Computer Science 15-312
Computational Effects

Robert Harper

Spring, 2000

1 Introduction

MinML is a purely functional language, in two senses. First, the only control mechanism is procedure call and return. Procedures may call themselves recursively, which is the sole means of looping in the language. Second, evaluation is purely a matter of calculation. A program is an expression whose value we wish to calculate. Expressions "stand for" their values; the primitive operations are defined in terms of values.

This simple model of evaluation must be generalized to account for *computational effects* (or simply *effects*) such as mutation of storage or non-local transfer of control. The purpose of this note is to study the extension of MinML with effects. First, we will add *mutable reference cells* to MinML in the style of Standard ML. With this extension programs not only have values, as before, but also engender *store effects*, modifications to mutable cells. Second, we will add *exceptions* to MinML that allow non-local transfers of control. Programs may now also exhibit *control effects*, or changes to the execution context effected by branching to an exception handler.

2 References

The MinML type language is extended with *reference types* $\tau\,\texttt{ref}$ whose elements are to be thought of as mutable storage cells. We correspondingly extend the expression language with these primitive operations:

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

As in Standard ML, $\texttt{ref(}e\texttt{)}$ allocates a "new" reference cell, $\texttt{!}e$ retrieves the contents of the cell e, and $e_1\texttt{:=}e_2$ sets the contents of the cell e_1 to the value

e_2. The variable l ranges over a set of *locations*, an infinite set of identifiers disjoint from variables. These are needed for the dynamic semantics, but are not expected to be notated directly by the programmer. The set of *values* is extended to include locations.

Typing judgments have the form $\Lambda;\Gamma \vdash e : \tau$, where Λ is a *location typing*, a finite function mapping locations to types; the other components of the judgement are as for MinML. The location typing Λ records the types of allocated locations during execution; this is critical for a precise statement and proof of type soundness.

The typing rules are those of MinML (extended to carry a location typing), plus the following rules governing the new constructs of the language:

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

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

Notice that the location typing is not extended during type checking! Locations arise only during execution, and are not part of complete programs, which must not have any free locations in them. The role of the location typing will become apparent in the proof of type safety for MinML extended with references.

A *memory* is a finite function mapping locations to closed values (but possibly involving locations). The dynamic semantics of MinML with references is given by an abstract machine. The states of this machine have the form (M, e), where M is a memory and e is an expression possibly involving free locations in the domain of M. The locations in $\mathrm{dom}(M)$ are bound simultaneously in (M, e); the names of locations may be changed at will without changing the identity of the state.

The transitions for this machine are similar to those of the M machine, but with these additional steps:

$$\frac{(M, e) \mapsto (M', e')}{(M, \texttt{ref}(e)) \mapsto (M, \texttt{ref}(e'))} \qquad \frac{(l \notin \mathrm{dom}(M))}{(M, \texttt{ref}(v)) \mapsto (M[l{=}v], l)}$$

$$\frac{(M, e) \mapsto (M', e')}{(M, \texttt{!}e) \mapsto (M', \texttt{!}e')} \qquad \frac{(l \in \mathrm{dom}(M))}{(M, \texttt{!}l) \mapsto (M, M(l))}$$

$$\frac{(M, e_1) \mapsto (M', e_1')}{(M, e_1\texttt{:=}e_2) \mapsto (M', e_1'\texttt{:=}e_2)} \qquad \frac{(M, e_2) \mapsto (M', e_2')}{(M, v_1\texttt{:=}e_2) \mapsto (M', v_1\texttt{:=}e_2')}$$

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

A state (M, e) is *final* iff e is a value (possibly a location).

To prove type safety for this extension we will make use of some auxiliary relations. Most importantly, the typing relation between memories and location typings, written $\vdash M : \Lambda$, is inductively defined by the following rule:

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

It is very important to study this rule carefully! First, we require that Λ and M govern the same set of locations. Second, for each location l in their common domain, we require that the value at location l, namely $M(l)$, have the type assigned to l, namely $\Lambda(l)$, relative to the *entire* location typing Λ. This means, in particular, that memories may be "circular" in the sense that the value at location l may contain an occurrence of l, for example if that value is a function.

The typing rule for memories is reminiscent of the typing rule for recursive functions — we are allowed to assume the typing that we are trying to prove while trying to prove it. This similarity is no accident, as the following example shows. Here we use ML notation, but the example can be readily translated into MinML extended with references:

```
(* 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 ``recurs'' through fc *)
fun f 0 = 1 | f n = n * ((!fc)(n-1))
(* tie the knot *)
val _ = fc := f
(* now call f *)
val n = f 5
```

This technique is called *backpatching*. It is used in some compilers to implement recursive functions (and other forms of looping construct).

Exercise 1

1. *Sketch the contents of the memory after each step in the above example. Observe that after the assignment to* `fc` *the memory is "circular" in the sense that some location contains a reference to itself.*

2. *Prove that every cycle in well-formed memory must "pass through" a function. Suppose that $M(l_1) = l_2$, $M(l_2) = l_3$, $\ldots$, $M(l_n) = l_1$ for some sequence $l_1, \ldots, l_n$ of locations. Show that there is no location typing Λ such that $\vdash M : \Lambda$.*

The well-formedness of a machine state is inductively defined by the following rule:

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

That is, (M, e) is well-formed iff there is a location typing for M relative to which e is well-typed.

Theorem 2 (Preservation)
If (M, e) ok and $(M, e) \mapsto (M', e')$, then (M', e') ok.

Proof: The trick is to prove a stronger result by induction on evaluation: 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 $\Lambda'; \bullet \vdash e' : \tau$. ■

Exercise 3
Prove Theorem 2. The strengthened form tells us that the location typing, and the memory, increase monotonically during evaluation — the type of a location never changes once it is established at the point of allocation. This is crucial for the induction.

Theorem 4 (Progress)
If (M, e) ok then either (M, e) is a final state or there exists (M', e') such that $(M, e) \mapsto (M', e')$.

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

Exercise 5
Prove Theorem 4 by induction on typing of machine states.

3 Exceptions

The purpose of an exception mechanism is to provide a means of effecting a non-local transfer of control to an exception handler. Exceptions are typically used to signal the occurrence of an unexpected or unusual event requiring special treatment. A significant advantage of an exception mechanism over explicit error checking is that the flow of control is not disrupted by the unusual case. In particular explicit error checks are easily forgotten or neglected, whereas exceptions cannot be simply ignored. For example, the typical case for opening a new file for writing might be to return a file descriptor, which only rarely fails due to file system overflow. By arranging the code for opening a new file to signal an exception in this case we can prevent accidental neglect of this condition without introducing unduly complex code.

We'll consider the extension of MinML with an exception mechanism similar to that of Standard ML, with the significant simplification that no value is associated with the exception — we simply signal the exception and thereby invoke the nearest dynamically enclosing handler. We'll come back to consider value-passing exceptions later.

The following grammar describes the extensions to MinML to support valueless exceptions:

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

The expression `fail` raises an exception. The expression $\texttt{try}\, e_1 \,\texttt{ow}\, e_2$ evaluates e_1. If it terminates normally, we return its value; otherwise, if it fails, we continue by evaluating e_2.

The static semantics of exceptions is quite straightforward:

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

Observe that a failure can have any type, precisely because it never returns. Both clauses of a handler must have the same type, to allow for either possible outcome of evaluation.

The dynamic semantics of exceptions is given in terms of the C machine with an explicit control stack. The set of frames is extended with the following additional clause:

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

The evaluation rules are extended as follows:

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

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

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

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

To evaluate $\texttt{try}\,e_1\,\texttt{ow}\,e_2$ we begin by evaluating e_1. If it achieves a value, we "pop" the pending handler and yield that value. If, however, it fails, we continue by evaluating the "otherwise" clause of the nearest enclosing handler. Notice that we explicitly "pop" non-handler frames while processing a failure; this is sometimes called *unwinding* the control stack. Finally, we regard the state $(\bullet, \texttt{fail})$ as a final state of computation, corresponding to an uncaught exception.

Exercise 6
Hand-simulate the evaluation of a few simple expressions with exceptions and handlers to get a feeling for how it works.

To prove safety we define well-formedness of machine states by the following rule:

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

That is, a state (K, e) is well-formed iff e is an expression of type τ and K is a τ-accepting control stack. The latter is defined by the following rules:

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

A stack is well-formed iff its frames compose properly. The type of a frame is defined by the following rules:

$$\frac{\vdash e_2 : \texttt{int}}{\vdash +(\Box, e_2) : (\texttt{int},\texttt{int})\ \texttt{frame}} \qquad \frac{v_1\ \mathsf{value} \quad \vdash v_1 : \texttt{int}}{\vdash +(v_1, \Box) : (\texttt{int},\texttt{int})\ \texttt{frame}}$$

$$\frac{\vdash e_1 : \tau \quad \vdash e_2 : \tau}{\vdash \texttt{if}\,\Box\,\texttt{then}\,e_1\,\texttt{else}\,e_2\,\texttt{fi} : (\texttt{bool},\tau)\ \texttt{frame}}$$

$$\frac{\vdash e_2 : \tau_2}{\vdash \texttt{apply}(\Box, e_2) : (\tau_2\to\tau,\tau)\ \texttt{frame}} \qquad \frac{v_1\ \mathsf{value} \quad \vdash v_1 : \tau_2\to\tau}{\vdash \texttt{apply}(v_1, \Box) : (\tau_2,\tau)\ \texttt{frame}}$$

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

Intuitively, a frame of type $(\tau_1,\tau_2)\ \texttt{frame}$ takes an "argument" of type τ_1 and yields a "result" of type τ_2. The argument is represented by the "$\Box$" in the frame; the result is the type of the frame once its hole has been filled with an expression of the given type.

With this in place we can state and prove safety.

Theorem 7 (Preservation)
If (K, e) ok *and* $(K, e) \mapsto (K', e')$*, then* (K, e) ok.

Proof: By induction on evaluation. ■

Exercise 8
Prove Theorem 7.

Theorem 9 (Progress)
If (K, e) ok *then either*

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

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

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

Proof: By induction on typing. ■

Exercise 10
Prove Theorem 9.

Exercise 11
Combine the treatment of references and exceptions to form a language with both of these features. You will face a choice of how to define the interaction between mutation and exceptions:

1. *As in ML, mutations are irrevocable, even in the face of exceptions that "backtrack" to a surrounding handler.*

2. *Invocation of a handler rolls back the memory to the state at the point of installation of the handler.*

Give a dynamic semantics for each alternative, and argue for and against each choice.

The dynamic semantics of exceptions is somewhat unsatisfactory because of the explicit unwinding of the control stack to find the nearest enclosing handler. While this does effect a non-local transfer of control, it does so by rather crude means, rather than by a direct "jump" to the handler. In practice exceptions are implemented as jumps, using the following ideas. A dedicated register is set aside to contain the "current" exception handler.

When an exception is raised, the current handler is retrieved from the exception register, and control is passed to it. Before doing so, however, we must reset the exception register to contain the nearest handler enclosing the new handler. This ensures that if the handler raises an exception the correct handler is invoked. How do we recover this handler? We maintain a stack of pending handlers that is pushed whenever a handler is installed, and popped whenever a handler is invoked. The exception register is the top element of this stack. Note that we must restore the control stack to the point at which the handler was installed before invoking the handler!

This can be modelled by a machine with states of the form (H, K, e), where

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

A handler stack consists of a stack of pairs consisting of a handler together its associated control stack:

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

A handler stack element consists of a "freeze dried" control stack paired with a pending handler.

The key transitions of the machine are given by the following rules. On failure we pop the control stack and pass to the exception stack:

$$\frac{}{((K', e') \triangleright H, K, \mathtt{fail}) \mapsto (H, K', e')}$$

We pop the handler stack, "thaw" the saved control stack, and invoke the saved handler expression. If there is no pending handler, we stop the machine:

$$\frac{}{(\bullet, K, \mathtt{fail}) \mapsto (\bullet, \bullet, \mathtt{fail})}$$

To install a handler we preserve the handler code and the current control stack:

$$\frac{}{(H, K, \mathtt{try}\, e_1\, \mathtt{ow}\, e_2) \mapsto ((K, e_2) \triangleright H, \mathtt{try}\, \Box\, \mathtt{ow}\, e_2 \triangleright K, e_1)}$$

We "freeze dry" the control stack, associate it with the unevaluated handler, and push it on the handler stack. We also push a frame on the control stack to remind us to remove the pending handler from the handler stack in the case of normal completion of evaluation of e_1:

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

Exercise 12
State and prove the safety of this formulation of exceptions.

The idea of "freeze-drying" an entire control stack and "thawing" it later may seem like an unusually heavy-weight operation. However, a key invariant governing a machine state (H, K, e) is the following *prefix property*: if $H = (K', e') \triangleright H'$, then K' is a prefix of K. This means that we can store a control stack by simply keeping a "finger" on some initial segment of it, and can restore a saved control stack by popping up to that finger.

Exercise 13
Prove that the prefix property is preserved by every step of evaluation.

Finally, let us consider value-passing exceptions such as are found in Standard ML. The main idea is to replace the failure expression, `fail`, by a more general *raise* expression, `raise(`e`)`, which associates a value (that of e) with the failure. Handlers are generalized so that the "otherwise" clause is a function accepting the value associated with the failure, and yielding a value of the same type as the "try" clause. Here is a sketch of the static semantics for this variation:

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

These rules are parameterized by the type of values associated with exceptions, τ_{exn}.

The question is: what should be the type τ_{exn}? The first thing to observe is that *all* exceptions should be of the same type, otherwise we cannot guarantee type safety. The reason is that a handler might be invoked by *any* raise expression occurring during the execution of its "try" clause. If one exception raised an integer, and another a boolean, the handler could not safely dispatch on the exception value. Given this, we must choose a type τ_{exn} that supports a flexible programming style.

For example, we might choose, say, `string`, for τ_{exn}, with the idea that the value associated with an exception is a description of the cause of the exception. For example, we might write

```
fun div (m, 0) = raise "Division by zero attempted."
  | div (m, n) = ... raise "Arithmetic overflow occurred." ...
```

However, consider the plight of the poor handler, which may wish to distinguish between division-by-zero and arithmetic overflow. How might it do that? If exception values were strings, it would have to parse the string, relying on the message to be in a standard format, and dispatch based on the parse. This is manifestly unworkable. For similar reasons we wouldn't choose τ_{exn} to be, say, `int`, since that would require coding up exceptions as numbers, much like "error numbers" in Unix. Again, completely unworkable in practice, and completely unmodular (different modules are bound to conflict over their numbering scheme).

A more reasonable choice would be to define τ_{exn} to be a given datatype `exc`. For example, we might have the declaration

```
datatype exc = Div | Overflow | Match | Bind
```

as part of the implicit prelude of every program. Then we'd write

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

Now the handler can easily dispatch on `Div` or `Overflow` using pattern matching, which is much better. However, this choice restricts all programs to a *fixed* set of exceptions, the value constructors associated with the predeclared `exc` datatype.

To allow extensibility Standard ML includes a special *extensible datatype* called `exn`. Values of type `exn` are similar to values of a datatype, namely they are constructed from other values using a constructor. Moreover, we may pattern match against values of type `exn` in the usual way. But, in addition, we may introduce *new* constructors of type `exn` "on the fly", rather than declare a fixed set at the beginning of the program. Such new constructors are introduced using an exception declaration such as the following:

```
exception Div
exception Overflow
```

Now `Div` and `Overflow` are constructors of type `exn`, and may be used in a `raise` expression or matched against by an exception handler. Exception declarations can occur anywhere in the program, and are guaranteed (by α-conversion) to be distinct from all other exceptions that may occur elsewhere in the program, even if they happen to have the same name. If two modules declare an exception named `Error`, then these are *different* exceptions; no confusion is possible.

The interesting thing about the `exn` type is that *it has nothing whatsoever to do with the exception mechanism* (beyond the fact that it is the type of values associated with exceptions). In particular, the `exception` declaration introduces a value constructor that has no inherent connection with the exception mechanism. We may use the `exn` type for other purposes; indeed, Java has an analogue of the type `exn`, called `Object`. This is the basis for downcasting and so-called typecase in Java.