Mutable Storage

COS 441
Princeton University
Fall 2004

Impure Languages

- MinML does not have support for mutable values
- MinML is a pure language it contains no effects
  - Lack of effects means evaluation order doesn’t really matter
- ML has effects
  - Control effects (we will come back to these)
  - Store effects changes to “state”

Changes to Syntax of MinML

**Types**

\[ \tau ::= \text{int} \mid \text{bool} \mid \tau_1 \rightarrow \tau_2 \mid \text{ref} \]

**Expr’s**

\[ e ::= \text{var}(c_1, \ldots, c_n) \mid \text{true} \mid \text{false} \mid\]
\[ \text{if } c_1 \text{ then } c_2 \text{ else } c_3 \mid \]
\[ \text{fun } (x : \tau_1) : \tau_2 \mid \text{is } c \mid \]
\[ \text{apply}(c_1, c_2) \mid \]
\[ \text{ref}(c) \mid e \mid c_1 := c_2 \]

Locations and Memory

- Locations (l) are like names/variables abstract values
  - Not user visible only used in internal semantics
- We only care about two locations being the same not what they actually are
  - Key property not true of C pointers
- Memory (M) mapping from locations to values

Modified Transition Semantics

- Set of states is pairs of memory and closed expressions \((M,e)\)
- Set of final states \((F)\) are values and memories \((M,v)\)

\[ v ::= l \mid \text{true} \mid \text{false} \mid n \mid \text{fun}(\tau_1, \tau_2, x,e) \]

Primitive Instructions

\[ \frac{(l \notin \text{dom}(M))}{(M, \text{ref}(v)) \leftrightarrow (M[l=v], l)} \]
\[ \frac{(l \in \text{dom}(M))}{(M, ! l) \leftrightarrow (M, M(l))} \]
\[ \frac{(l \in \text{dom}(M))}{(M, l := v) \leftrightarrow (M[l=v], v)} \]
Search Rules
\[
\begin{align*}
(M,e) & \mapsto (M',e') \\
(M,\text{ref}(e)) & \mapsto (M',\text{ref}(e')) \\
(M, e) & \mapsto (M', e') \\
(M, !e) & \mapsto (M', !e')
\end{align*}
\]

Search Rules (cont.)
\[
\begin{align*}
(M,e_1) & \mapsto (M', e'_1) \\
(M,e_1 := e_2) & \mapsto (M', e'_1 := e_2) \\
(M,e_2) & \mapsto (M', e'_2) \\
(M,v_1 := e_2) & \mapsto (M', v_1 := e'_2)
\end{align*}
\]

Modified M-Machine Rules
\[
\begin{align*}
+(m,n) & \mapsto m + n \\
e_2 & \mapsto e'_2 \\
e_1 & \mapsto e'_1 \\
+(v_1,e_2) & \mapsto +(v_1,e'_2) \\
+(e_1,e_2) & \mapsto +(e'_1,e'_2)
\end{align*}
\]

Store Typing
- Store typing ($\Lambda$) maps locations to their type
- Need to prove stronger induction hypothesis for type safety proof
- Never extended in type checking rules
- Used to verify a memory is wellformed

Typing Rules for Expressions
\[
\begin{align*}
(\Lambda(l) = \tau) & \\
\Lambda;\Gamma \vdash l : \tau & \\
\Lambda;\Gamma \vdash e : \tau & \\
\Lambda;\Gamma \vdash \text{ref}(e) : \tau & \\
\Lambda;\Gamma \vdash e : \tau & \\
\Lambda;\Gamma \vdash !e : \tau & \\
\Lambda;\Gamma \vdash e_1 : \tau_2 & \\
\Lambda;\Gamma \vdash e_2 : \tau_2 & \\
\Lambda;\Gamma \vdash e_1 := e_2 : \tau_2
\end{align*}
\]

Modified Typing Rules
\[
\begin{align*}
\Gamma \vdash e_1 : \text{int} & \\
\Gamma \vdash e_2 : \text{int} & \\
\Lambda;\Gamma \vdash +(e_1,e_2) : \text{int}
\end{align*}
\]
Well-Formed Store

\[ \text{dom}(M) = \text{dom}(\Lambda) \quad \forall l \in \text{dom}(\Lambda) \quad \Lambda(l) \vdash M(l) : \Lambda(l) \]
\[ \vdash M : \Lambda \]

- Defines when every location in the store has the right type
- Allows for circular references!
  - More about this later

Well-Formed Program

\[ \vdash M : \Lambda \quad \Lambda ; \vdash e : \tau \]
\[ (M, e) \text{ ok} \]

- Program is well-formed if memory is well-formed with respect to a particular store-typing under which the expression is typeable

Modified Soundness Proof

Theorem 14.2 (Preservation)
If \((M, e) \text{ ok and } (M', e') \subseteq (M, e')\), then \((M', e') \text{ ok} \).

Proof: The trick is to prove a stronger result by induction on evaluation:
if \((M, e) \subseteq (M', e')\), \(\vdash M : \Lambda\) and \(\Lambda ; \vdash e : \tau\), then there exists \(\Lambda' \supseteq \Lambda\) such that \(\vdash M' : \Lambda'\) and \(\Lambda' ; \vdash e' : \tau'\).

Theorem 14.4 (Progress)
If \((M, e) \text{ ok then either } (M', e') \text{ is a final state or there exists } (M', e') \text{ such that } (M', e') \subseteq (M, e')\).

Proof: The proof is by induction on typing: if \(\vdash M : \Lambda\) and \(\Lambda ; \vdash e : \tau\), then either \(e\) is a value or there exists \(\Lambda' \supseteq \Lambda\) and \(\vdash \) such that \((M', e') \subseteq (M, e')\).

Ref’s and Typed \(\lambda\)-Calculus

- Introduce typed version of \(\lambda\)-calculus
- Discuss properties of TLC and how references interact with it

Simply Typed \(\lambda\)-calculus

Names \( x \in \ldots \)
Types \( \tau ::= \text{unit} \mid \tau_1 \to \tau_2 \)
Expr’s \( e ::= () \mid (\lambda x.e) \mid (e_1 e_2) \mid x \)
Values \( v ::= () \mid (\lambda x.e) \mid t \)

\[ \Gamma \vdash x : \tau \]
\[ \Gamma \vdash (\lambda x.e) : \tau_1 \to \tau_2 \]
\[ \Gamma \vdash e_1 : \tau_1 \to \tau_2 \]
\[ \Gamma \vdash e_2 : \tau_1 \]

Dynamic semantics same as untyped version

Typing Puzzles

How can we type the following terms?

- \(x\)
- \((\lambda x.x)\)
- \(((\lambda x.(x x)) (\lambda x.(x x)))\)
Typing Puzzles
How can we type the following terms?

?? ⊢ x : ??
?? ⊢ (λx.x) : ??
?? ⊢ ((λx.(x x)) (λx.(x x))) : ??

Typing Puzzles
How can we type the following terms?

{x→unit} ⊢ x : unit
?? ⊢ (λx.x) : ??
?? ⊢ ((λx.(x x)) (λx.(x x))) : ??

Typing Puzzles
How can we type the following terms?

Γ[x : τ] ⊢ x : τ
{}
?? ⊢ (λx.x) : ??
?? ⊢ ((λx.(x x)) (λx.(x x))) : ??

Typing Puzzles
How can we type the following terms?

Γ[x : τ] ⊢ x : τ
{}
?? ⊢ (λx.x) : unit → unit
?? ⊢ ((λx.(x x)) (λx.(x x))) : ??
Typing Puzzles
How can we type the following terms?

\[ \Gamma \vdash x : \tau \]

\[ \{ \} \vdash (\lambda x. x) : (\text{unit} \to \text{unit}) \to (\text{unit} \to \text{unit}) \]

\[ ?? \vdash ((\lambda x. (x \cdot x)) \cdot (\lambda x. (x \cdot x))) : ?? \]

Strong Normalization Theorem
- Typeable term always reduces to a value!
- Infinitely looping term does not reduce to a value so it is NOT typeable
- Untyped calculus originally used as a logic
  - Discovered paradox with untyped versions
  - Types introduced to rule out logical paradoxes
- Distinction between a “class” and proper set added to set theory for the same reasons

Simply Typed \( \lambda \)-calculus with Refs

- Names \( x \in \ldots \)
- Locations \( l \in \ldots \)
- Types \( \tau ::= \text{unit} \mid \tau_1 \to \tau_2 \mid (\tau) \text{ ref} \)
- Expr’s \( e ::= () \mid (\lambda x. e) \mid (e_1 \cdot e_2) \mid x \mid \text{ref}(e) \mid le \mid e_1 \Rightarrow e_2 \mid l \)
- Values \( v ::= () \mid (\lambda x. e) \mid l \)
- Memories \( M ::= \{(r \mapsto v_1, \ldots, r \mapsto v_n)\} \)
- Programs \( P ::= (M, e) \)
- Answers \( A ::= (M, v) \)
- Should be able to fill in the details of typing rules and semantics yourself

Backpatching and References
- Our “paradoxical” term
  \(((\lambda x. (x \cdot x)) \cdot (\lambda x. (x \cdot x)))\)
  still is NOT typeable
- However this term is
  \f \equiv \text{ref}(\lambda x. ()) \in
  \begin{align*}
  f &::= (\lambda x. (f ())) \cdot (f ())
  \end{align*}

where
\[ e_1 ; e_2 \equiv \text{let } x \text{ be } e_1 \text{ in } e_2 \]
and
\[ \text{let } x \text{ be } e_1 \text{ in } e_2 \equiv ((\lambda x. e_2) \cdot e_1) \]
let $f$ be $\text{ref}(\lambda x.())$ in
let $t := (\lambda x.() f())$ in

$(!f())$

where
$e_1; e_2 \equiv \text{let } x \text{ be } e_1 \text{ in } e_2 \quad x \notin \text{FN}(e_2)$
and
$\text{let } x \text{ be } e_1 \text{ in } e_2 \equiv ((\lambda x.e_2) e_1)$