Polymorphic Reference Types

[§3, p25]
Formal type systems

- Constitute the precise, mathematical characterisation of informal type systems (such as occur in the manuals of most typed languages.)

- Basis for type soundness theorems: “any well-typed program cannot produce run-time errors (of some specified kind).”

- Can decouple specification of typing aspects of a language from algorithmic concerns: the formal type system can define typing independently of particular implementations of type-checking algorithms.
ML types and expressions for mutable references

\[
\tau ::= \ldots \\
| \text{unit} \quad \text{unit type} \\
| \tau \text{ref} \quad \text{reference type}
\]

\[
M ::= \ldots \\
| () \quad \text{unit value} \\
| \text{ref } M \quad \text{reference creation} \\
| !M \quad \text{dereference} \\
| M := M \quad \text{assignment}
\]
Midi-ML’s extra typing rules

\[
\begin{array}{c}
\text{(unit)}
\hline
\Gamma \vdash () : \text{unit}
\end{array}
\]
Midi-ML’s extra typing rules

\[ \Gamma \vdash () : unit \]

\[ \Gamma \vdash M : \tau \]
\[ \Gamma \vdash \text{ref } M : \tau \text{ ref} \]
Midi-ML’s extra typing rules

[unit] \[
\Gamma \vdash () : \text{unit}
\]

[ref] \[
\Gamma \vdash M : \tau \\
\Gamma \vdash \text{ref } M : \tau \text{ ref}
\]

[get] \[
\Gamma \vdash M : \tau \text{ ref} \\
\Gamma \vdash !M : \tau
\]
Midi-ML’s extra typing rules

\[
\text{(unit)} \quad \frac{}{\Gamma \vdash () : \text{unit}}
\]

\[
\text{(ref)} \quad \frac{\Gamma \vdash M : \tau}{\Gamma \vdash \text{ref } M : \tau \text{ ref}}
\]

\[
\text{(get)} \quad \frac{\Gamma \vdash M : \tau \text{ ref}}{\Gamma \vdash !M : \tau}
\]

\[
\text{(set)} \quad \frac{\Gamma \vdash M_1 : \tau \text{ ref} \quad \Gamma \vdash M_2 : \tau}{\Gamma \vdash M_1 := M_2 : \text{unit}}
\]
Example

The expression

\[
\text{let } r = \text{ref } \lambda x (x) \text{ in }
\text{let } u = (r := \lambda x' (\text{ref }!x')) \text{ in }
(\!r)()
\]

has type \textit{unit}. 
The expression

\[
\begin{align*}
\text{let } r &= \text{ref } \lambda x \ (x) \text{ in} \\
\text{let } u &= (r := \lambda x' \ (\text{ref } !x')) \text{ in} \\
(!r)()
\end{align*}
\]

has type \textit{unit}.
Example

The expression

\[
\begin{align*}
\text{let } r &= \text{ref } \lambda x (x) \text{ in} \\
\text{let } u &= (r := \lambda x' (\text{ref } !x')) \text{ in} \\
(\text{!}r)()
\end{align*}
\]

has type \textit{unit}. 
The expression

\[
\text{let } r = \text{ref } \lambda x \,(x) \text{ in } \\
\text{let } u = (r := \lambda x' \,(\text{ref }!x')) \text{ in } \\
(!r)()
\]

has type \textit{unit}.
Example

The expression

\[
\begin{align*}
\text{let } r &= \text{ref } \lambda x (x) \text{ in} \\
\text{let } u &= (r := \lambda x' (\text{ref } x')) \text{ in} \\
(\text{ref })()
\end{align*}
\]

has type \textit{unit}. 

\[
\forall \alpha \ ((\alpha \to \alpha) \text{ ref})
\]
Example

\[ \sigma \triangleq \forall \alpha ((\alpha \to \alpha) \text{ref}) \]

The expression

\[
\begin{align*}
\text{let } r &= \text{ref}\ \lambda x\ (x) \text{ in} \\
\text{let } u &= (r := \lambda x' (\text{ref}!x')) \text{ in} \\
(\text{!}r)(())
\end{align*}
\]

has type \textit{unit}.

\[ \sigma \Rightarrow (\beta \text{ref} \to \beta \text{ref}) \text{ref} \]

\[ \sigma \Rightarrow (\text{unit} \to \text{unit}) \text{ref} \]
Formal type systems

- Constitute the precise, mathematical characterisation of informal type systems (such as occur in the manuals of most typed languages.)

- Basis for type soundness theorems: “any well-typed program cannot produce run-time errors (of some specified kind).”

- Can decouple specification of typing aspects of a language from algorithmic concerns: the formal type system can define typing independently of particular implementations of type-checking algorithms.
Midi-ML transition system

Small-step transition relations

\[ \langle M, s \rangle \rightarrow \langle M', s' \rangle \]
\[ \langle M, s \rangle \rightarrow FAIL \]
Midi-ML transition system

Small-step transition relations

\[ \langle M, s \rangle \rightarrow \langle M', s' \rangle \]
\[ \langle M, s \rangle \rightarrow \text{FAIL} \]

where

- \( M, M' \) range over Midi-ML expressions
- \( s, s' \) range over \textit{states} = finite functions
  \( s = \{ x_1 \mapsto V_1, \ldots, x_n \mapsto V_n \} \) mapping variables \( x_i \) to \textit{values} \( V_i \):

\[ V ::= x | \lambda x (M) | () | \text{true} | \text{false} | \text{nil} | V :: V \]
Midi-ML transition system

Small-step transition relations

\[ \langle M, s \rangle \rightarrow \langle M', s' \rangle \]
\[ \langle M, s \rangle \rightarrow \text{FAIL} \]

where

- \( M, M' \) range over Midi-ML expressions
- \( s, s' \) range over states = finite functions
  \[ s = \{ x_1 \mapsto V_1, \ldots, x_n \mapsto V_n \} \]
  mapping variables \( x_i \) to values \( V_i \):

\[ V ::= x \mid \lambda x (M) \mid () \mid \text{true} \mid \text{false} \mid \text{nil} \mid V :: V \]

- configurations \( \langle M, s \rangle \) are required to satisfy that the free variables of expression \( M \) are in the domain of definition of the state \( s \)
- symbol \( \text{FAIL} \) represents a run-time error
Midi-ML transition system

Small-step transition relations

\[ \langle M, s \rangle \rightarrow \langle M', s' \rangle \]
\[ \langle M, s \rangle \rightarrow \text{FAIL} \]

where

- \( M, M' \) range over Midi-ML expressions
- \( s, s' \) range over states = finite functions
  \( s = \{x_1 \mapsto V_1, \ldots, x_n \mapsto V_n\} \) mapping variables \( x_i \) to values \( V_i \):

\[ V ::= x | \lambda x (M) | () | \text{true} | \text{false} | \text{nil} | V :: V \]

- configurations \( \langle M, s \rangle \) are required to satisfy that the free variables of expression \( M \) are in the domain of definition of the state \( s \)
- symbol \( \text{FAIL} \) represents a run-time error

are inductively defined by syntax-directed rules…
Midi-ML transitions involving references

\[ \langle !x, s \rangle \rightarrow \langle s(x), s \rangle \text{ if } x \in \text{dom}(s) \]

where \( V \) ranges over values:

\[ V ::= x | \lambda x (M) | () | \text{true} | \text{false} | \text{nil} | V :: V \]
Midi-ML transitions involving references

\[ \langle !x, s \rangle \rightarrow \langle s(x), s \rangle \text{ if } x \in \text{dom}(s) \]

\[ \langle !V, s \rangle \rightarrow \text{FAIL} \text{ if } V \text{ not a variable} \]

where \( V \) ranges over values:

\[ V ::= x | \lambda x (M) | () | \text{true} | \text{false} | \text{nil} | V :: V \]
Midi-ML transitions involving references

\[ \langle !x, s \rangle \rightarrow \langle s(x), s \rangle \quad \text{if } x \in \text{dom}(s) \]

\[ \langle !V, s \rangle \rightarrow \text{FAIL} \quad \text{if } V \text{ not a variable} \]

\[ \langle x := V', s \rangle \rightarrow \langle (), s[x \mapsto V'] \rangle \]

where \( V \) ranges over values:

\[ V ::= x \mid \lambda x (M) \mid () \mid \text{true} \mid \text{false} \mid \text{nil} \mid V :: V \]
Midi-ML transitions involving references

\[ \langle !x, s \rangle \rightarrow \langle s(x), s \rangle \text{ if } x \in \text{dom}(s) \]

\[ \langle !V, s \rangle \rightarrow \text{FAIL} \text{ if } V \text{ not a variable} \]

\[ \langle x := V', s \rangle \rightarrow \langle (), s|x \mapsto V'| \rangle \]

\[ \langle V := V', s \rangle \rightarrow \text{FAIL} \text{ if } V \text{ not a variable} \]

where \( V \) ranges over values:

\[ V ::= x | \lambda x (M) | () | \text{true} | \text{false} | \text{nil} | V :: V \]
Midi-ML transitions involving references

\[ \langle !x, s \rangle \rightarrow \langle s(x), s \rangle \text{ if } x \in \text{dom}(s) \]

\[ \langle !V, s \rangle \rightarrow \text{FAIL} \text{ if } V \text{ not a variable} \]

\[ \langle x := V', s \rangle \rightarrow \langle (), s[x \mapsto V'] \rangle \]

\[ \langle V := V', s \rangle \rightarrow \text{FAIL} \text{ if } V \text{ not a variable} \]

\[ \langle \text{ref } V, s \rangle \rightarrow \langle x, s[x \mapsto V] \rangle \text{ if } x \notin \text{dom}(s) \]

where \( V \) ranges over values:

\[ V ::= x | \lambda x (M) | () | \text{true} | \text{false} | \text{nil} | V :: V \]
\[
\begin{align*}
\langle M, s \rangle \rightarrow \langle M', s' \rangle \\
\langle E[M], s \rangle \rightarrow \langle E[M'], s' \rangle \\
\langle M, s \rangle \rightarrow \text{FAIL} \\
\langle E[M], s \rangle \rightarrow \text{FAIL}
\end{align*}
\]

where \( E \) ranges over evaluation contexts:

\[E ::= - \mid \text{let } x = E \text{ in } M \mid \text{ref } E \mid ! E \mid E := M \mid V ::= E \mid \ldots\]
\[
\begin{align*}
&\langle \text{let } r = \text{ref } \lambda x (x) \text{ in } \\
&\langle \text{let } u = (r := \lambda x' (\text{ref } !x')) \text{ in } \text{(!r)}(), \{\} \rangle \\
&\quad \rightarrow^* \langle \text{let } u = (r := \lambda x' (\text{ref } !x')) \text{ in } \text{(!r)}(), \{r \mapsto \lambda x (x)\} \rangle
\end{align*}
\]
let $r = \text{ref } \lambda x \ (x)$ in

\begin{align*}
\langle & \text{let } u = (r := \lambda x' \ (\text{ref } !x')) \text{ in } (!r)(), \ \{\} \rangle \\
\rightarrow^* & \langle \text{let } u = (r := \lambda x' \ (\text{ref } !x')) \text{ in } (!r)(), \ \{r \mapsto \lambda x \ (x)\} \rangle \\
\rightarrow^* & \langle (!r)(), \ \{r \mapsto \lambda x' \ (\text{ref } !x')\} \rangle
\end{align*}
let \( r = \text{ref } \lambda x (x) \) in
\[
\langle \text{let } u = (r := \lambda x' (\text{ref }!x')) \text{ in } (!r)() , \{\} \rangle
\]

\[\rightarrow^* \langle \text{let } u = (r := \lambda x' (\text{ref }!x')) \text{ in } (!r)() , \{r \mapsto \lambda x (x)\} \rangle\]

\[\rightarrow^* \langle (!r)() , \{r \mapsto \lambda x' (\text{ref }!x')\} \rangle\]

\[\rightarrow \langle \lambda x' (\text{ref }!x') () , \{r \mapsto \lambda x' (\text{ref }!x')\} \rangle\]
\[
\begin{align*}
\langle & \text{let } r = \text{ref } \lambda x (x) \text{ in} \\
& \langle \text{let } u = (r := \lambda x' (\text{ref } !x')) \text{ in } (!r)(()) , \{\} \rangle \\
\rightarrow^* & \langle \text{let } u = (r := \lambda x' (\text{ref } !x')) \text{ in } (!r)(()) , \{r \mapsto \lambda x (x)\} \rangle \\
\rightarrow^* & \langle (!r)(()) , \{r \mapsto \lambda x' (\text{ref } !x')\} \rangle \\
\rightarrow & \langle \lambda x' (\text{ref } !x') () , \{r \mapsto \lambda x' (\text{ref } !x')\} \rangle \\
\rightarrow & \langle \text{ref } !(()) , \{r \mapsto \lambda x' (\text{ref } !x')\} \rangle
\end{align*}
\]
\[
\begin{align*}
\langle & \text{let } r = \text{ref } \lambda x (x) \text{ in } \\
& \langle \text{let } u = (r := \lambda x' (\text{ref } x')) \text{ in } (\text{!r})(()) , \{\} \rangle \rangle \\
\rightarrow^* & \langle \text{let } u = (r := \lambda x' (\text{ref } x')) \text{ in } (\text{!r})(()) , \{ r \mapsto \lambda x (x) \} \rangle \\
\rightarrow^* & \langle (\text{!r})(()) , \{ r \mapsto \lambda x' (\text{ref } x') \} \rangle \\
\rightarrow & \langle \lambda x' (\text{ref } x') () , \{ r \mapsto \lambda x' (\text{ref } x') \} \rangle \\
\rightarrow & \langle \text{ref } !(()) , \{ r \mapsto \lambda x' (\text{ref } x') \} \rangle \\
\rightarrow & \quad \text{FAIL}
\end{align*}
\]
Example

\[ \sigma \triangleq \forall \alpha \ (\alpha \to \alpha) \text{ref} \]

The expression

\[
\begin{align*}
&\text{let } r = \text{ref } \lambda x \ (x) \text{ in} \\
&\text{let } u = (r := \lambda x' (\text{ref }!x')) \text{ in} \\
&(\!r)(())
\end{align*}
\]

has type \textit{unit}.

\[
\sigma \triangleright (\beta \text{ref} \to \beta \text{ref}) \text{ref}
\]

\[
\sigma \triangleright (\text{unit} \to \text{unit}) \text{ref}
\]
Value-restricted typing rule for \texttt{let}-expressions

\[
\text{(letv)} \quad \frac{\Gamma \vdash M_1 : \tau_1 \quad \Gamma, x : \forall A (\tau_1) \vdash M_2 : \tau_2}{\Gamma \vdash \text{let } x = M_1 \text{ in } M_2 : \tau_2}
\] (†)
Value-restricted typing rule for let-expressions

\[
\frac{\Gamma \vdash M_1 : \tau_1 \quad \Gamma, x : \forall A(\tau_1) \vdash M_2 : \tau_2}{\Gamma \vdash \text{let } x = M_1 \text{ in } M_2 : \tau_2}
\] (†)

(†) provided \( x \notin \text{dom}(\Gamma) \) and
Value-restricted typing rule for \texttt{let}-expressions

\[ (\text{letv}) \quad \frac{\Gamma \vdash M_1 : \tau_1 \quad \Gamma, x : \forall A (\tau_1) \vdash M_2 : \tau_2}{\Gamma \vdash \text{let } x = M_1 \text{ in } M_2 : \tau_2} \quad (\dagger) \]

(\dagger) provided \( x \notin \text{dom}(\Gamma) \) and

\[ A = \begin{cases} 
\{ \} & \text{if } M_1 \text{ is not a value} \\
\text{ftv}(\tau_1) - \text{ftv}(\Gamma) & \text{if } M_1 \text{ is a value}
\end{cases} \]

Recall that values are given by
\[ V ::= x \mid \lambda x(M) \mid () \mid \text{true} \mid \text{false} \mid \text{nil} \mid V :: V \]
Example

With (letv) rule, this gets type scheme

\[
\sigma' \triangleq \forall \cdot \{ (\alpha \to \alpha) \text{ref} \}
\]

The expression

```
let \(r = \text{ref} \lambda x \,(x)\) in
let \(u = (r := \lambda x' \,(\text{ref}!x'))\) in
(!r)(())
```

has type \textit{unit}. 
Example

with (letv) rule, this gets type scheme

\[ \sigma' \triangleq \forall \{ \} \left( (\alpha \rightarrow \alpha) \text{ref} \right) \]

The expression

\[
\text{let } r = \text{ref } \lambda x \ (x) \text{ in }
\text{let } u = (r := \lambda x' (\text{ref} \! x')) \text{ in }
(!r)()\
\]

has type \textit{unit}.

\[ \sigma' \not\vdash \left( \beta \text{ref} \rightarrow \beta \text{ref} \right) \text{ref} \]

\[ \sigma' \not\vdash \left( \text{unit} \rightarrow \text{unit} \right) \text{ref} \]
Type soundness for Midi-ML with the value restriction

For any closed Midi-ML expression $M$, if there is some type scheme $\sigma$ for which

$\vdash M : \sigma$

is provable in the value-restricted type system

$$(\var) + (\text{bool}) + (\text{if}) + (\text{nil}) + (\text{cons}) + (\text{case}) + (\text{fn}) + (\text{app}) + (\text{unit}) + (\text{ref}) + (\text{get}) + (\text{set}) + (\text{letv})$$

then evaluation of $M$ does not fail,
Type soundness for
Midi-ML with the value restriction

For any closed Midi-ML expression $M$, if there is some type scheme $\sigma$ for which

$$\vdash M : \sigma$$

is provable in the value-restricted type system

$$(\text{var } \triangleright) + (\text{bool}) + (\text{if}) + (\text{nil}) + (\text{cons}) + (\text{case}) + (\text{fn}) +
(\text{app}) + (\text{unit}) + (\text{ref}) + (\text{get}) + (\text{set}) + (\text{letv})$$

then \textit{evaluation of }$M$\textit{ does not fail},
i.e. there is no sequence of transitions of the form

$$\langle M, \{ \} \rangle \rightarrow \cdots \rightarrow \text{FAIL}$$

for the transition system $\rightarrow$ defined in Figure 4
(where $\{ \}$ denotes the empty state).
Type soundness for Midi-ML with the value restriction

For any closed Midi-ML expression $M$, if there is some type scheme $\sigma$ for which

$$\vdash M : \sigma$$

is provable in the value-restricted type system

$$(\text{var } \rhd) + (\text{bool}) + (\text{if}) + (\text{nil}) + (\text{cons}) + (\text{case}) + (\text{fn}) + (\text{app}) + (\text{unit}) + (\text{ref}) + (\text{get}) + (\text{set}) + (\text{letv})$$

then \textit{evaluation of }$M$\textit{ does not fail}, (and typing is preserved by }$\rightarrow$	extit{)}

i.e. there is no sequence of transitions of the form

$$\langle M, \{ \} \rangle \rightarrow \cdots \rightarrow \text{FAIL}$$

for the transition system $\rightarrow$ defined in Figure 4 (where $\{ \}$ denotes the empty state).
In Midi-ML’s value-restricted type system, some expressions that were typeable using \(\text{let}\) become untypeable using \(\text{letv}\).
In Midi-ML’s value-restricted type system, some expressions that were typeable using (let) become untypeable using (letv).

For example (exercise):

\[
\text{let } f = (\lambda x. x) \lambda y. y \text{ in } (f \text{ true}) :: (f \text{ nil})
\]
In Midi-ML’s value-restricted type system, some expressions that were typeable using \(\texttt{let}\) become untypeable using \(\texttt{letv}\).

For example (exercise):

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

But one can often\(^1\) use \(\eta\)-expansion

replace \(M\) by \(\lambda x (M \, x)\) (where \(x \notin \text{fv}(M)\))

or \(\beta\)-reduction

replace \((\lambda x (M)) \, N\) by \(M[N/x]\)

to get around the problem.

(\(^1\) *These* transformations do not always preserve meaning [contextual equivalence].)