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	::= 	 unit τ ref	unit type reference type
Μ	::=	 () ref <i>M</i> ! <i>M</i> <i>M</i> := <i>M</i>	unit value reference creation dereference assignment

(unit) $\Gamma \vdash (): unit$

$$(unit) \frac{\Gamma \vdash (): unit}{\Gamma \vdash M: \tau}$$
$$(ref) \frac{\Gamma \vdash M: \tau}{\Gamma \vdash ref M: \tau ref}$$

(unit)

$$\Gamma \vdash (): unit$$
(ref)

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

$$\frac{\Gamma \vdash M: \tau ref}{\Gamma \vdash !M: \tau}$$

$$(unit) \qquad \Gamma \vdash () : unit$$

$$(ref) \qquad \Gamma \vdash M : \tau \\ \hline \Gamma \vdash ref M : \tau ref$$

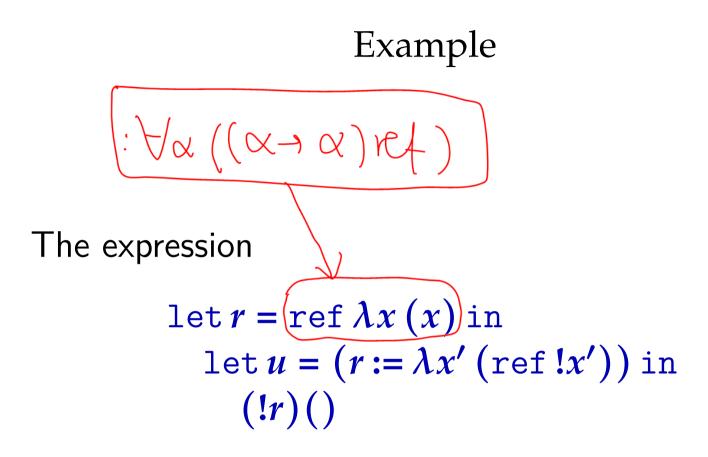
$$(get) \qquad \Gamma \vdash M : \tau ref \\ \hline \Gamma \vdash !M : \tau$$

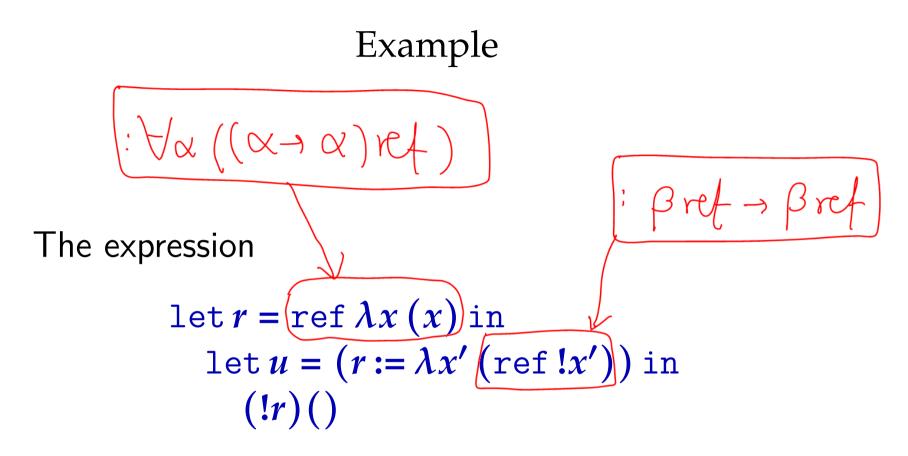
$$(set) \qquad \Gamma \vdash M_1 : \tau ref \quad \Gamma \vdash M_2 : \tau \\ \hline \Gamma \vdash M_1 := M_2 : unit$$

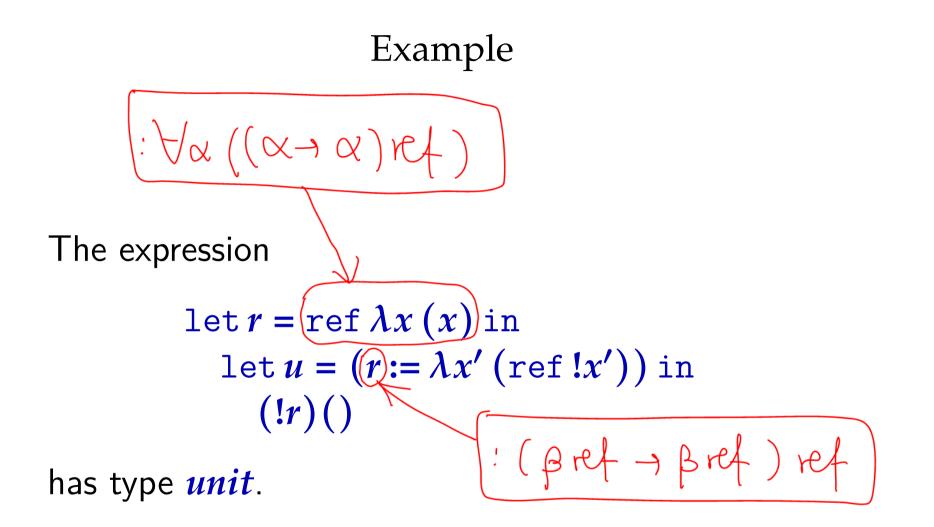
Example

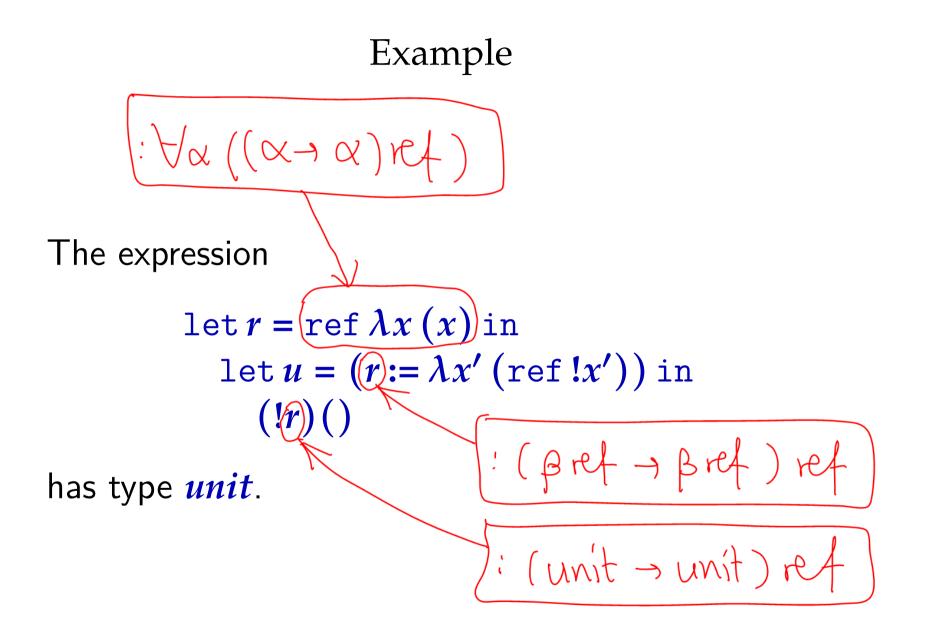
```
The expression
```

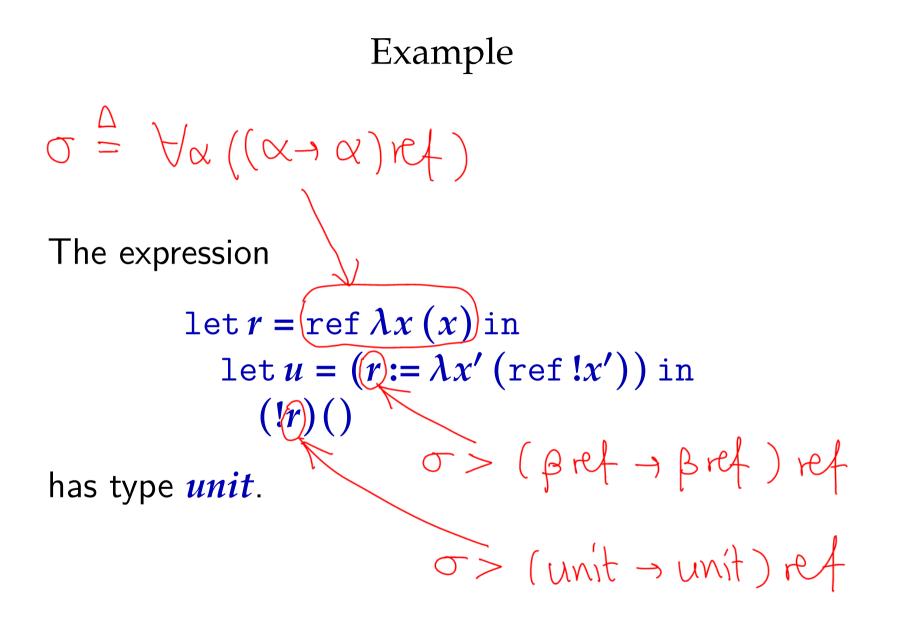
let r = ref
$$\lambda x(x)$$
 in
let u = (r := $\lambda x'(ref!x')$) in
(!r)()











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.

Small-step transition relations

$$\langle M, s \rangle \rightarrow \langle M', s' \rangle$$

 $\langle M, s \rangle \rightarrow FAIL$

Small-step transition relations

$$\langle M, s \rangle \rightarrow \langle M', s' \rangle$$

 $\langle M, s \rangle \rightarrow FAIL$

where

 \blacktriangleright *M*, *M'* range over Midi-ML expressions

▶ s, s' range over states = finite functions $s = \{x_1 \mapsto V_1, \dots, x_n \mapsto V_n\}$ mapping variables x_i to values V_i :

Small-step transition relations

$$\langle M, s \rangle \rightarrow \langle M', s' \rangle$$

 $\langle M, s \rangle \rightarrow FAIL$

where

• M, M' range over Midi-ML expressions

▶ s, s' range over states = finite functions $s = \{x_1 \mapsto V_1, \dots, x_n \mapsto V_n\}$ mapping variables x_i to values V_i :

- configurations (M, s) 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

Small-step transition relations

$$\langle M, s \rangle \rightarrow \langle M', s' \rangle$$

 $\langle M, s \rangle \rightarrow FAIL$

where

• M, M' range over Midi-ML expressions

▶ s, s' range over states = finite functions $s = \{x_1 \mapsto V_1, \dots, x_n \mapsto V_n\}$ 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 FAIL represents a run-time error

are inductively defined by syntax-directed rules...

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

where V ranges over values:

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

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

where V ranges over values:

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

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

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

where V ranges over values:

 $\langle !x, s \rangle \rightarrow \langle s(x), s \rangle$ if $x \in dom(s)$ $\langle !V, s \rangle \rightarrow FAIL$ if V not a variable $\langle x := V', s \rangle \rightarrow \langle (), s[x \mapsto V'] \rangle$ $\langle V := V', s \rangle \rightarrow FAIL$ if V not a variable

where V ranges over values:

 $\langle !x, s \rangle \to \langle s(x), s \rangle \quad \text{if } x \in dom(s)$ $\langle !V, s \rangle \to FAIL \quad \text{if } V \text{ not a variable}$ $\langle x := V', s \rangle \to \langle (), s[x \mapsto V'] \rangle$ $\langle V := V', s \rangle \to FAIL \quad \text{if } V \text{ not a variable}$ $\langle \operatorname{ref} V, s \rangle \to \langle x, s[x \mapsto V] \rangle \quad \text{if } x \notin dom(s)$

where V ranges over values:

[fig.4, page 28]

$$\frac{\langle M_{1}s \rangle \rightarrow \langle M'_{1}s' \rangle}{\langle E[M]_{1}s \rangle \rightarrow \langle E[M']_{1}s' \rangle}$$

$$\frac{\langle M_{1}s \rangle \rightarrow \langle E[M']_{1}s' \rangle}{\langle E[M]_{1}s \rangle \rightarrow FAIL}$$
where E ranges over evaluation contexts:

$$\mathcal{E} := - | \det x = \mathcal{E} \text{ in } M | \operatorname{ref} \mathcal{E} | ! \mathcal{E} | \mathcal{E} := \mathcal{M} | V :: = \mathcal{E} | ...$$

$$\left\langle \operatorname{let} r = \operatorname{ref} \lambda x(x) \operatorname{in} \\ \operatorname{let} u = (r := \lambda x' (\operatorname{ref} ! x')) \operatorname{in} (!r)(), \{\} \right\rangle$$

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

$$\left\langle \operatorname{let} r = \operatorname{ref} \lambda x(x) \operatorname{in} \\ \operatorname{let} u = (r := \lambda x' (\operatorname{ref} ! x')) \operatorname{in} (!r)(), \{\} \right\rangle$$

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

$$ightarrow^* \hspace{0.2cm} \langle (!r)() \hspace{0.1cm}$$
, $\{r\mapsto \lambda x' \hspace{0.1cm} (ext{ref} \hspace{0.1cm} !x')\}
angle$

$$\left\langle \operatorname{let} r = \operatorname{ref} \lambda x (x) \operatorname{in} \\ \operatorname{let} u = (r := \lambda x' (\operatorname{ref} ! x')) \operatorname{in} (!r)(), \{\} \right\rangle$$

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

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

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

$$\left\langle \operatorname{let} r = \operatorname{ref} \lambda x (x) \operatorname{in} \\ \operatorname{let} u = (r := \lambda x' (\operatorname{ref} ! x')) \operatorname{in} (!r)(), \{\} \right\rangle$$

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

$$ightarrow^* \ \langle (!r)() , \{r \mapsto \lambda x' \, (ext{ref} \, !x')\}
angle$$

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

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

$$\left\langle \operatorname{let} r = \operatorname{ref} \lambda x (x) \operatorname{in} \\ \operatorname{let} u = (r := \lambda x' (\operatorname{ref} ! x')) \operatorname{in} (!r)(), \{\} \right\rangle$$

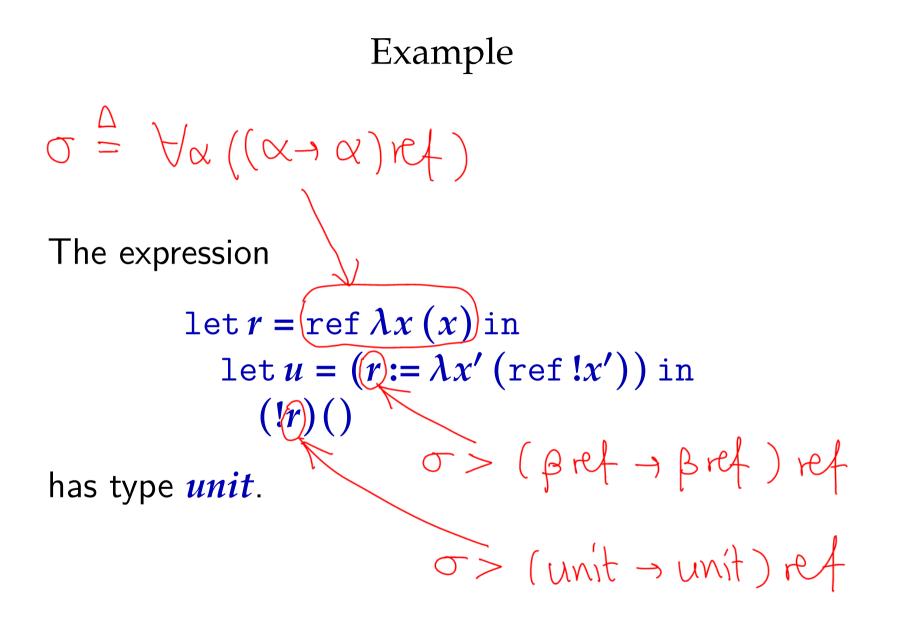
$$\rightarrow^{*} \quad \langle \operatorname{let} u = (r := \lambda x' (\operatorname{ref} ! x')) \operatorname{in} (!r)(), \{r \mapsto \lambda x (x)\} \rangle$$

$$ightarrow^* \langle (!r)(), \{r \mapsto \lambda x' (\operatorname{ref} !x')\}
angle$$

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

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

\rightarrow FAIL



Value-restricted typing rule for let-expressions

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

Value-restricted typing rule for let-expressions

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

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

Value-restricted typing rule for let-expressions

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

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

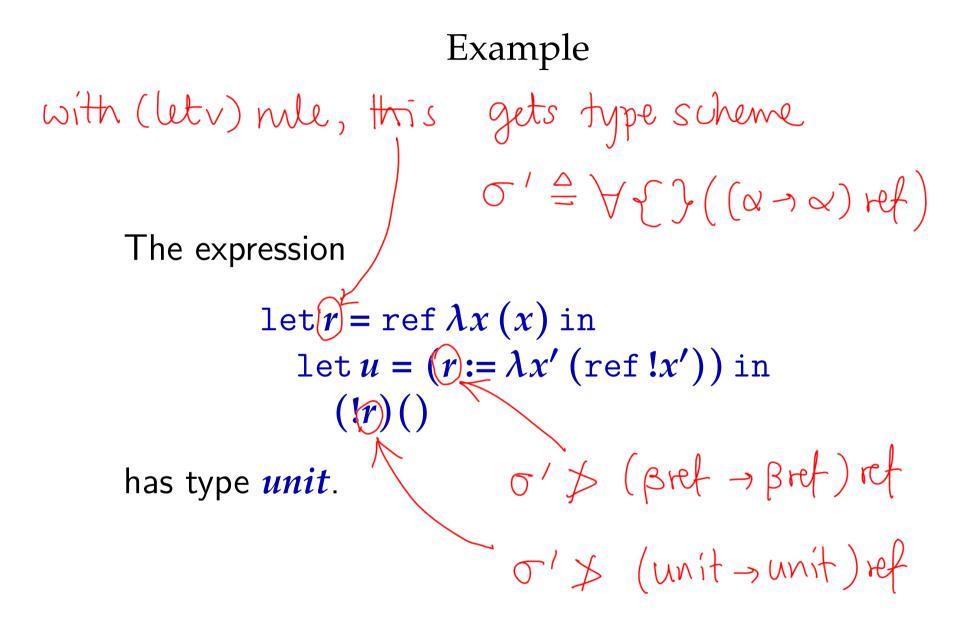
$$A = \begin{cases} \{ \} & \text{if } M_1 \text{ is not a value} \\ ftv(\tau_1) - ftv(\Gamma) & \text{if } M_1 \text{ is a value} \end{cases}$$

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

Example
with (letv) rule, this gets type scheme

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

The expression
 $\operatorname{let} r = \operatorname{ref} \lambda x (x) \operatorname{in}$
 $\operatorname{let} u = (r := \lambda x' (\operatorname{ref} ! x')) \operatorname{in}$
 $(!r)()$



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

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

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 σ for which

$\vdash M : \sigma$

is provable in the value-restricted type system

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

then evaluation of M does not fail,

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

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

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 σ for which

$\vdash M : \sigma$

is provable in the value-restricted type system

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

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

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

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 (let) become untypeable using (letv).

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

For example (exercise):

 $let f = (\lambda x (x)) \lambda y (y) in (f true) :: (f nil)$

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

For example (exercise):

 $let f = (\lambda x (x)) \lambda y (y) in (f true) :: (f nil)$

But one can often¹ 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].)