G54FOP: Lecture 13

Types and Type Systems II

Henrik Nilsson

University of Nottingham, UK

This Lecture

Extensions:

- typing let-expressions
- typing functions

Much of this lecture follows parts of the first few chapters of B. C. Pierce 2002 *Types and Programming Languages* closely.

Recap: Example Language (1)

Abstract syntax:

t	$\longrightarrow$		*terms:*
		`true`	*constant true*
	$\mid$	`false`	*constant false*
	$\mid$	`if` t `then` t `else` t	*conditional*
	$\mid$	`0`	*constant zero*
	$\mid$	`succ` t	*successor*
	$\mid$	`pred` t	*predecessor*
	$\mid$	`iszero` t	*zero test*

Recap: Example Language (2)

Values:

v	$\longrightarrow$		*values:*
		true	*true value*
	|	**false**	*false value*
	|	nv	*numeric value*
nv	$\longrightarrow$		*numeric values:*
		0	*zero value*
	|	**succ** nv	*successor value*

Recap: Example Language (3)

Dynamic semantics:

$t \longrightarrow t'$ Read: t evaluates to t' in one step

$$\texttt{if true then } t_2 \texttt{ else } t_3 \longrightarrow t_2 \quad \text{(E-IFTRUE)}$$

$$\texttt{if false then } t_2 \texttt{ else } t_3 \longrightarrow t_3$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \longrightarrow \texttt{if } t_1' \texttt{ then } t_2 \texttt{ else } t_3} \quad \text{(E-IF)}$$

Recap: Example Language (4)

$$\frac{t_1 \longrightarrow t_1'}{\texttt{succ}\ t_1 \longrightarrow \texttt{succ}\ t_1'} \quad \text{(E-SUCC)}$$

$$\texttt{pred}\ \texttt{0} \longrightarrow \texttt{0} \quad \text{(E-PREDZERO)}$$

$$\texttt{pred}\ (\texttt{succ}\ nv_1) \longrightarrow nv_1 \quad \text{(E-PREDSUCC)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{pred}\ t_1 \longrightarrow \texttt{pred}\ t_1'} \quad \text{(E-PRED)}$$

Recap: Example Language (5)

$$\texttt{iszero}\ \texttt{0} \longrightarrow \texttt{true} \qquad \text{(E-IsZeroZero)}$$

$$\texttt{iszero}\ (\texttt{succ}\ nv_1) \longrightarrow \texttt{false} \qquad \text{(E-IsZeroSucc)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{iszero}\ t_1 \longrightarrow \texttt{iszero}\ t_1'} \qquad \text{(E-IsZero)}$$

Recap: Example Language (6)

Types and typing rules::

$$
\begin{array}{rcll}
T & \rightarrow & & \textit{types:} \\
 & & \texttt{Bool} & \textit{type of booleans} \\
 & | & \texttt{Nat} & \textit{type of natural numbers}
\end{array}
$$

Recap: Example Language (7)

$$\texttt{true} : \texttt{Bool} \qquad \text{(T-TRUE)}$$

$$\texttt{false} : \texttt{Bool} \qquad \text{(T-FALSE)}$$

$$\frac{t_1 : \texttt{Bool} \quad t_2 : T \quad t_3 : T}{\texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3 : T}$$

$$\texttt{0} : \texttt{Nat} \qquad \text{(T-ZERO)}$$

$$\frac{t_1 : \texttt{Nat}}{\texttt{succ}\ t_1 : \texttt{Nat}} \qquad \text{(T-SUCC)}$$

$$\frac{t_1 : \texttt{Nat}}{\texttt{pred}\ t_1 : \texttt{Nat}} \qquad \text{(T-PRED)}$$

$$\frac{t_1 : \texttt{Nat}}{\texttt{iszero}\ t_1 : \texttt{Bool}} \qquad \text{(T-ISZERO)}$$

Extension: Let Bindings (1)

Syntactic extension:

$$
\begin{array}{rcll}
t & \rightarrow & \ldots & \textit{terms:} \\
  & | & x & \textit{variable} \\
  & | & \texttt{let}\ x\ \texttt{=}\ t\ \texttt{in}\ t & \textit{let binding}
\end{array}
$$

New evaluation rules:

$$\texttt{let}\ x\ \texttt{=}\ v_1\ \texttt{in}\ t_2 \longrightarrow [x \mapsto v_1]\ t_2 \qquad \text{(E-LETV)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{let}\ x\ \texttt{=}\ t_1\ \texttt{in}\ t_2 \longrightarrow \texttt{let}\ x\ \texttt{=}\ t_1'\ \texttt{in}\ t_2} \qquad \text{(E-LET)}$$

Extension: Let Bindings (2)

We now need a ***typing context*** or ***type environment*** to keep track of types of variables.

The typing relation thus become a ***ternary relation***:

$$\Gamma \vdash t : T$$

Read: term t has type T in type environment Γ.

Extension: Let Bindings (3)

Context-related notation:

- Empty context:

$$\emptyset$$

- Extending a context:

$$\Gamma, x : T$$

  New declaration understood to replace any earlier declaration for variable with same name.

- Stating that type of a variable is given by context:

$$x : T \in \Gamma \quad \text{or} \quad \Gamma(x) = T$$

Extension: Let Bindings (4)

Updated typing rules:

$$\Gamma \vdash \texttt{true} : \texttt{Bool} \qquad \text{(T-TRUE)}$$

$$\Gamma \vdash \texttt{false} : \texttt{Bool} \qquad \text{(T-FALSE)}$$

$$\frac{\Gamma \vdash t_1 : \texttt{Bool} \quad \Gamma \vdash t_2 : T \quad \Gamma \vdash t_3 : T}{\Gamma \vdash \texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3 : T} \qquad \text{(T-IF)}$$

Extension: Let Bindings (5)

Updated typing rules:

$$\Gamma \vdash \mathtt{0} : \mathtt{Nat} \qquad \text{(T-ZERO)}$$

$$\frac{\Gamma \vdash t_1 : \mathtt{Nat}}{\Gamma \vdash \mathtt{succ}\ t_1 : \mathtt{Nat}} \qquad \text{(T-SUCC)}$$

$$\frac{\Gamma \vdash t_1 : \mathtt{Nat}}{\Gamma \vdash \mathtt{pred}\ t_1 : \mathtt{Nat}} \qquad \text{(T-PRED)}$$

$$\frac{\Gamma \vdash t_1 : \mathtt{Nat}}{\Gamma \vdash \mathtt{iszero}\ t_1 : \mathtt{Bool}} \qquad \text{(T-ISZERO)}$$

Extension: Let Bindings (6)

New typing rules:

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad \text{(T-VAR)}$$

Extension: Let Bindings (6)

New typing rules:

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad \text{(T-VAR)}$$

$$\frac{\Gamma \vdash t_1 : T_1 \quad \Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \texttt{let}\ x = t_1\ \texttt{in}\ t_2 : T_2} \qquad \text{(T-LET)}$$

Exercise

Derive:

$$\emptyset \vdash \texttt{let x = (let y = 0 in y) in succ x} : \texttt{Nat}$$

Extension: Functions (1)

Syntactic extension:

t	$\longrightarrow$	$\ldots$	*terms:*
	$\mid$	$\lambda x{:}T.t$	*abstraction*
	$\mid$	$t\ t$	*application*
v	$\longrightarrow$	$\ldots$	*values:*
	$\mid$	$\lambda x{:}T.t$	*abstraction value*
T	$\longrightarrow$	$\ldots$	*types:*
	$\mid$	$T \rightarrow T$	*type of functions*

Extension: Functions (2)

New evaluation rules:

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \qquad \text{(E-APP1)}$$

$$\frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'} \qquad \text{(E-APP2)}$$

$$(\lambda x\!:\!T_{11}.\,t_{12})\ v_2 \longrightarrow [x \mapsto v_2]\ t_{12} \qquad \text{(E-APPABS)}$$

Note:

- left to right evaluation order: first the function (E-APP1), then the argument (E-APP2)
- ***call-by-value***: the argument fully evaluated before function “invoked” (E-APPABS).

Extension: Functions (3)

New typing rules:

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x \mathbin{:} T_1 \,.\, t_2 : T_1 \rightarrow T_2} \qquad \text{(T-ABS)}$$

Extension: Functions (3)

New typing rules:

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1 . t_2 : T_1 \rightarrow T_2} \quad \text{(T-ABS)}$$

$$\frac{\Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}} \quad \text{(T-APP)}$$

Homework

Derive:

$$\Gamma_1 \vdash (\lambda \texttt{f:Nat}\rightarrow\texttt{Nat.f 0})\ \texttt{double} : \texttt{Nat}$$

given

$$\Gamma_1 = \emptyset, \texttt{double} : \texttt{Nat}\rightarrow\texttt{Nat}$$

The Simply Typed λ-Calculus (1)

The "function fragment" of our language is known as the (pure) ***simply typed λ-Calculus ($\lambda_{\rightarrow}$)***:

$$
\begin{array}{lcll}
T & \rightarrow & & \textit{types:} \\
& | & B & \textit{fixed set of base types} \\
& | & T \rightarrow T & \textit{type of functions} \\
t & \rightarrow & & \textit{terms:} \\
& | & x & \textit{variable} \\
& | & c & \textit{constant (optional)} \\
& | & \lambda x : T \,.\, t & \textit{abstraction} \\
& | & t\ t & \textit{application}
\end{array}
$$

The Simply Typed λ-Calculus (2)

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad \text{(T-VAR)}$$

$$\frac{c \text{ is a constant of type } T}{\Gamma \vdash c : T} \qquad \text{(T-CONST-c)}$$

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1 \,.\, t_2 : T_1 \to T_2} \qquad \text{(T-ABS)}$$

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}} \qquad \text{(T-APP)}$$

The Simply Typed λ-Calculus (3)

- At least ***one*** base type needed, or not possible to construct finite types.

The Simply Typed λ-Calculus (3)

- At least ***one*** base type needed, or not possible to construct finite types.
- Frequently, B is taken to consist of only one type, o, "the type of propositions", without ***any*** term constants.

The Simply Typed λ-Calculus (3)

- At least ***one*** base type needed, or not possible to construct finite types.
- Frequently, B is taken to consist of only one type, o, "the type of propositions", without ***any*** term constants.
- The simply typed lambda calculus is ***strongly normalizing***: well-typed terms ***always*** reduce to a value (regardless of reduction order).

The Simply Typed λ-Calculus (3)

- At least ***one*** base type needed, or not possible to construct finite types.
- Frequently, B is taken to consist of only one type, o, “the type of propositions”, without ***any*** term constants.
- The simply typed lambda calculus is ***strongly normalizing***: well-typed terms ***always*** reduce to a value (regardless of reduction order).
- Why? Because ***self application*** as used in the definition of e.g. Y or $\omega \equiv \lambda x . x\ x$ cannot be typed. Thus no way to express recursion.

The Simply Typed λ-Calculus (3)

- To see this, note that to type $x\ x$, we need both

$$
\begin{aligned}
x &: T_1 \rightarrow T_2 \\
x &: T_1
\end{aligned}
$$

for some types T_1 and T_2; i.e., $T_1 = T_1 \rightarrow T_2$. But there is no (finite) solution to this equation.

The Simply Typed λ-Calculus (3)

- To see this, note that to type $x\ x$, we need both

$$\begin{aligned} x &: T_1 \rightarrow T_2 \\ x &: T_1 \end{aligned}$$

  for some types T_1 and T_2; i.e., $T_1 = T_1 \rightarrow T_2$. But there is no (finite) solution to this equation.
- However, general recursion can be regained by adding a special fixed-point operator (parametrised on type α):

$$\mathrm{fix}_\alpha : (\alpha \rightarrow \alpha) \rightarrow \alpha$$

Aside: Strong and Weak Normalization

- ***Strong normalization:*** Reduction always terminates for all terms, regardless of reduction order.
- ***Weak normalization:*** There is at least one terminating reduction sequence for each term.

The Simply Typed λ-Calculus (4)

Q: Why are the constants “optional”?

The Simply Typed λ-Calculus (4)

Q: Why are the constants "optional"?

A: As we have seen, constants like **`true`**, **`false`**, **`0`**, and associated functions can be ***encoded*** in the base calculus. However:

The Simply Typed λ-Calculus (4)

Q: Why are the constants "optional"?

A: As we have seen, constants like **`true`**, **`false`**, **`0`**, and associated functions can be ***encoded*** in the base calculus. However:

- It is often ***convenient*** to add constants explicitly, even if λ-definable, along with δ***-reduction rules*** that describe their behaviour.

The Simply Typed λ-Calculus (4)

Q: Why are the constants "optional"?

A: As we have seen, constants like **`true`**, **`false`**, **`0`**, and associated functions can be ***encoded*** in the base calculus. However:

- It is often ***convenient*** to add constants explicitly, even if λ-definable, along with ***δ-reduction rules*** that describe their behaviour.
- It is sometimes ***necessary*** to add constants that cannot be encoded, e.g. fixed-point combinators.