510: Programming Languages

State

David Walker

Computational Effects

MinML is a **purely functional**, or **pure**, language.

- Expressions are evaluated for their value only.

- There is no mutation, I/O, exceptions, ...

Such phenomena are called **computational effects**. Languages with effects are said to be **impure**.

Computational Effects

We will study two forms of computational effects:

1. **Store effects**. Mutation of storage via reference cells.

2. **Control effects**. Non-local transfers of control via exceptions.

Store Effects

Extend MinML expressions with these ML-like primitives:

$$e \ ::= \ l \mid \texttt{ref}(e) \mid \texttt{!}e \mid e_1\texttt{:=}e_2$$

NB: locations l are expressions. Locations are values!

Also extend MinML types with **reference types** $\tau\,\texttt{ref}$.

Static Semantics for References

A **location typing** is a finite function $\Lambda : \mathrm{Loc} \rightharpoonup \mathrm{Type}$.

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

- Λ is a location typing;
- Γ is a variable typing;
- e is an expression with locations from $\mathrm{dom}(\Lambda)$ and free variables from $\mathrm{dom}(\Gamma)$.
- τ is a type expression.

Typing Rules for References

$$\frac{(\Lambda(l) = \tau)}{\Lambda; \Gamma \vdash l : \tau\, \texttt{ref}}$$

$$\frac{\Lambda; \Gamma \vdash e : \tau}{\Lambda; \Gamma \vdash \texttt{ref}(e) : \tau\, \texttt{ref}}$$

$$\frac{\Lambda; \Gamma \vdash e : \tau\, \texttt{ref}}{\Lambda; \Gamma \vdash \texttt{!}e : \tau}$$

$$\frac{\Lambda; \Gamma \vdash e_1 : \tau_2\, \texttt{ref} \qquad \Lambda; \Gamma \vdash e_2 : \tau_2}{\Lambda; \Gamma \vdash e_1 \texttt{:=} e_2 : \tau_2}$$

Dynamic Semantics for References

Abstract machine with states (M, e), where

- M is a memory with domain dom(M).
- e is an expression with locations in dom(M).

A **memory** is a finite function M : Loc $\rightharpoonup$ Val such that

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

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 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.

Dynamic Semantics for References

$$\frac{(M, e) \mapsto (M', e')}{(M, \texttt{ref}(e)) \mapsto (M', \texttt{ref}(e'))}$$

$$\frac{(l \notin \mathsf{dom}(M))}{(M, \texttt{ref}(v)) \mapsto (M[l{=}v], l)}$$

$$\frac{(M, e) \mapsto (M', e')}{(M, \texttt{!}e) \mapsto (M', \texttt{!}e')}$$

$$\frac{(l \in \mathsf{dom}(M))}{(M, \texttt{!}l) \mapsto (M, M(l))}$$

$$\frac{(M, e_1) \mapsto (M', e_1')}{(M, e_1 \texttt{:=} e_2) \mapsto (M', e_1' \texttt{:=} e_2)}$$

$$\frac{(M, e_2) \mapsto (M', e_2')}{(M, v_1 \texttt{:=} e_2) \mapsto (M', v_1 \texttt{:=} e_2')}$$

$$\frac{(l \in \mathsf{dom}(M))}{(M, l \texttt{:=} v) \mapsto (M[l{=}v], v)}$$

Type Safety for References

Two judgements:

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

Well-formedness of machine states:

$$\frac{\vdash M : \Lambda \qquad \Lambda; \bullet \vdash e : \tau}{(M, e)\ \mathsf{ok}}$$

Memory Typing

The definition of memory typing is **crucial**:

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

Important: typing for $M(l)$ uses **all** of Λ!

- Allows self-reference: $M(l) =$ `fn x => (!`l`)(x)`.

- Allows cyclic references:

  $M(l) =$ `fn x => (!`l'`)(x)`

  and

  $M(l') =$ `fn x => (!`l`)(x)`

Memory Typing

Suppose that $M(l) = \texttt{fn x => (!}l\texttt{)(x)}$ and that $\Lambda(l) = \texttt{int}{\to}\texttt{int}$.

To check that $\vdash M : \Lambda$, we must check

$$\Lambda; \bullet \vdash \texttt{fn x => (!}l\texttt{)(x)} : \texttt{int}{\to}\texttt{int}$$

This is easily verified, since we are **assuming** that l has type $\texttt{int}{\to}\texttt{int}\,\texttt{ref}$.

Backpatching

```
(* loop forever when called *)
fun diverge (x:int):int = diverge x
(* allocate a reference cell *)
val fc : (int->int) ref = ref (diverge)
(* a function that ``recurs'' 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 Theorem

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

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

- Memory grows monotonically.
- Type never changes once allocated.

Preservation for References

We prove instead this stronger form:

Lemma 2

If $(M, e) \mapsto (M', e')$, $\vdash M : \Lambda$, *and* $\Lambda; \bullet \vdash e : \tau$, **then** *there exists* $\Lambda' \supseteq \Lambda$ *such that* $\vdash M' : \Lambda'$ *and* $\Lambda'; \bullet \vdash e' : \tau$.

Proof is by induction on evaluation.

Proof of Preservation

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

Suppose that $\vdash M : \Lambda$ and $\Lambda; \emptyset \vdash e : \tau\,\texttt{ref}$.

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

Finally, note that $\vdash M' : \Lambda'$ and $\Lambda'; \emptyset \vdash e' : \tau\,\texttt{ref}$.

Progress for References

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

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

Lemma 4
Suppose that $\Lambda; \emptyset \vdash v : \tau' \, \mathbf{\texttt{ref}}$. *Then* $v = l \in$ dom(Λ) *and* $\Lambda(l) = \tau$.

A Puzzle

This used to be a type-correct ML program:

```
(* loops when applied *)
fun diverge x = diverge x
(* allocate a cell *)
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!

A Puzzle

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

- Assuming `ref` has type `'a -> 'a ref`, `!` has type `'a ref -> 'a`, and `:=` has type `'a ref * 'a -> unit`.

- No other restrictions on typing.

A Puzzle

Questions:

- What went wrong? Which typing rule is incorrect?

- How would you fix the problem?

There have been various answers to both questions. It is a remarkably subtle problem.