Lecture Notes on Monads

15-312: Foundations of Programming Languages
Frank Pfenning

Lecture 15
October 14, 2003

The way we have extended MinML with mutable storage has several drawbacks. The principal difficulty with programming with effects is that the type system does not track them properly. So when we examine the type of a function $\tau_1 \to \tau_2$ we cannot tell if the function simply returns a value of type τ_2 or if it could also have an effect. This complicates reasoning about programs and their correctness tremendously.

An alternative is to try to express in the type system that certain functions may have effects, while others do not have effects. This is the purpose of *monads* that are quite popular in the Haskell community. Haskell is a lazy[1] functional language in which all effects are isolated in a monad. We will see that monadic programming has its own drawbacks. The last word in the debate on how to integrate imperative and pure functional programming has not yet been spoken.

We introduce monads in two steps. The first step is the generic framework, which can be instantiated to different kinds of effects. In this lecture we introduce mutable storage as an effect, just as we did in the previous lecture on mutable storage in ML. In Assignment 5 you are asked to instantiate the monadic framework instead by defining a simple semantics of input and output.

In the generic framework, we extend MinML by adding a new syntactic category of *monadic expressions*, denoted by m. Correspondingly, there is a new typing judgment

$$\Gamma \vdash m \div \tau$$

[1] Lazy here means call-by-name with memoization of the suspension.

expressing that the monadic expression m has type τ in context Γ. We think of a monadic expression as one whose evaluation returns not only a value of type τ, but also has an effect. We introduce this separate category so that the ordinary expressions we have used so far can remain pure, that is, free of effects.

Any particular use of the monadic framework will add particular new monadic expressions, and also possibly new pure expressions. But first the constructs that are independent of the kind of effect we want to consider. The first principle is that a pure expression e can be considered as a monadic expression $[e]$ which happens to have no effect.

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash [e] \div \tau}$$

The notation $[e]$ should not be confused with the concrete syntax for lists in ML.

The second idea is that we can quote a monadic expression and thereby turn it into a pure expression. It has no effects because the monadic expression will not be executed. We write the quotation operator as $\mathsf{val}(m)$.

$$\frac{\Gamma \vdash m \div \tau}{\Gamma \vdash \mathsf{val}(m) : \bigcirc\tau} \qquad \frac{}{\mathsf{val}(m)\ \ \mathsf{value}}$$

Finally, we must be able to unwrap and thereby actually execute a quoted monadic expression. However, we cannot do this anywhere in a pure expression, because evaluating such a supposedly pure expression would then have an effect. Instead, we can only do this if we are within an explicit sequence of monadic expressions! This yields the following construct

$$\frac{\Gamma \vdash e : \bigcirc\tau \quad \Gamma, x{:}\tau \vdash m \div \sigma}{\Gamma \vdash \mathsf{let\ val}\, x = e\, \mathsf{in}\, m\, \mathsf{end} \div \sigma}$$

Note that m and $\mathsf{let\ val}\, x = e\, \mathsf{in}\, m\, \mathsf{end}$ are monadic expressions (and therefore may have an effect), while e is a pure expression of monadic type. We think of the effects are being staged as follows:

(1) We evaluate e which should yield a value $\mathsf{val}(m')$.

(2) We execute the monadic expression m', which will have some effects but also return a value in the form $[v]$.

(3) Substitute v for x in m and then execute the resulting monadic expression.

In order to specify this properly we need to be able to describe the effect that may be engendered by executing a monadic expression. For this we introduce the concept of *worlds* w that encapsulate all state that may be changed by an effect. In the case of the storage monads, this will be the memory M. In the case of the I/O monad, this will be input and output streams.

The judgment for executing monadic expressions then has the form

$$\langle w, m\rangle \mapsto \langle w', m'\rangle$$

where the world changes from w to w' and the expression steps from m to m'. According to the considerations above, we obtain the following rules.

$$\frac{e \mapsto e'}{\langle w, [e]\rangle \mapsto \langle w, [e']\rangle}$$

We can see that the transition judgment on ordinary expressions looks the same as before and that it can have no effect. Contrast this with the situation in ML from the previous lecture where we needed to change *every* transition rule to account for possible effects.

The next sequence of three rules implement items (1), (2), and (3) above.

$$\frac{e \mapsto e'}{\langle w, \mathsf{let\ val}\ x = e\ \mathsf{in}\ m\ \mathsf{end}\rangle \mapsto \langle w, \mathsf{let\ val}\ x = e'\ \mathsf{in}\ m\ \mathsf{end}\rangle}$$

$$\frac{\langle w, m_1\rangle \mapsto \langle w', m_1'\rangle}{\langle w, \mathsf{let\ val}\ x = \mathsf{val}(m_1)\ \mathsf{in}\ m\ \mathsf{end}\rangle \mapsto \langle w', \mathsf{let\ val}\ x = \mathsf{val}(m_1')\ \mathsf{in}\ m\ \mathsf{end}\rangle}$$

$$\frac{}{\langle w, \mathsf{let\ val}\ x = \mathsf{val}([v])\ \mathsf{in}\ m\ \mathsf{end}\rangle \mapsto \langle w, \{v/x\}m\rangle}$$

Note that the substitution in the last rule is appropriate. The substitution principle for pure values into monadic expressions is straightforward precisely because v is cannot have effects.

We will not state here the generic forms of the preservation and progress theorems. They are somewhat trivialized because our language, while designed with effects in mind, does not yet have any actual effects.

In order to define the monad for mutable storage we introduce a new form of type, $\tau\ \mathsf{ref}$ and three new forms of monadic expressions, namely $\mathsf{ref}(e)$, $e_1 := e_2$ and $!e$. In addition we need one new form of pure expression, namely locations l which are declared in a store typing Λ with their type. Recall the form of store typings.

$$\textit{Store Typings} \qquad \Lambda ::= \cdot \mid \Lambda, l{:}\tau$$

Locations can be pure because creating, assigning, or dereferencing them is an effect, and the types prevent any other operations on them. The store typing must now be taking into account when checking expressions that are created a runtime. They are, however, not needed for compile-time checking because the program itself, before it is started, cannot directly refer to locations. We just uniformly add "Λ;" to all the typing judgments—they are simply additional hypotheses of a slightly different form than what is recorded in Γ.

$$\frac{\Lambda;\Gamma \vdash e : \tau}{\Lambda;\Gamma \vdash \mathsf{ref}(e) \div \tau\ \mathsf{ref}} \qquad \frac{\Lambda;\Gamma \vdash e_1 : \tau\ \mathsf{ref} \quad \Lambda;\Gamma \vdash e_2 : \tau}{\Lambda;\Gamma \vdash e_1 := e_2 \div \tau}$$

$$\frac{\Lambda;\Gamma \vdash e : \tau\ \mathsf{ref}}{\Lambda;\Gamma \vdash !e \div \tau} \qquad \frac{l{:}\tau\ \text{in}\ \Lambda}{\Lambda;\Gamma \vdash l : \tau\ \mathsf{ref}}$$

Note that the constituents of the new monadic expressions are pure expressions. This guarantees that they cannot have effects: all effects must be explicitly sequenced using the letval form.

In order to describe the operational semantics we need to make the worlds explicit. In this case a world consists simply of the current memory M. Recall the form of stores.

$$\textit{Stores} \qquad M ::= \cdot \mid M, l{=}v$$

Now the additional rules for new expressions are analogous to those from the previous lecture.

$$\frac{e \mapsto e'}{\langle M, \mathsf{ref}(e)\rangle \mapsto \langle M, \mathsf{ref}(e')\rangle} \qquad \frac{v\ \ \mathsf{value}}{\langle M, \mathsf{ref}(v)\rangle \mapsto \langle (M, l{=}v), [l]\rangle} \qquad \frac{}{l\ \ \mathsf{value}}$$

$$\frac{e_1 \mapsto e_1'}{\langle M, e_1 := e_2\rangle \mapsto \langle M, e_1' := e_2\rangle} \qquad \frac{v_1\ \ \mathsf{value} \quad e_2 \mapsto e_2'}{\langle M, v_1 := e_2\rangle \mapsto \langle M, v_1 := e_2'\rangle}$$

$$\frac{M = (M_1, l{=}v_1, M_2) \quad \text{and} \quad M' = (M_1, l{=}v_2, M_2)}{\langle M, l := v_2\rangle \mapsto \langle M', [v_2]\rangle}$$

$$\frac{e \mapsto e'}{\langle M, !e\rangle \mapsto \langle M, !e'\rangle} \qquad \frac{M = (M_1, l{=}v, M_2)}{\langle M, !l\rangle \mapsto \langle M, [v]\rangle}$$

The progress and type preservation theorems now need to be extended to cover both pure and monadic expressions. We also seen to verify that a store satisfies a store typing. Recall the rules for the judgment $\Lambda_0; \cdot \vdash M : \Lambda$.

$$\frac{}{\Lambda_0; \cdot \vdash (\cdot) : (\cdot)} \qquad \frac{\Lambda_0; \cdot \vdash M : \Lambda \quad \Lambda_0; \cdot \vdash v : \tau \quad v\ \ \mathsf{value}}{\Lambda_0; \cdot \vdash (M, l{=}v) : (\Lambda, l{:}\tau)}$$

We can now formulate the appropriate generalizations of type preservation and progress. We write $\Lambda' \geq \Lambda$ if Λ' is an extension of the store typing Λ with some additional locations. In this particular case, for a single step, we need at most one new location.

Theorem 1 (Type Preservation)
(1) If $\Lambda; \cdot \vdash e : \tau$ *and* $e \mapsto e'$ *then* $\Lambda; \cdot \vdash e' : \tau$.

(2) If $\Lambda; \cdot \vdash m \div \tau$ *and* $\Lambda; \cdot \vdash M : \Lambda$ *and* $\langle M, m\rangle \mapsto \langle M', m'\rangle$ *then for some* $\Lambda' \geq \Lambda$ *and memory* M' *we have* $\Lambda'; \cdot \vdash m' \div \tau$ *and* $\Lambda'; \cdot \vdash M' : \Lambda'$.

Proof: By induction on the derivation of the computation judgment, applying inversion on the typing assumptions. ■

Theorem 2 (Progress)
(1) If $\Lambda; \cdot \vdash e : \tau$ *then either*

(i) e value, *or*

(ii) $e \mapsto e'$ *for some* e'.

(2) If $\Lambda; \cdot \vdash m \div \tau$ *and* $\Lambda; \cdot \vdash M : \Lambda$ *then either*

(i) $m = [v]$ *for some* v value, *or*

(ii) $\langle M, m\rangle \mapsto \langle M', m'\rangle$ *for some* M' *and* m'.

Proof: By induction on the derivation of the typing judgment, analyzing all possible cases. ■

As an example consider a function inc : int ref $\rightarrow \bigcirc$int which takes a location as an argument, increments it, and returns the incremented value. The return type has to be protected by the monadic type, since the function has an effect.

$$
\begin{array}{rcl}
inc & : & \mathsf{int\ ref} \rightarrow \bigcirc\mathsf{int} \\
& = & \lambda r.\mathsf{val}(\mathsf{let\ val}(x_1) = \mathsf{val}(!r)\ \mathsf{in} \\
& & \qquad \mathsf{let\ val}(x_2) = \mathsf{val}(r := x_1 + 1)\ \mathsf{in} \\
& & \qquad [x_2]\mathsf{end\ end})
\end{array}
$$

Several things to note about this definition. When inc is called on a location, it *returns* an effectful computation, but it does not carry it out (val(m) quotes m). Secondly, the first let expression is necessary, because $r := !r + 1$

incorrectly uses the monadic expression $!r$ in a place where a pure expression is expected. The uses of $\mathsf{val}(m)$ on the right-hand side of the lets can be avoid by introducing appropriate definitions such as

$$\begin{array}{lcl}
\mathsf{new} & : & \forall t.t \to \bigcirc(t\,\mathsf{ref}) \\
 & = & \Lambda t.\lambda x.\mathsf{val}(\mathsf{ref}(x)) \\
\mathsf{get} & : & \forall t.t\,\mathsf{ref} \to \bigcirc t \\
 & = & \Lambda t.\lambda r.\mathsf{val}(!r) \\
\mathsf{set} & : & \forall t.t\,\mathsf{ref} \to t \to \bigcirc t \\
 & = & \Lambda t.\lambda r.\lambda x.\mathsf{val}(r := x)
\end{array}$$

Furthermore, the Haskell language creates some syntactic sugar that makes it easier to write a sequence of let val forms in a row.

In order to create a cell, initialize it with 0, increment it once and return the cell's contents we can write the following monadic expression at the top level.

$$\begin{array}{l}
\mathsf{let}\ \mathsf{val}(x_1) = new(0)\ \mathsf{in} \\
\mathsf{let}\ \mathsf{val}(x_2) = inc(x_1)\ \mathsf{in} \\
\mathsf{let}\ \mathsf{val}(x_3) = get(x_1)\ \mathsf{in} \\
[x_3]\ \mathsf{end}\ \mathsf{end}\ \mathsf{end} \\
\div\mathsf{int}
\end{array}$$

When started in the empty memory, the above monadic expression executes and evaluates to $\langle (l{=}1), 1\rangle$. 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.