Lecture Notes on
Monadic Input and Output

15-312: Foundations of Programming Languages
Frank Pfenning

Lecture 17
October 28, 2004

After reviewing the basic idea behind the encapsulation of effects, we introduce input and output as a specific kind of effect. After store effects in the last lecture, this will be our second example. For simplicity, we don't consider store and input/output simultaneously.

In review, in a pure functional language programs are evaluated only to obtain a value. A loose characterization of an effect is simply everything else that a function might perform. Allocating, mutating, and reading storage cells is one example of effects, input and output are two more. We say that effects are *encapsulated* if they do not interfere with the meaning of the pure expressions in a language. Standard ML does not encapsulate effects, Haskell does.

Encapsulation of effects is achieved by separating *pure expressions* (e) from potentially *effectful expressions* (written as m). All the usual constructs in MinML remain pure expressions.

$$\begin{array}{rcll} \text{Types} & \tau & ::= & \tau_1 \to \tau_2 \mid \cdots \mid \tau\ \mathsf{ref} \mid \bigcirc\tau \\ \text{Pure Expressions} & e & ::= & \mathsf{fn}(x.e) \mid \mathsf{apply}(e_1, e_2) \mid \cdots \mid \mathsf{comp}(m) \\ \text{Monadic Expressions} & m & ::= & e \mid \mathsf{letcomp}(e, x.m) \\ & & & \mid \mathsf{ref}(e) \mid \mathsf{assign}(e_1, e_2) \mid \mathsf{deref}(e) \end{array}$$

In the concrete syntax, we write τ `comp` for $\bigcirc\tau$ and $\mathsf{let\ comp}\ x = e\ \mathsf{in}\ m\ \mathsf{end}$ for $\mathsf{letcomp}(e, x.\,m)$. As an example, consider the following signature and implementation.

```
signature C =
sig
```

```
  type c
  val new : c comp
  val inc : c -> unit comp
  val get : c -> int comp
end;

structure C :> C =
struct
  type c = int ref
  val new = comp (ref 0)
  val inc = fn r => comp (let comp x = !r in r := x+1 end)
  val get = fn r => comp (!r)
end;
```

Now we can create a cell, increment and read it with the following (monadic) expression, using a slight shorthand by allowing multiple declarations as in Standard ML.

```
let
  comp x = C.new
  comp _ = C.inc x
  comp y = C.get x
in
  y
end;
```

When started in the empty memory, the above monadic expression executes and evaluates to $\langle (l{=}1), 1\rangle$ for some l. It is worth writing out this computation step by step to see exactly how computation proceeds and effects and effect-free computations may be interleaved.

In order to model input and output, the state that monadic expressions may refer to contains two streams: an input stream and an output stream. Streams are potentially infinite sequence of integers, $k_1 \cdots k_n \cdots$. The empty stream is denoted by ϵ. We denote streams by s and write (s_I, s_O) for the pair of input and output streams. We have the constructs read, eof and $\mathsf{write}(e)$ for reading from the input stream, testing if the input stream is empty, and writing the value of e to the output stream, respectively. These constructs must be monadic expressions, since they have an effect.

$$\frac{}{\Gamma \vdash \mathsf{read} \div \mathsf{int}}$$

$$\frac{}{\Gamma \vdash \mathsf{eof} \div \mathsf{bool}}$$

$$\frac{\Gamma \vdash e : \mathsf{int}}{\Gamma \vdash \mathsf{write}(e) \div 1}$$

As in the case of mutable storage, the operational semantics distinguishes states for monadic expressions (which must include the input and output streams) and pure expressions (which does not include the input and output streams).

$$\frac{}{\langle (k \cdot s_I, s_O), \mathsf{read} \rangle \mapsto \langle (s_I, s_0), \mathsf{int}(k) \rangle}\ \mathit{Read}$$

$$\frac{}{\langle (k \cdot s_I, s_O), \mathsf{eof} \rangle \mapsto \langle (k \cdot s_I, s_O), \mathsf{false} \rangle}\ \mathit{EofFalse}$$

$$\frac{}{\langle (\epsilon, s_O), \mathsf{eof} \rangle \mapsto \langle (\epsilon, s_O), \mathsf{true} \rangle}\ \mathit{EofTrue}$$

$$\frac{e \mapsto e'}{\langle (s_I, s_O), \mathsf{write}(e) \rangle \mapsto \langle (s_I, s_O), \mathsf{write}(e') \rangle}\ \mathit{WriteArg}$$

$$\frac{v \;\; \mathsf{value}}{\langle (s_I, s_O), \mathsf{write}(\mathsf{int}(k)) \rangle \mapsto \langle (s_I, k \cdot s_O), \langle\,\rangle \rangle}\ \mathit{Write}$$

This language extension seems simple, although it is not completely trivial to write programs using the monadic syntax. Moreover, we have to formulate the progress theorem carefully. In fact, with the stated rules it fails! The reason is that if we try to read from an empty input stream no rule is applicable. It is quite instructive to see where the proof of progress fails unless we incorporate the possibility that m may be blocked. In a concurrent language (or even in a realistic sequential language) such blocking on input can certainly occur, so it seems reasonable to allow for it and model it. Of course, our language has no explicit mechanism of unblocking, but this will change later on. So we are aiming at the following version of the progress theorem.

Theorem 1 (Progress)

(1) If $\cdot \vdash e : \tau$ *then either*

- *(i)* $e \mapsto e'$ *for some* e'*, or*
- *(ii)* $e\ \mathsf{value}$

(2) If $\cdot \vdash m \div \tau$ *then either*

- *(i)* $\langle (s_I, s_O), m \rangle \mapsto \langle (s'_I, s'_O), m' \rangle$ *for some* s'_I*,* s'_O *and* m'*, or*
- *(ii)* $m = v$ *and* $v\ \mathsf{value}$*, or*
- *(iii)* $s_I = \epsilon$ *and* $m\ \mathsf{blocked}$*.*

The first thought on how to define the judgment $m\ \mathsf{blocked}$ would be to simply write

$$\frac{}{\mathsf{read\ blocked}}\ \mathit{BlockedRead}$$

However, this is not enough as, again, a failed proof of the progress theorem should tell you. We may also be in the situation where the read is not at the top level, but is the first monadic expression to be executed. In other words, the search rules may lead us to a read expression. The general way to capture this is with the rule

$$\frac{m_1\ \mathsf{blocked}}{\mathsf{letcomp}(\mathsf{comp}(m_1), x.m_2)\ \mathsf{blocked}}\ \mathit{BlockedLet}$$

Note that we never need to look at m_2, nor do we need to account for the case of $\mathsf{letcomp}(e, x.m_2)$ where e is not a value, because a pure expression e cannot have an effect unless it is situated as in the *BlockedLet* rule.

With the right definition of blocked states, it is then easy to prove the progress theorem, employing value inversion as in other progress proofs we have carried out up to this point. We just have to be sure to cover all the possible cases.

We close the lecture with two simple examples. The first is a (non-recursive) computation which reads one integer and then writes to the output.

```
val copyOne : unit comp =
  comp (let comp x = read in write x end)
```

The next one is a recursive function to copy the whole input stream to the output stream. This function should loop forever, if the input stream is infinite.

```
val copy : unit comp =
  rec copy =>
    comp (let
            comp b = eof
            comp x = if b then comp (let
                                         comp _ = copyOne
                                         comp _ = copy
                                       in () end)
                          else comp ()
           in () end)
```

This last example has some subtleties. For example, the conditional is a pure expression (and not a monadic expression). In order to properly interleave pure and effectful computation, it must be essentially where it is: on the right-hand side of a comp declaration, where both branches are again computations.