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

τ	$::=$	$\ldots$	
	$\mid$	unit	unit type
	$\mid$	$\tau\ \mathit{ref}$	reference type
M	$::=$	$\ldots$	
	$\mid$	$()$	unit value
	$\mid$	$\texttt{ref}\ M$	reference creation
	$\mid$	$!M$	dereference
	$\mid$	$M := M$	assignment

Midi-ML's extra typing rules

$$\textbf{(unit)}\ \frac{}{\Gamma \vdash () : \mathit{unit}}$$

Midi-ML's extra typing rules

$$\textbf{(unit)}\ \frac{}{\Gamma \vdash () : \mathit{unit}}$$

$$\textbf{(ref)}\ \frac{\Gamma \vdash M : \tau}{\Gamma \vdash \texttt{ref}\ M : \tau\ \mathit{ref}}$$

Midi-ML's extra typing rules

$$(\textbf{unit})\ \frac{}{\Gamma \vdash () : \mathit{unit}}$$

$$(\textbf{ref})\ \frac{\Gamma \vdash M : \tau}{\Gamma \vdash \texttt{ref}\ M : \tau\ \mathit{ref}}$$

$$(\textbf{get})\ \frac{\Gamma \vdash M : \tau\ \mathit{ref}}{\Gamma \vdash\ !M : \tau}$$

Midi-ML's extra typing rules

$$(\textbf{unit})\ \frac{}{\Gamma \vdash () : \mathit{unit}}$$

$$(\textbf{ref})\ \frac{\Gamma \vdash M : \tau}{\Gamma \vdash \texttt{ref}\ M : \tau\ \mathit{ref}}$$

$$(\textbf{get})\ \frac{\Gamma \vdash M : \tau\ \mathit{ref}}{\Gamma \vdash\ !M : \tau}$$

$$(\textbf{set})\ \frac{\Gamma \vdash M_1 : \tau\ \mathit{ref} \qquad \Gamma \vdash M_2 : \tau}{\Gamma \vdash M_1 := M_2 : \mathit{unit}}$$

Example

The expression

$$\begin{array}{l}\texttt{let}\, r = \texttt{ref}\, \lambda x\,(x)\,\texttt{in} \\ \quad \texttt{let}\, u = (r := \lambda x'\,(\texttt{ref}\, !x'))\,\texttt{in} \\ \quad\quad (!r)\,()\end{array}$$

has type $unit$.

Example

$: \forall\alpha\, ((\alpha \to \alpha)\, \mathrm{ref})$

The expression

$$
\begin{array}{l}
\texttt{let}\ r = \texttt{ref}\ \lambda x\,(x)\ \texttt{in} \\
\quad \texttt{let}\ u = (r := \lambda x'\,(\texttt{ref}\ !x'))\ \texttt{in} \\
\qquad (!r)\,()
\end{array}
$$

has type unit.

Example

$: \forall\alpha\,((\alpha \to \alpha)\,\mathrm{ref})$

$: \beta\,\mathrm{ref} \to \beta\,\mathrm{ref}$

The expression

$$
\begin{array}{l}
\mathtt{let}\; r = \mathtt{ref}\; \lambda x\,(x)\; \mathtt{in} \\
\quad \mathtt{let}\; u = (r := \lambda x'\,(\mathtt{ref}\; !x'))\; \mathtt{in} \\
\quad\quad (!r)\,()
\end{array}
$$

has type ***unit***.

Example

$: \forall\alpha\,((\alpha \to \alpha)\,ref)$

The expression

$$\begin{array}{l}
\texttt{let}\; r = \texttt{ref}\; \lambda x\,(x)\; \texttt{in} \\
\quad \texttt{let}\; u = (r := \lambda x'\,(\texttt{ref}\; !x'))\; \texttt{in} \\
\qquad (!r)\,()
\end{array}$$

$: (\beta\, ref \to \beta\, ref)\, ref$

has type ***unit***.

Example

$: \forall \alpha \, ((\alpha \to \alpha) \, ref)$

The expression

$$\mathtt{let}\ r = \mathtt{ref}\ \lambda x\,(x)\ \mathtt{in}$$
$$\quad \mathtt{let}\ u = (r := \lambda x'\,(\mathtt{ref}\ !x'))\ \mathtt{in}$$
$$\quad\quad (!r)\,()$$

$: (\beta \, ref \to \beta \, ref) \, ref$

$: (unit \to unit) \, ref$

has type $\boldsymbol{unit}$.

Example

$$\sigma \triangleq \forall \alpha \, ((\alpha \to \alpha)\, ref)$$

The expression

$$\texttt{let}\; r = \texttt{ref}\; \lambda x\,(x)\; \texttt{in}$$
$$\quad \texttt{let}\; u = (r := \lambda x'\,(\texttt{ref}\; !x'))\; \texttt{in}$$
$$\qquad (!r)\,()$$

has type ***unit***.

$$\sigma > (\beta\, ref \to \beta\, ref)\, ref$$

$$\sigma > (unit \to unit)\, ref$$

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

$$\langle M, s \rangle \to \langle M', s' \rangle$$
$$\langle M, s \rangle \to FAIL$$

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:

$$V ::= x \mid \lambda x\,(M) \mid () \mid \texttt{true} \mid \texttt{false} \mid \texttt{nil} \mid V :: V$$

Midi-ML transition system

Small-step transition relations

$$\langle M, s \rangle \to \langle M', s' \rangle$$
$$\langle M, s \rangle \to FAIL$$

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:

$$V ::= x \mid \lambda x\,(M) \mid () \mid \texttt{true} \mid \texttt{false} \mid \texttt{nil} \mid V :: V$$

- 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 $FAIL$ represents a run-time error

Midi-ML transition system

Small-step transition relations

$$\langle M, s\rangle \rightarrow \langle M', s'\rangle$$
$$\langle M, s\rangle \rightarrow \mathit{FAIL}$$

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:

$$V ::= x \mid \lambda x\,(M) \mid () \mid \mathtt{true} \mid \mathtt{false} \mid \mathtt{nil} \mid V :: V$$

- 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 FAIL represents a run-time error

are inductively defined by syntax-directed rules. . .

Midi-ML transitions involving references

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

where V ranges over values:

$$V ::= x \mid \lambda x\,(M) \mid () \mid \texttt{true} \mid \texttt{false} \mid \texttt{nil} \mid V :: V$$

Midi-ML transitions involving references

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

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

where V ranges over values:

$$V ::= x \mid \lambda x\,(M) \mid () \mid \texttt{true} \mid \texttt{false} \mid \texttt{nil} \mid V :: V$$

Midi-ML transitions involving references

$\langle !x, s\rangle \to \langle s(x), s\rangle$ if $x \in dom(s)$

$\langle !V, s\rangle \to FAIL$ if V not a variable

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

where V ranges over values:

$$V ::= x \mid \lambda x\,(M) \mid () \mid \texttt{true} \mid \texttt{false} \mid \texttt{nil} \mid V :: V$$

Midi-ML transitions involving references

$$\langle !x, s\rangle \to \langle s(x), s\rangle \quad \text{if } x \in dom(s)$$

$$\langle !V, s\rangle \to FAIL \quad \text{if } V \text{ not a variable}$$

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

$$\langle V := V', s\rangle \to FAIL \quad \text{if } V \text{ not a variable}$$

where V ranges over values:

$$V ::= x \mid \lambda x\,(M) \mid () \mid \texttt{true} \mid \texttt{false} \mid \texttt{nil} \mid V :: V$$

Midi-ML transitions involving references

$$\langle !x, s\rangle \to \langle s(x), s\rangle \quad \text{if } x \in dom(s)$$

$$\langle !V, s\rangle \to FAIL \quad \text{if } V \text{ not a variable}$$

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

$$\langle V := V', s\rangle \to FAIL \quad \text{if } V \text{ not a variable}$$

$$\langle \texttt{ref}\ V, s\rangle \to \langle x, s[x \mapsto V]\rangle \quad \text{if } x \notin dom(s)$$

where V ranges over values:

$$V ::= x \mid \lambda x\,(M) \mid () \mid \texttt{true} \mid \texttt{false} \mid \texttt{nil} \mid V :: V$$

[fig.4, page 28]

$$\frac{\langle M, s\rangle \to \langle M', s'\rangle}{\langle \mathcal{E}[M], s\rangle \to \langle \mathcal{E}[M'], s'\rangle}$$

$$\frac{\langle M, s\rangle \to \text{FAIL}}{\langle \mathcal{E}[M], s\rangle \to \text{FAIL}}$$

where $\mathcal{E}$ ranges over evaluation contexts:

$$\mathcal{E} ::= - \mid \text{let } x = \mathcal{E} \text{ in } M \mid \text{ref } \mathcal{E} \mid\ !\mathcal{E} \mid \mathcal{E} := M \mid V := \mathcal{E} \mid \cdots$$

$$\left\langle \begin{array}{l}\texttt{let}\ r = \texttt{ref}\ \lambda x\,(x)\ \texttt{in} \\ \texttt{let}\ u = (r := \lambda x'\,(\texttt{ref}\ !x'))\ \texttt{in}\ (!r)()\ ,\ \{\} \\ \end{array} \right\rangle$$

$$\to^{*} \quad \langle \texttt{let}\ u = (r := \lambda x'\,(\texttt{ref}\ !x'))\ \texttt{in}\ (!r)()\ ,\ \{r \mapsto \lambda x\,(x)\}\rangle$$

$$\left\langle \begin{array}{l} \texttt{let}\, r = \texttt{ref}\, \lambda x\,(x)\ \texttt{in} \\ \texttt{let}\, u = (r := \lambda x'\,(\texttt{ref}\ !x'))\ \texttt{in}\ (!r)\,()\ ,\ \{\} \end{array} \right\rangle$$

$$\to^* \quad \langle \texttt{let}\, u = (r := \lambda x'\,(\texttt{ref}\ !x'))\ \texttt{in}\ (!r)\,()\ ,\ \{r \mapsto \lambda x\,(x)\}\rangle$$

$$\to^* \quad \langle (!r)\,()\ ,\ \{r \mapsto \lambda x'\,(\texttt{ref}\ !x')\}\rangle$$

$$\left\langle \begin{array}{l} \texttt{let}\ r = \texttt{ref}\ \lambda x\,(x)\ \texttt{in} \\ \texttt{let}\ u = (r := \lambda x'\,(\texttt{ref}\ !x'))\ \texttt{in}\ (!r)\,()\ ,\ \{\} \end{array} \right\rangle$$

$$\to^{*} \quad \langle \texttt{let}\ u = (r := \lambda x'\,(\texttt{ref}\ !x'))\ \texttt{in}\ (!r)\,()\ ,\ \{r \mapsto \lambda x\,(x)\}\rangle$$

$$\to^{*} \quad \langle (!r)\,()\ ,\ \{r \mapsto \lambda x'\,(\texttt{ref}\ !x')\}\rangle$$

$$\to \quad \langle \lambda x'\,(\texttt{ref}\ !x')\,()\ ,\ \{r \mapsto \lambda x'\,(\texttt{ref}\ !x')\}\rangle$$

$$\left\langle \begin{array}{l} \texttt{let}\ r = \texttt{ref}\ \lambda x\,(x)\ \texttt{in} \\ \texttt{let}\ u = (r := \lambda x'\,(\texttt{ref}\,!x'))\ \texttt{in}\ (!r)\,()\ ,\ \{\} \end{array} \right\rangle$$

$$\to^{*} \quad \langle \texttt{let}\ u = (r := \lambda x'\,(\texttt{ref}\,!x'))\ \texttt{in}\ (!r)\,()\ ,\ \{r \mapsto \lambda x\,(x)\}\rangle$$

$$\to^{*} \quad \langle (!r)\,()\ ,\ \{r \mapsto \lambda x'\,(\texttt{ref}\,!x')\}\rangle$$

$$\to \quad \langle \lambda x'\,(\texttt{ref}\,!x')\,()\ ,\ \{r \mapsto \lambda x'\,(\texttt{ref}\,!x')\}\rangle$$

$$\to \quad \langle \texttt{ref}\,!()\ ,\ \{r \mapsto \lambda x'\,(\texttt{ref}\,!x')\}\rangle$$

$$\left\langle \begin{array}{l} \texttt{let}\ r = \texttt{ref}\ \lambda x\,(x)\ \texttt{in} \\ \texttt{let}\ u = (r := \lambda x'\,(\texttt{ref}\ !x'))\ \texttt{in}\ (!r)\,()\ ,\ \{\} \end{array} \right\rangle$$

$$\rightarrow^* \quad \langle \texttt{let}\ u = (r := \lambda x'\,(\texttt{ref}\ !x'))\ \texttt{in}\ (!r)\,()\ ,\ \{r \mapsto \lambda x\,(x)\}\rangle$$

$$\rightarrow^* \quad \langle (!r)\,()\ ,\ \{r \mapsto \lambda x'\,(\texttt{ref}\ !x')\}\rangle$$

$$\rightarrow \quad \langle \lambda x'\,(\texttt{ref}\ !x')\ ()\ ,\ \{r \mapsto \lambda x'\,(\texttt{ref}\ !x')\}\rangle$$

$$\rightarrow \quad \langle \texttt{ref}\ !()\ ,\ \{r \mapsto \lambda x'\,(\texttt{ref}\ !x')\}\rangle$$

$$\rightarrow \quad \textit{FAIL}$$

Example

$\sigma \triangleq \forall \alpha\, ((\alpha \to \alpha)\,\mathrm{ref})$

The expression

$$\begin{array}{l} \mathtt{let}\; r = \mathtt{ref}\; \lambda x\,(x)\;\mathtt{in} \\ \quad \mathtt{let}\; u = (r := \lambda x'\,(\mathtt{ref}\; !x'))\;\mathtt{in} \\ \qquad (!r)\,() \end{array}$$

$\sigma > (\beta\,\mathrm{ref} \to \beta\,\mathrm{ref})\,\mathrm{ref}$

$\sigma > (\mathrm{unit} \to \mathrm{unit})\,\mathrm{ref}$

has type ***unit***.

Value-restricted typing rule for `let`-expressions

$$\textbf{(letv)}\ \frac{\Gamma \vdash M_1 : \tau_1 \qquad \Gamma, x : \forall A\,(\tau_1) \vdash M_2 : \tau_2}{\Gamma \vdash \texttt{let}\ x = M_1\ \texttt{in}\ M_2 : \tau_2} \quad (\dagger)$$

Value-restricted typing rule for `let`-expressions

$$\textbf{(letv)}\ \frac{\Gamma \vdash M_1 : \tau_1 \qquad \Gamma, x : \forall A\,(\tau_1) \vdash M_2 : \tau_2}{\Gamma \vdash \texttt{let}\, x = M_1\, \texttt{in}\, M_2 : \tau_2}\ (\dagger)$$

($\dagger$) provided $x \notin dom(\Gamma)$ and

Value-restricted typing rule for `let`-expressions

$$(\mathbf{letv})\ \frac{\Gamma \vdash M_1 : \tau_1 \qquad \Gamma, x : \forall A\,(\tau_1) \vdash M_2 : \tau_2}{\Gamma \vdash \texttt{let}\, x = M_1\, \texttt{in}\, M_2 : \tau_2}\quad (\dagger)$$

($\dagger$) provided $x \notin dom(\Gamma)$ and

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

Recall that values are given by
$V ::= x \mid \lambda x\,(M) \mid () \mid \texttt{true} \mid \texttt{false} \mid \texttt{nil} \mid V :: V$

Example

with (letv) rule, this gets type scheme

$$\sigma' \triangleq \forall \{\} ((\alpha \to \alpha)\, ref)$$

The expression

$$\begin{array}{l} \texttt{let}\ r = \texttt{ref}\ \lambda x\,(x)\ \texttt{in} \\ \quad \texttt{let}\ u = (r := \lambda x'\,(\texttt{ref}\ !x'))\ \texttt{in} \\ \qquad (!r)\,() \end{array}$$

has type unit.

Example

with (letv) rule, this gets type scheme

$\sigma' \triangleq \forall \{\} ((\alpha \to \alpha)\ ref)$

The expression

$$\texttt{let}\ r = \texttt{ref}\ \lambda x\,(x)\ \texttt{in}$$
$$\quad \texttt{let}\ u = (r := \lambda x'\,(\texttt{ref}\ !x'))\ \texttt{in}$$
$$\qquad (!r)\,()$$

has type ***unit***.

$\sigma' \succ (\beta\ ref \to \beta\ ref)\ ref$

$\sigma' \succ (unit \to unit)\ ref$

Type soundness for Midi-ML with the value restriction

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

$$\vdash M : \sigma$$

is provable in the value-restricted type system

$$\begin{array}{l}(\mathbf{var} \succ) + (\mathbf{bool}) + (\mathbf{if}) + (\mathbf{nil}) + (\mathbf{cons}) + (\mathbf{case}) + (\mathbf{fn}) + \\ (\mathbf{app}) + (\mathbf{unit}) + (\mathbf{ref}) + (\mathbf{get}) + (\mathbf{set}) + (\mathbf{letv})\end{array}$$

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 σ for which

$$\vdash M : \sigma$$

is provable in the value-restricted type system

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

then *evaluation of M does not fail*,
i.e. there is no sequence of transitions of the form

$$\langle M, \{\,\}\rangle \to \cdots \to FAIL$$

for the transition system $\to$ 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 σ for which

$$\vdash M : \sigma$$

is provable in the value-restricted type system

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

then *evaluation of M does not fail*, (and typing is preserved by $\to$)
i.e. there is no sequence of transitions of the form

$$\langle M, \{\,\}\rangle \to \cdots \to \mathit{FAIL}$$

for the transition system $\to$ defined in Figure 4
(where $\{\,\}$ denotes the empty state).

In Midi-ML's value-restricted type system, some expressions that were typeable using **(let)** become untypeable using **(letv)**.

In Midi-ML's value-restricted type system, some expressions that were typeable using $(\mathbf{let})$ become untypeable using $(\mathbf{letv})$.

For example (exercise):

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

In Midi-ML's value-restricted type system, some expressions that were typeable using **(let)** become untypeable using **(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 η-expansion

replace M by $\lambda x\,(M\,x)$ (where $x \notin fv(M)$)

or β-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].)