Polymorphism of \texttt{let}-bound variables in ML

For example in

\[
\text{let } f = \lambda x(x) \text{ in } (f \text{ true}) :: (f \text{ nil})
\]

\(\lambda x(x)\) has type \(\tau \rightarrow \tau\) for any type \(\tau\), and the variable \(f\) to which it is bound is used polymorphically:

- in \((f \text{ true})\), \(f\) has type \(\text{bool} \rightarrow \text{bool}\)

- in \((f \text{ nil})\), \(f\) has type \(\text{bool list} \rightarrow \text{bool list}\)

Overall, the expression has type \(\text{bool list}\).
Mini-ML expressions, $M$

$$\ ::= \ x$$

| $true$ | variable
| $false$ | boolean values
| $if \ M \ then \ M \ else \ M$ | conditional
| $\lambda x(M)$ | function abstraction
| $MM$ | function application
| $let \ x = M \ in \ M$ | local declaration
| $nil$ | nil list
| $M :: M$ | list cons
| $case \ M \ of \ nil \ => \ M \ | \ x :: x \ => \ M$ | case expression
Mini-ML types and type schemes

**Types**

\[ \tau ::= \alpha \quad \text{type variable} \]
\[ | \quad \text{bool} \quad \text{type of booleans} \]
\[ | \quad \tau \rightarrow \tau \quad \text{function type} \]
\[ | \quad \tau \text{list} \quad \text{list type} \]

where \( \alpha \) ranges over a fixed, countably infinite set \( \text{TyVar} \).

**Type Schemes**

\[ \sigma ::= \forall A (\tau) \]

where \( A \) ranges over finite subsets of the set \( \text{TyVar} \).

When \( A = \{ \alpha_1, \ldots, \alpha_n \} \), we write \( \forall A (\tau) \) as

\[ \forall \alpha_1, \ldots, \alpha_n (\tau). \]

E.g.s of type schemes: \( \forall \alpha, \beta (\alpha \rightarrow \beta) \) \( \forall \alpha (\alpha \text{list} \rightarrow \beta) \)
Mini-ML types and type schemes

**Types**

\[ \tau ::= \alpha \text{ type variable} \]
\[ | \ \text{bool} \text{ type of booleans} \]
\[ | \ \tau \rightarrow \tau \text{ function type} \]
\[ | \ \tau \text{ list} \text{ list type} \]

where \( \alpha \) ranges over a fixed, countably infinite set \( \text{TyVar} \).

**Type Schemes**

\[ \sigma ::= \forall A (\tau) \]

where \( A \) ranges over finite subsets of the set \( \text{TyVar} \).

When \( A = \{ \alpha_1, \ldots, \alpha_n \} \), we write \( \forall A (\tau) \) as

\[ \forall \alpha_1, \ldots, \alpha_n (\tau). \]

E.g. s of type schemes: \( \forall \alpha, \beta (\alpha \rightarrow \beta) \)  \( \forall \alpha (\alpha \text{ list} \rightarrow \beta) \)  \( \forall \exists \gamma (\alpha \rightarrow \text{bool}) \)
Mini-ML typing judgement

takes the form $\Gamma \vdash M : \tau$ where

- the **typing environment** $\Gamma$ is a finite function from variables to type schemes.
  
  (We write $\Gamma = \{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$ to indicate that $\Gamma$ has domain of definition $\text{dom}(\Gamma) = \{x_1, \ldots, x_n\}$ and maps each $x_i$ to the type scheme $\sigma_i$ for $i = 1..n$.)

- $M$ is an Mini-ML expression

- $\tau$ is an Mini-ML type.
Mini-ML type system, I

(var $\succ$) \[ \Gamma \vdash x : \tau \quad \text{if} \ (x : \sigma) \in \Gamma \quad \text{and} \quad \sigma \succ \tau \]

(bool) \[ \Gamma \vdash B : bool \quad \text{if} \ B \in \{\text{true}, \text{false}\} \]

(if) \[ \Gamma \vdash M_1 : bool \quad \Gamma \vdash M_2 : \tau \quad \Gamma \vdash M_3 : \tau \]

\[ \Gamma \vdash \text{if } M_1 \text{ then } M_2 \text{ else } M_3 : \tau \]
The “generalises” relation between type schemes and types

We say a type scheme \( \sigma = \forall \alpha_1, \ldots, \alpha_n (\tau') \) generalises a type \( \tau \), and write \( \sigma \succ \tau \) if \( \tau \) can be obtained from the type \( \tau' \) by simultaneously substituting some types \( \tau_i \) for the type variables \( \alpha_i \) \((i = 1, \ldots, n)\):

\[
\tau = \tau'[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n].
\]

(N.B. The relation is unaffected by the particular choice of names of bound type variables in \( \sigma \).)

The converse relation is called specialisation: a type \( \tau \) is a specialisation of a type scheme \( \sigma \) if \( \sigma \succ \tau \).

E.g. \( \forall \alpha, \beta (\alpha \rightarrow \beta) \succ \text{bool} \rightarrow \text{bool} \)

but \( \forall \alpha (\alpha \rightarrow \beta) \not\succ \text{bool} \rightarrow \text{bool} \)
The “generalises” relation between type schemes and types

We say a type scheme \( \sigma = \forall \alpha_1, \ldots, \alpha_n (\tau') \) generalises a type \( \tau \), and write \( \sigma \succ \tau \) if \( \tau \) can be obtained from the type \( \tau' \) by simultaneously substituting some types \( \tau_i \) for the type variables \( \alpha_i \) (\( i = 1, \ldots, n \)):

\[
\tau = \tau'[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n].
\]

(N.B. The relation is unaffected by the particular choice of names of bound type variables in \( \sigma \).)

The converse relation is called specialisation: a type \( \tau \) is a specialisation of a type scheme \( \sigma \) if \( \sigma \succ \tau \).

So we identify type schemes up to renaming bound type vars.

\[\forall \alpha(\alpha \to \alpha') = \forall \alpha''(\alpha'' \to \alpha') \neq \forall \alpha'(\alpha' \to \alpha')\]
(nil) \( \Gamma \vdash \text{nil} : \tau \ list \)

(cons) \[ \frac{\Gamma \vdash M_1 : \tau \quad \Gamma \vdash M_2 : \tau \ list}{\Gamma \vdash M_1 :: M_2 : \tau \ list} \]

(case) \[ \frac{\Gamma \vdash M_1 : \tau_1 \ list \quad \Gamma \vdash M_2 : \tau_2}{\Gamma \vdash \text{case } M_1 \text{ of nil } \Rightarrow M_2 \quad \mid x_1 :: x_2 \Rightarrow M_3 : \tau_2}{\Gamma \vdash \text{case } M_1 \text{ of nil } \Rightarrow M_2 \quad \mid x_1 :: x_2 \Rightarrow M_3 : \tau_2}{\text{if } x_1, x_2 \notin \text{dom}(\Gamma) \quad \text{and } x_1 \neq x_2} \]
(nil) \[ \Gamma \vdash \text{nil} : \tau \text{list} \]

(cons) \[ \frac{\Gamma \vdash M_1 : \tau \quad \Gamma \vdash M_2 : \tau \text{list}}{\Gamma \vdash M_1 :: M_2 : \tau \text{list}} \]

(case) \[ \frac{\Gamma \vdash M_1 : \tau_1 \text{list} \quad \Gamma \vdash M_2 : \tau_2 \quad \Gamma, x_1 : \tau_1, x_2 : \tau_1 \text{list} \vdash M_3 : \tau_2}{\Gamma \vdash \text{case } M_1 \text{ of nil } \Rightarrow M_2}{\mid x_1 :: x_2 \Rightarrow M_3 : \tau_2} \]

if \( x_1, x_2 \notin \text{dom}(\Gamma) \) and \( x_1 \neq x_2 \)

abbreviation for \( \Gamma, x : A \vdash \exists y. \text{let } z = y \text{ in } z \)
Mini-ML type system, III

(fn)

\[ \frac{\Gamma, x : \tau_1 \vdash M : \tau_2}{\Gamma \vdash \lambda x(M) : \tau_1 \rightarrow \tau_2} \]

if \( x \notin \text{dom}(\Gamma) \)

(app)

\[ \frac{\Gamma \vdash M_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash M_2 : \tau_1}{\Gamma \vdash M_1 M_2 : \tau_2} \]
(let)

\[\Gamma \vdash M_1 : \tau\]
\[\Gamma, x : \forall A \left(\tau\right) \vdash M_2 : \tau'\]
\[\Gamma \vdash \text{let } x = M_1 \text{ in } M_2 : \tau'\]

if \(x \not\in \text{dom}(\Gamma)\) and \(A = \text{ftv}(\tau) - \text{ftv}(\Gamma)\)

\(\text{ftv}(\tau) = \text{all type vars occurring in type } \tau\)

\(\text{ftv}\{x_1 : \sigma_1, \ldots, x_n : \sigma_n\} = \text{ftv}(\sigma_1) \cup \ldots \cup \text{ftv}(\sigma_n)\)

where if \(\sigma = \forall A (\tau)\), then \(\text{ftv}(\sigma) = \text{ftv}(\tau) - A\)
Example of the (let) rule

\[ \Gamma \vdash M_1 : \tau \text{ is } \{y : \beta, z : \forall x (x \to x \to \text{bool})\} \vdash \lambda u(y) : \alpha \to \beta \]

so A is \{\alpha, \beta\} - \{\beta\} = \{\alpha\}
Example of the (let) rule

\[ \Gamma \vdash M_1 : \tau \text{ is } \{y : \beta, z : \forall y (x \rightarrow x \rightarrow \text{bool})\} \vdash \lambda u(y) : \alpha \rightarrow \beta \]

so \( A \) is \( \{\alpha, \beta\} - \{\beta\} = \{\alpha \}

\[ \Gamma, x : \forall A(x) \vdash M_2 : \tau' \text{ is } \{y : \beta, z : \forall y (x \rightarrow x \rightarrow \text{bool}), x : \forall \alpha (\alpha \rightarrow \beta)\} \vdash z(x y)(x \text{ nit}) : \text{bool} \]
Example of the (let) rule

$$\Gamma \vdash M_1 : \tau \text{ is } \{y : \beta, z : \forall y (y \rightarrow \tau \rightarrow \text{bool})\} \vdash \lambda u(y) : \alpha \rightarrow \beta$$

so $$\Delta$$ is $$\{\alpha, \beta\} - \{\beta\} = \{\alpha\}$$

$$\Gamma, x : \forall A(c) \vdash M_2 : \tau' \text{ is } \{y : \beta, z : \forall y (y \rightarrow \tau \rightarrow \text{bool}), x : \forall\alpha (\alpha \rightarrow \beta)\} \vdash z(xy)(x\text{nil}) : \text{bool}$$

Applying (let) we get

$$\{y : \beta, z : \forall y (y \rightarrow \tau \rightarrow \text{bool})\} \vdash \text{let } x = \lambda u(y) \text{ in } z(xy)(x\text{nil}) : \text{bool}$$
Assigning type schemes to Mini-ML expressions

Given a type scheme $\sigma = \forall A \ (\tau)$, write

$$\Gamma \vdash M : \sigma$$

if $A = \text{ftv}(\tau) - \text{ftv}(\Gamma)$ and $\Gamma \vdash M : \tau$ is derivable from the axiom and rules on Slides 16–19.

When $\Gamma = \{\}$ we just write $\vdash M : \sigma$ for $\{\} \vdash M : \sigma$ and say that the (necessarily closed—see Exercise 2.5.2) expression $M$ is typeable in Mini-ML with type scheme $\sigma$.

["closed" = "has no free variables"]
[cf. Slide 7]

(a) A Mini-ML type checking problem:

\[
\text{given closed } M \text{ and } \sigma, \text{ does } \vdash M : \sigma \text{ hold?}
\]

(b) A Mini-ML typeability problem:

\[
\text{given closed } M, \text{ does there exist a closed } \sigma \text{ such that } \vdash M : \sigma \text{ holds?}
\]

N.B. Solving (a) entails solving (b) because of the form of the (let) typing rule.
Two examples involving self-application

\[
M \overset{\text{def}}{=} \text{let } f = \lambda x_1(\lambda x_2(x_1)) \text{ in } f \ f \\
M' \overset{\text{def}}{=} (\lambda f(f \ f)) \ \lambda x_1(\lambda x_2(x_1))
\]

Are \( M \) and \( M' \) typeable in the Mini-ML type system?