Supplementary Notes on Exceptions

15-312: Foundations of Programming Languages
Frank Pfenning

Lecture 9
September 25, 2002

In this lecture we first give an implementation of the C-machine for the fragment containing integers, booleans, and functions using higher-order functions. We then discuss exceptions as an extension of the C-machine [Ch. 13]

The implementation of the C-machine is to represent the stack as a *continuation* that encapsulates the rest of the computation to be performed.[1]

First, in our implementation, both expressions and value have type `exp`. We nonetheless use different names to track our intuition, even though the type system of ML does not help use verify the correctness of this intuition.

```
type value = exp
e : exp
v : value
```

Next, the stack k is represented by an ML function

```
k : value -> value
```

Applying this function to a value v will carry out the rest of the computation of the machine, returning the final answer. Finally, we have two functions

[1]We give some code excerpts here; the full code can be found at `http://www.cs.cmu.edu/~fp/courses/312/code/09-exceptions/`.

```
eval : exp -> (value -> value) -> value
return : value -> (value -> value) -> value
```

satisfying the specification:

(i) `eval` $e\ \ k \Downarrow a$ iff $k > e \mapsto_{\mathsf{c}}^{*} \bullet < a$

(ii) `return` $v\ \ k \Downarrow a$ iff $k < v \mapsto_{\mathsf{c}}^{*} \bullet < a$

In order to implement stacks as ML functions, it is useful to introduce some new auxiliary functions to represent the frames. We give in the table below the association between forms of the stack and the corresponding ML function. We omit only the case for primops which requires a simple treatment of lists.

$k \triangleright \mathtt{if}(\Box, e_2, e_3)$	`(fn v1 => ifFrame (v1, e2, e3) k)`
$k \triangleright \mathtt{apply}(\Box, e_2)$	`(fn v1 => applyFrame1 (v1, e2) k)`
$k \triangleright \mathtt{apply}(v_1, \Box)$	`(fn v2 => applyFrame2 (v1, v2) k)`
$k \triangleright \mathtt{let}(\Box, x.e_2)$	`(fn v1 => letFrame (v1, ((), e2)) k)`
$\bullet$	`(fn v => v)`

The case of the empty stack corresponds to the initial continuation, which simply returns the value passed to it as the result of the overal computation

Now we can piece together the whole code elegantly, as advertised. We have elided only the case for primitive operations, which can be found with the complete code at the address given above.

```
fun eval (v as Int _) k = return v k
    (* elided primops *)
  | eval (v as Bool _) k = return v k
  | eval (If(e1, e2, e3)) k =
      eval e1 (fn v1 => ifFrame (v1, e2, e3) k)
  | eval (v as Fun _) k = return v k
  | eval (Apply(e1, e2)) k =
      eval e1 (fn v1 => applyFrame1 (v1, e2) k)
  | eval (Let(e1, ((), e2))) k =
      eval e1 (fn v1 => letFrame (v1, ((), e2)) k)
  (* eval (Var _) k impossible by MinML typing *)
and ifFrame (Bool(true), e2, e3) k = eval e2 k
  | ifFrame (Bool(false), e2, e3) k = eval e3 k
  (* other expressions impossible by MinML typing *)
and applyFrame1 (v1, e2) k =
       eval e2 (fn v2 => applyFrame2 (v1, v2) k)
and applyFrame2 (v1 as Fun(_, _, ((), (), e1')), v2) k =
       eval (Subst.subst (v1, 2, Subst.subst (v2, 1, e1'))) k
  (* other expressions impossible by MinML typing *)
and letFrame (v1, ((), e2)) k = eval (Subst.subst (v1, 1, e2)) k
and return v k = k v
```

The overall evaluation just starts with the initial continuation which corresponds to the empty stack.

```
fun evaluate e = eval e (fn v => v)
```

This style of writing an interpreter is also refered to as *continuation-passing style*. It is quite flexible and elegant, and will be exercised in Assignment 4.

Next we come to exceptions. We introduce a new form of state

$$k \ll \mathsf{fail}$$

which signals that we are propagation an exception upwards in the control stack k, looking for a handler or stopping at the empty stack. This "uncaught exception" is a particularly common form of implementing run-time errors. We do not distinguish different exceptions, only failure. For more complex variations of exceptions, see [Ch. 13] and Assignment 4.

We have two new forms of expressions `fail`(τ) (with concrete syntax

$\texttt{fail}[\tau])$[2] and $\texttt{try}(e_1, e_2)$ (with concrete syntax $\texttt{try}\, e_1\, \texttt{ow}\, e_2$). Informally, $\texttt{try}(e_1, e_2)$ evaluates e_1 and returns its value. If the evaluation of e_1 fails, that is, an exception is raised, then we evaluate e_2 instead and returns its value (or propagate *its* exception). These rules are formalized in the C-machine as follows.

$$\begin{array}{lll}
k > \texttt{try}(e_1, e_2) & \mapsto_{\mathsf{c}} & k \triangleright \texttt{try}(\Box, e_2) > e_1 \\
k \triangleright \texttt{try}(\Box, e_2) < v_1 & \mapsto_{\mathsf{c}} & k < v_1 \\
k > \texttt{fail}(\tau) & \mapsto_{\mathsf{c}} & k \ll \mathsf{fail} \\
k \triangleright f \ll \mathsf{fail} & \mapsto_{\mathsf{c}} & k \ll \mathsf{fail} \qquad \text{for } f \neq \texttt{try}(\Box, \_) \\
k \triangleright \texttt{try}(\Box, e_2) \ll \mathsf{fail} & \mapsto_{\mathsf{c}} & k > e_2
\end{array}$$

In order to verify that these rules are sensible, we should prove appropriate progress and preservation theorems. In order to do this, we need to introduce some typing judgments for machine states and the new forms of expressions. First, expressions:

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

The new judgment for typing states depends on a typing for stacks. A stack is characterized by the type of the argument it expects and the type of the final answer it returns. We write ρ for the type of the final answer. Note that during the whole computation of a machine, this never changes. The new judgments are

$s\ \ \mathsf{OK}_\rho$	state s is well-formed with final answer type ρ
$k\ : \tau\ \mathsf{stack}_\rho$	stack k accepts a value of type τ and returns a final answer of type ρ

Since ρ never changes for any run of the machine, we omit the subcript in some of the rules below. However, keep in mind that it is implicitly present. Note also that judgments on states and stacks do not need to be hypothetical judgments, since they never contain free variables. First, the rules for states which ensure that the type expected by a stack matches the type of the expression to be evaluated, or value being returned.

[2] The type is written here in order to preserve the property that every well-typed expression has a unique type.

$$\frac{k \ : \tau\ \mathsf{stack}_\rho \quad \cdot \vdash e : \tau}{k > e \ \ \mathsf{OK}_\rho}$$

$$\frac{k \ : \tau\ \mathsf{stack}_\rho \quad \cdot \vdash v : \tau \quad v\ \ \mathsf{value}}{k < v \ \ \mathsf{OK}_\rho}$$

$$\frac{k \ : \tau\ \mathsf{stack}_\rho}{k \ll \mathsf{fail}\ \ \mathsf{OK}_\rho}$$

The rules for stacks are straightforward, given a few examples below.

$$\frac{}{\bullet \ : \rho\ \mathsf{stack}_\rho}$$

$$\frac{k \ : \tau_1\ \mathsf{stack} \quad \cdot \vdash e_2 : \tau_2}{k \triangleright \mathtt{apply}(\Box, e_2) \ : \tau_2 \to \tau_1\ \mathsf{stack}}$$

$$\frac{k \ : \tau_1\ \mathsf{stack} \quad \cdot \vdash v_1 : \tau_2 \to \tau_1 \quad v_1\ \mathsf{value}}{k \triangleright \mathtt{apply}(v_1, \Box) \ : \tau_2\ \mathsf{stack}}$$

$$\frac{k \ : \tau\ \mathsf{stack} \quad \cdot \vdash e_2 : \tau \quad \cdot \vdash e_3 : \tau}{k \triangleright \mathtt{if}(\Box, e_2, e_3) \ : \mathtt{bool}\ \mathsf{stack}}$$

$$\frac{k \ : \tau\ \mathsf{stack} \quad \cdot \vdash e_2 : \tau}{k \triangleright \mathtt{try}(\Box, e_2) \ : \tau\ \mathsf{stack}}$$

$$\frac{k \ : \tau_2\ \mathsf{stack} \quad x{:}\tau_1 \vdash e_2 : \tau_2}{k \triangleright \mathtt{let}(\Box, x.e_2) \ : \tau_1\ \mathsf{stack}}$$

We can now state (without proof) the preservation and progress properties. The proofs follow previous patterns (see [Ch. 13]) and Lecture 5 on *Type Safety*.

1. (Preservation) If $s\ \ \mathsf{OK}_\rho$ and $s \mapsto s'$ then $s'\ \ \mathsf{OK}_\rho$.

2. (Progress) If $s\ \ \mathsf{OK}_\rho$ then either

   (i) $s \mapsto s'$ for some s', or

   (ii) $s = \bullet < v$ with v value, or

   (iii) $s = \bullet \ll$ fail.

The manner in which the C-machine operates with respect to exceptions may seem a bit unrealistic, since the stack is unwound frame by frame. However, in languages like Java this is not an unusual implementation method. In ML, there is more frequently a second stack containing only handlers for exceptions. The handler at the top of the stack is innermost and a `fail` expression can jump to it directly.

Overall, this machine should be equivalent to the specification of exceptions above, but potentially more efficient. Often, we want to describe several aspects of execution behavior of a language constructs in several different machines, keeping the first as high-level as possible.

In our simple language, the handler stack h contains only frames $\mathtt{ow}(k, e_2)$ while the control stack contains the usual frames, and $\mathtt{try}(\Box)$ (the "otherwise" clause has moved to the handler stack). All the usual rules are augmented to carry a control stack and a handler stack, and leave the handler unchanged.

$$\begin{array}{lll}
(h, k) > \mathtt{apply}(e_1, e_2) & \mapsto_{\mathsf{c}} & (h, k \triangleright \mathtt{apply}(\Box, e_2)) > e_1 \\
\ldots & & \\
(h, k) > \mathtt{try}(e_1, e_2) & \mapsto_{\mathsf{c}} & (h \triangleright \mathtt{ow}(k, e_2), k \triangleright \mathtt{try}(\Box)) > e_1 \\
(h \triangleright \mathtt{ow}(k', e_2), k \triangleright \mathtt{try}(\Box)) < v_1 & \mapsto_{\mathsf{c}} & (h, k) < v_1 \\
(h \triangleright \mathtt{ow}(k', e_2), k) > \mathtt{fail}(\tau) & \mapsto_{\mathsf{c}} & (h, k') > e_2
\end{array}$$

Note that we do not unwind the control stack explicilty, but jump directly to the handler when an exception is raised. This handler must story a copy of the control stack in effect at the time the `try` expression was executed. Fortunately, this can be implemented without the apparent copying of the stack in the rule for `try`, because we can just keep a pointer to the right frame in the control stack [Ch. 13].

Note also in case of a regular return for the subject of a `try` expression, we need to pop the corresponding handler off the handler stack.