Computational Effects

In contrast, C (or C++ or Java) is an imperative language.

- Expressions have both a value and an effect on variables.
- Bindings may change by assignment.

Effects in ML

The ML family of languages is impure. They have

- **Store effects**: allocation and mutation of storage.
- **Control effects**: non-local transfers of control.

We’ve previously considered control effects, namely exceptions and continuations.

In this lecture we’ll study store effects.

Store Effects

Extend MinML expressions with this abstract syntax:

- **Types**
  
  \[ \begin{align*}
  \tau & ::= \text{ref}(\tau) \\
  \end{align*} \]

- **Exprs**
  
  \[ \begin{align*}
  e & ::= l | \text{new}(v) | \text{get}(e) | \text{set}(e_1, e_2) \\
  \end{align*} \]

- **Values**
  
  \[ \begin{align*}
  v & ::= l \\
  \end{align*} \]

Locations are values!

Static Semantics for References

A **location typing** is a finite function \( \Lambda : \text{Loc} \rightarrow \text{Type} \).

A **typing assertion** is a quadruple \( \Lambda; \Gamma \vdash e : \tau \).

- \( \Lambda \) is a location typing;
- \( \Gamma \) is a variable typing;
- \( e \) is an expression with locations from \( \text{dom}(\Lambda) \) and free variables from \( \text{dom}(\Gamma) \);
- \( \tau \) is a type expression.
Type Safety for References

Two judgements:

- $\text{(M, e) ok = machine state (M, e) is well-formed.}$
- $\text{M : } \Lambda = \text{memory M has location typing } \Lambda.$

Well-formedness of machine states:

$$\text{M : } \Lambda \quad \Lambda \vdash e : \tau \quad \frac{\text{(M, e) ok}}{\text{M, e ok}}.$$

Dynamic Semantics for References

Memory locations act like bound variables of the machine state!

- The “names” of locations don’t matter, and can be changed at will.
- In a state $(M, e)$ the locations in $\text{dom}(M)$ are bound simultaneously in $M$ and in $e$.

We regard as equivalent any two states that differ only in the names of locations.

Typing Rules for References

$$\frac{}{\Lambda, \Gamma \vdash \ell : \text{ref}(\tau)}$$

$$\frac{}{\Lambda, \Gamma \vdash \text{new}(e) : \text{ref}(\tau)}$$

$$\frac{}{\Lambda, \Gamma \vdash \text{get}(e) : \tau}$$

$$\frac{}{\Lambda, \Gamma \vdash e_b : \text{ref}(\tau_c) \quad \Lambda, \Gamma \vdash e_c : \tau_c \quad \Lambda, \Gamma \vdash \text{set}(e_b, e_c) : \tau_c}$$

Dynamic Semantics for References

Abstract machine with states $(M, e)$, where

- $(M, e)$ is a memory with domain $\text{dom}(M)$.
- $e$ is an expression with locations in $\text{dom}(M)$.

A memory is a finite function $M : \text{Loc} \to \text{Val}$ such that

- $M(l)$ has no free variables;
- the locations in $M(l)$ are in $\text{dom}(M)$.

Memory Typing

The definition of memory typing is crucial:

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

Important: typing for $M(l)$ uses all of $\Lambda$!

- Allows self-reference: $M(l) = \text{fn } x \mapsto (\text{!l})(x)$.
- Allows cyclic references: $M(l) = \text{fn } x \mapsto (\text{!l'})(x)$ and $M(l') = \text{fn } x \mapsto (\text{!l})(x)$. 
Proof of Preservation
Suppose that $M(l) = \text{fn } x \Rightarrow ((l) x)$ and that $\Lambda(l) = \text{int} \rightarrow \text{int}$. To check that $M : \Lambda$, we must check
$$\Lambda \vdash \text{fn } x \Rightarrow ((l) x) : \text{int} \rightarrow \text{int}$$
This is easily verified, since we are assuming that $l$ has type $\text{ref} \!(\text{int} \rightarrow \text{int})$.

Preservation Theorem
Theorem 1 (Preservation)
If $M, e$ ok and $M, e \rightarrow M', e'$, then $(M', e')$ ok.

Doesn’t state relation between pre- and post-typing of memory!

• Memory grows monotonically.

• Type never changes once allocated.

Proof of Preservation
Suppose that $e = \text{new}(v)$ so that $M' = M[l = v]$, where $l \notin \text{dom}(M)$, and $e' = l$.

Suppose that $M : \Lambda$ and $\Lambda ; \vdash e : \text{ref}(\tau)$.

Choose $\Lambda' := \Lambda[l : \tau]$. Note that $l \notin \text{dom}(\Lambda)$ and that $\Lambda' \supseteq \Lambda$.

Finally, note that $\vdash M' : \Lambda'$ and $\Lambda' ; \vdash e' : \text{ref}(\tau)$.

Backpatching
(* loop forever when called *)
fun diverge (x:int):int = diverge x
(* allocate a reference cell *)
val fc : (int->int) ref = ref (diverge)
(* define a function that ’’recurse’’ through fc *)
fun f 0 = 1 | f n = n * ((!fc)(n-1))
(* tie the knot *)
val _ = fc := f
(* now call f *)
val n : int = f 5

Preservation for References
We prove instead this stronger form:
Lemma 2
If $M, e \rightarrow (M', e'), \vdash M : \Lambda$, and $\Lambda ; \vdash e : \tau$, then there exists $\Lambda' \supseteq \Lambda$ such that $\vdash M' : \Lambda'$ and $\Lambda' ; \vdash e' : \tau$.

Proof is by induction on evaluation.

Progress for References
Theorem 3 (Progress)
If $M, e$ ok then either $e$ is a value or there exists $(M', e')$ such that $(M, e) \rightarrow (M', e')$.

The proof is by induction on typing, using an extended canonical forms lemma.

Lemma 4
Suppose that $\Lambda ; \vdash v : \text{ref}(\tau)$. Then $v = l \in \text{dom}(\Lambda)$ and $\Lambda(l) = \tau$. 
A Puzzle

This used to be a type-correct ML program:

```ml
fun diverge x = diverge x
val r : ('a -> 'b) ref = ref (diverge)  (* store something into it *)
val _ = r := (fn () => true)  (* retrieve at another type *)
val n : int = (!r)()+1
Kaboom!
```

Monadic Store Effects

Extend MinML expressions with this abstract syntax:

| Types  | \( \tau \) ::=
|--------|-----------------------
| Pure   | \( e \) ::= \( l \mid \text{comp}(m) \)
| Impure | \( m \) ::= \text{return}(e) \mid \text{letcomp}(e,x.m) \mid \text{new}(c) \mid \text{get}(c) \mid \text{set}(c_1,c_2) \)
| Values | \( v \) ::= \( l \mid \text{comp}(m) \)

A Puzzle

This program clearly goes wrong . . . yet it seems to be well-typed!

- Assuming \( \text{ref} \) has type \('a -> 'a \text{ref}\), \( \text{!} \) has type \('a \text{ref} -> 'a\), and \( := \) has type \('a \text{ref} * 'a -> \text{unit}\).
- No other restrictions on typing.

Static Semantics for Monads

Two forms of typing judgements:

- \( e : \tau \) = meaning pure expression \( e \) has type \( \tau \);
- \( m \sim \tau \) = meaning impure expression \( m \) has type \( \tau \).

References

There have been various answers to both questions. It is a remarkably subtle problem.
Dynamic Semantics for Monads

Dynamic semantics for monads structured in two parts:

- $e \rightarrow e'$ for pure expressions.
- $(M, m) \rightarrow (M', m')$ for impure expressions.

Both are defined as they were for the $M$ machine and the abstract machine for references, respectively, extended with the following rules for monadic constructs:

Typing Rules for Monads

\[
\begin{align*}
\Gamma, \Gamma \vdash m \sim \tau & \quad & \Gamma, \Gamma \vdash \text{comp}(m) \mathbin{\triangleleft} \text{comp}(\tau) \\
\Gamma, \Gamma \vdash e : \tau & \quad & \Gamma, \Gamma \vdash \text{return}(e) \sim \tau \\
\Gamma, \Gamma \vdash e : \text{comp}(\tau) & \quad & \Gamma, \Gamma, x : \tau \vdash m \sim \tau' \\
\Gamma, \Gamma \vdash \text{letcomp}(e, x.m) \sim \tau' \\
\end{align*}
\]

Dynamic Semantics for Monads

\[
\begin{align*}
(M, \text{return}(e)) & \rightarrow (M, \text{return}(e')) \\
(M, \text{letcomp}(e, x.m)) & \rightarrow (M, \text{letcomp}(e', x.m)) \\
(M, m_1) & \rightarrow (M', m'_1) \\
(M, \text{letcomp}(m_1, x.m_2)) & \rightarrow (M, \text{letcomp}(m'_1, x.m_2)) \\
(M, \text{letcomp}(\text{return}(v)), x.m) & \rightarrow (M, [x ← v].m) \\
\end{align*}
\]