtt{T}} \qquad \text{(T-Var)}$$

$$\frac{\Gamma,\ \mathtt{x}:\mathtt{T1} \vdash \mathtt{t} : \mathtt{T2}}{\Gamma \vdash \lambda\mathtt{x}.\mathtt{t} : \mathtt{T1} \rightarrow \mathtt{T2}} \qquad \text{(T-Abs)}$$

$$\frac{\Gamma \vdash \mathtt{t1} : \mathtt{T1} \rightarrow \mathtt{T2} \qquad \Gamma \vdash \mathtt{t2} : \mathtt{T1}}{\Gamma \vdash \mathtt{t1}\ \mathtt{t2} : \mathtt{T2}} \qquad \text{(T-App)}$$

9.2 The Typing Relation (Cont.)

Typing rules for the boolean constants and conditional expressions are the same as before.

$$\frac{\Gamma \vdash \texttt{t1 : Bool} \quad \Gamma \vdash \texttt{t2 : T} \quad \Gamma \vdash \texttt{t3 : T}}{\Gamma \vdash \texttt{if t1 then t2 else t3 : T}} \quad \text{(T-If)}$$

cf.

$$\frac{\texttt{t1 : Bool} \quad \texttt{t2 : T} \quad \texttt{t3 : T}}{\texttt{if t1 then t2 else t3 : T}}$$

Q. Rewrite all the typing rules for the arithmetic PL in 3-ary relation.

9.2 The Typing Relation (Cont.)

The typing derivation for “(λx:Bool. x) true”

$$\dfrac{\dfrac{\dfrac{\texttt{x:Bool} \in \texttt{x:Bool}}{\texttt{x:Bool} \vdash \texttt{x : Bool}}\ \textsc{T-Var}}{\vdash \lambda\texttt{x:Bool.x : Bool}\to\texttt{Bool}}\ \textsc{T-Abs} \qquad \dfrac{}{\vdash \texttt{true : Bool}}\ \textsc{T-True}}{\vdash (\lambda\texttt{x:Bool.x) true : Bool}}\ \textsc{T-App}$$

Note that we write $\vdash$ t : T (or $\emptyset \vdash$ t : T) when Γ is empty.

Q. Show (by drawing derivation trees) that the following terms have the indicated types:

- f:Bool$\to$Bool $\vdash$ f (if false then true else false) : Bool
- f:Bool$\to$Bool $\vdash$ λx:Bool. f (if x then false else x) : Bool$\to$Bool

9.2 The Typing Relation (Cont.)

Q Find a context Γ under which the term "f x y" has type Bool.

Q. Can you give a simple description of the set of *all* such contexts?

9.3 Properties of Typing

The property of the type system for the lambda calculus is safety (also called soundness)

- If $\vdash$ `t : T`, then the evaluation of the term, `t`, will never get stuck.
- cf. The stuck terms in the lambda calculus are not values but have no progress by the evaluation rules.

We show the type safety by proving two properites: (1) progress and (2) preservation

9.3 Properties of Typing (Cont.)

Progress: A well-typed term is not stuck (either it is a value or it can take a step according to the evaluation rules).

Theorem[Progress] Suppose t is a closed term. If $\vdash$ `t : T` then

- either `t` is a value
- or there is some `t'` with `t` $\rightarrow$ `t'`.

Q. Prove this theorem by induction on typing derivations $\vdash$ `t : T`.

9.3 Properties of Typing (Cont.)

Preservation: If a well-typed term takes a step of evaluation, then the resulting term is also well-typed.

Theorem[Preservation] If $\Gamma \vdash \texttt{t} : \texttt{T}$ and $\texttt{t} \to \texttt{t'}$ then $\Gamma \vdash \texttt{t'} : \texttt{T}$.

Q. Prove this theorem by induction on a derivation of $\Gamma \vdash \texttt{t} : \texttt{T}$ using the following substitution lemma.

Lemma[Preservation of types under substitution] If $\Gamma, \texttt{x}:\texttt{S} \vdash \texttt{t}:\texttt{T}$ and $\Gamma \vdash \texttt{s}:\texttt{S}$, then $\Gamma \vdash [\texttt{x} \mapsto \texttt{s}]\texttt{t}:\texttt{T}$.

- Provable by induction on a derivation of the statement $\Gamma, \texttt{x}:\texttt{S} \vdash \texttt{t}:\texttt{T}$.

9.4 The Curry-Howard Correspondence

A connection between type theory and logic known as the *Curry-Howard correspondence* or *Curry-Howard isomorphism* (Curry and Feys, 1958; Howard, 1980).

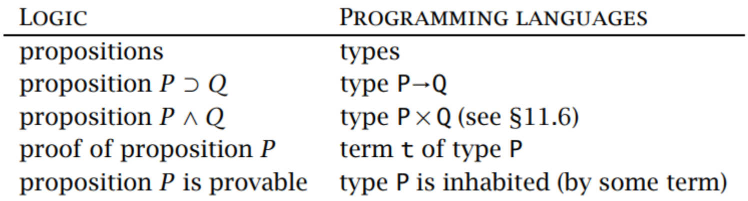
The idea is that, in constructive logics, a proof of a porposition p consists of concrete *evidence* for P.

For example,

- A proof of a proposition $P \supset Q$ can be viewed as a mechanical procedure that, given a proof of P, constructs a proof of Q.
- A proof of $P \wedge Q$ consists of a proof of P together with a proof of Q.

9.4 The Curry-Howard Correspondence (Cont.)

This observation gives rise to the following correspondence:



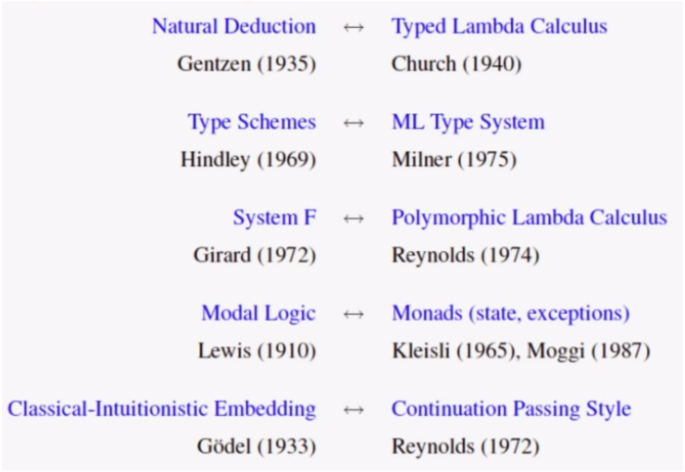
LOGIC	PROGRAMMING LANGUAGES
propositions	types
proposition $P \supset Q$	type $\mathsf{P} \rightarrow \mathsf{Q}$
proposition $P \wedge Q$	type $\mathsf{P} \times \mathsf{Q}$ (see §11.6)
proof of proposition P	term `t` of type P
proposition P is provable	type P is inhabited (by some term)

On this view, a term of the simply typed lambda calculus is a proof of a logical proposition corresponding to its type.

Computation-reduction of lambda-terms- corresponds to the logical operation of proof simplification by *cut elimination*.

9.4 The Curry-Howard Correspondence (Cont.)

The beauty of the Curry-Howard correspondence is that it can be extended to a huge variety of type systems and logics.



Natural Deduction	↔	Typed Lambda Calculus
Gentzen (1935)		Church (1940)
Type Schemes	↔	ML Type System
Hindley (1969)		Milner (1975)
System F	↔	Polymorphic Lambda Calculus
Girard (1972)		Reynolds (1974)
Modal Logic	↔	Monads (state, exceptions)
Lewis (1910)		Kleisli (1965), Moggi (1987)
Classical-Intuitionistic Embedding	↔	Continuation Passing Style
Gödel (1933)		Reynolds (1972)

9.5 Erasure and Typability

We have defined the evaluation relation directly on simply typed terms. But programs can be converted back to an untyped form before they are evaluated.

9.6 Curry-style vs. Church-style

Curry-style : Semantics is prior to typing.

Church-style : Typing is prior to semantics.

Implicitly typed presentations of lambda-calculi are often given in the Curry style, while Church-style presentations are common only for explicitly typed systems.