510: Programming Languages

Computational Effects

David Walker

Spring, 2001

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'll 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 $\Lambda : \mathsf{Loc} \rightharpoonup \mathsf{Type}$.

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:

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

Memory Typing

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

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

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

- Allows self-reference: $M(l) = \texttt{fn x => (!}l\texttt{)(x)}$.

- Allows cyclic references:

  $$M(l) = \texttt{fn x => (!}l'\texttt{)(x)}$$

  and

  $$M(l') = \texttt{fn x => (!}l\texttt{)(x)}$$

Memory Typing

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

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

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

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

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 $\Lambda; \emptyset \vdash v : \tau' \, \mathtt{ref}$. *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.

Exceptions

Add a primitive **failure** mechanism:

$$e \; ::= \; \ldots \mid \texttt{fail} \mid \texttt{try}\, e_1 \,\texttt{ow}\, e_2$$

Think of `fail` as raising a fixed, anonymous exception.

Think of $\texttt{try}\, e_1 \,\texttt{ow}\, e_2$ as e_1 `handle _ =>` e_2.

Static Semantics of Exceptions

No surprises here:

$$\frac{}{\Gamma \vdash \texttt{fail} : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{try}\, e_1 \,\texttt{ow}\, e_2 : \tau}$$

Failures have **any** type.

Normal and failure return for handler must have the **same** type.

Dynamic Semantics of Exceptions

Use the C Machine since it provides access to the control stack.

- Could use the M Machine (exercise).
- Another solution below relies on explicit control stack.

Additional frame:

$$\frac{e_2\ \mathsf{expr}}{\mathtt{try}\,\Box\,\mathtt{ow}\,e_2\ \mathsf{frame}}$$

Dynamic Semantics of Exceptions

Evaluate "try" clause:

$$\overline{(K, \texttt{try}\, e_1\, \texttt{ow}\, e_2) \mapsto (\texttt{try}\, \Box\, \texttt{ow}\, e_2 \triangleright K, e_1)}$$

If it achieves a value, return it and drop handler:

$$\overline{(\texttt{try}\, \Box\, \texttt{ow}\, e_2 \triangleright K, v) \mapsto (K, v)}$$

Dynamic Semantics of Exceptions

If "try" clause fails, unwind stack to nearest enclosing handler.

$$\frac{(F \neq \texttt{try}\,\Box\,\texttt{ow}\,e_2)}{(F \triangleright K, \texttt{fail}) \mapsto (K, \texttt{fail})}$$

Then invoke pending handler.

$$\frac{}{(\texttt{try}\,\Box\,\texttt{ow}\,e_2 \triangleright K, \texttt{fail}) \mapsto (K, e_2)}$$

Safety for Exceptions

Well-formed machine states:

$$\frac{\vdash K : \tau\, \texttt{stack} \qquad \vdash e : \tau}{(K, e)\ \mathsf{ok}}$$

That is, it must be type correct to pass (the value of) e to K.

Stack Typing

A stack is a composition of frames:

$$\frac{}{\vdash \bullet : \tau\,\texttt{stack}}$$

$$\frac{\vdash F : (\tau,\tau')\ \texttt{frame} \qquad \vdash K : \tau'\,\texttt{stack}}{\vdash F \triangleright K : \tau\,\texttt{stack}}$$

Frame Typing

$$\frac{\vdash e_2 : \mathtt{int}}{\vdash +(\Box, e_2) : (\mathtt{int},\mathtt{int})\ \mathtt{frame}}$$

$$\frac{v_1 \ \mathsf{value} \qquad \vdash v_1 : \mathtt{int}}{\vdash +(v_1, \Box) : (\mathtt{int},\mathtt{int})\ \mathtt{frame}}$$

Frame Typing

$$\frac{\vdash e_2 : \tau_2}{\vdash \texttt{apply}(\Box, e_2) : (\tau_2 \rightarrow \tau \texttt{,} \tau)\ \texttt{frame}}$$

$$\frac{v_1\ \textsf{value} \qquad \vdash v_1 : \tau_2 \rightarrow \tau}{\vdash \texttt{apply}(v_1, \Box) : (\tau_2 \texttt{,} \tau)\ \texttt{frame}}$$

Frame Typing

$$\frac{\vdash e_1 : \tau \qquad \vdash e_2 : \tau}{\vdash \mathtt{if}_\tau \, \square \, \mathtt{then} \, e_1 \, \mathtt{else} \, e_2 \, \mathtt{fi} : (\mathtt{bool}, \tau) \ \mathtt{frame}}$$

$$\frac{\vdash e_2 : \tau}{\vdash \mathtt{try} \, \square \, \mathtt{ow} \, e_2 : (\tau, \tau') \ \mathtt{frame}}$$

Type Safety

Theorem 5 (Preservation)
If (K, e) ok *and* $(K, e) \mapsto (K', e')$, *then* (K, e) ok.

Theorem 6 (Progress)
If (K, e) ok *then either*

1. $K = \bullet$ *and* e value, *or*

2. $K = \bullet$ *and* $e = \texttt{fail}$, *or*

3. *there exists* K' *and* e' *such that* $(K, e) \mapsto (K', e')$.

A More Realistic Semantics

Raising an exception walks the stack to find nearest enclosing handler.

- Slow: linear in depth of stack, not depth of handler nesting.
- Penalizes code that doesn't raise exceptions by poppin unused exception handlers.
- Requires run-time information to detect handler.

We expect a failure to jump **directly** to the handler and non-failing code to execute normally.

A More Realistic Semantics

Informal idea:

- Maintain a register containing **current handler**.
- On failure, **jump** directly to contents of this register.
- Restore **previous** handler before jump and before normal return.
- Restore control stack before invoking handler.

A More Realistic Semantics

This semantics requires:

- A **handler stack** to enclose pending handlers. Current handler is top of stack. Stack is modified during execution.

- A means of **freeze-drying** and **thawing** stacks to save stack appropriate to handler.

A More Realistic Semantics

Machine state: (H, K, e), where

- H is a handler stack;
- K is a control stack;
- e is a closed expression

A More Realistic Semantics

Handler stacks:

$$\frac{}{\bullet \ \mathsf{hstack}}$$

$$\frac{K \ \mathsf{stack} \quad e \ \mathsf{expr} \quad H \ \mathsf{hstack}}{(K, e) \triangleright H \ \mathsf{hstack}}$$

Store exception handler paired with control stack.

A More Realistic Semantics

Handlers preserve state on handler stack:

$$\overline{(H, K, \texttt{try}\, e_1\, \texttt{ow}\, e_2) \mapsto ((K, e_2) \triangleright H, \texttt{try}\,\square\,\texttt{ow}\, e_2 \triangleright K, e_1)}$$

On failure, pop the handler stack:

$$\overline{((K', e') \triangleright H, K, \texttt{fail}) \mapsto (H, K', e')}$$

If no handler, stop execution immediately:

$$\overline{(\bullet, K, \texttt{fail}) \mapsto (\bullet, \bullet, \texttt{fail})}$$

A More Realistic Semantics

If guarded expression evaluates normally, we must remember to pop the handler stack:

$$\frac{}{((K, e_2) \triangleright H, \texttt{try} \,\square\, \texttt{ow}\, e_2 \triangleright K, v_1) \mapsto (H, K, v_1)}$$

A More Realistic Semantics

Is freeze-drying a stack realistic?

- Sounds like a heavyweight operation.

- Can we "hot swap" stacks on a running program?

A More Realistic Semantics

Lemma 7
Let (H, K, e) be a state of the abstract machine. If $H = (K', e') \triangleright H'$, then K' is an initial segment of K. Moreover, the same property holds of (H', K, e).

Thus stored stacks may be represented as "fingers" pointing into the control stack.

- Restore by multi-pop to a given point earlier in the stack.

- *cf.* `setjmp/longjmp` in C.

Value-Passing Exceptions

It's important to be able to distinguish different sorts of failures.

- Division-by-zero, arithmetic overflow.
- Match and bind failures.
- User-defined failures.

Solution: pass values along with exceptions.

Value-Passing Exceptions

A schematic formulation:

$$\frac{\Gamma \vdash e : \tau_{\mathsf{exn}}}{\Gamma \vdash \mathtt{raise}(e) : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau_{\mathsf{exn}} \rightarrow \tau}{\Gamma \vdash \mathtt{try}\, e_1\, \mathtt{ow}\, e_2 : \tau}$$

Question: how to choose τ_{exn}?

Value-Passing Exceptions

Observation: there must be **one** choice governing **all** exceptions.

- Handler cannot tell which exception will be raised.
- Handler usually analyzes value associated with the exception.

Value-Passing Exceptions

A naïve choice: $\tau_{\text{exn}} = \texttt{string}$.

```
fun div (m, 0) = raise "Division by zero"
  | div (m, n) = ... raise "Overflow" ...
```

But how can the handler distinguish exceptions?

- Must parse the string.

- Must rely on formatting conventions.

Unworkable in practice!

Value-Passing Exceptions

A more reasonable choice: $\tau_{\mathsf{exn}} = \mathtt{exc}$.

```
Datatype exc = Div | Overflow | Match | ···
```

Then we can easily distinguish exceptions using pattern matching:

```
fun div (m, 0) = raise Div
  | div (m, n) = ... raise Overflow ...
fun hdlr Div = ...
  | hdlr Overflow = ...
```

Value-Passing Exceptions

Requires that we **fix in advance** the set of exceptions.

- Non-modular. Makes writing libraries difficult.

- Non-extensible. No user-defined exceptions.

Value-Passing Exceptions

Better: a **dynamically extensible** datatype.

- Called `exn` in ML.
- Add a **new** constructor using `exception` declaration.
- Pattern match to dispatch on exception.

```
exception Div
exception Overflow
fun div (m, 0) = raise Div
  | div (m, n) = ... raise Overflow ...
```

Exn's Are Not Exceptions!

The `exn` type has nothing to do with the exception mechanism!

- Useful on its own as an extensible type.
- So-called exceptions are really value constructs introduced on the fly.
- Exception mechanism passes values of type `exn`.

Exn's Are Not Exceptions!

What's important about `exn`:

- Dynamically extensible.
- Supports tagging and dispatch.

The Java type `Object` is very similar to ML's `exn`.

Summary

Effects can be integrated into MinML with minimal pain.

Abstract machines play a stronger role in the presence of effects.

Reference typing is subtle because of cycles.

The exception type and the exception mechanism are conceptually distinct.