Functional Programming with Effects

Andrzej Filinski
aof@dcs.ed.ac.uk

ML Club, 4 December 1997

Overview

- Adding an effect (e.g., exceptions) to a functional language
  - Bad: *rework* entire existing semantics
  - Good: *instantiate* generic semantics [monads]
  - Better: *define* new effect within the language [monadic reflection]
- Implementing general programmer-defined effects with fixed, low-level ones
- Conclusions, future work

A simple functional language

Let us define a Tiny Programming Language:

$$
\begin{aligned}
\sigma &::= \iota \mid \sigma_1 \to \sigma_2 \\
E &::= x \mid \mathsf{z} \mid \mathsf{s}\,E \mid \mathsf{case}_\sigma(E_1, E_2, x.\,E_3) \\
&\quad\ \mid \mathsf{fun}\ f(x^{\,\sigma_1})^{\,\sigma_2}.\,E \mid E_1\,(E_2) \\
f, x &\in \mathit{Vars}
\end{aligned}
$$

Abbrev: $\lambda x.\,E \equiv \mathsf{fun}\ f(x).\,E$ with $f \notin FV(E)$.

Typing rules, $\Gamma \vdash E : \sigma$:

$$
\frac{\Gamma(x) = \sigma}{\Gamma \vdash x : \sigma}
\qquad
\frac{}{\Gamma \vdash \mathsf{z} : \iota}
\qquad
\frac{\Gamma \vdash E : \iota}{\Gamma \vdash \mathsf{s}\,E : \iota}
$$

$$
\frac{\Gamma \vdash E_1 : \iota \qquad \Gamma \vdash E_2 : \sigma \qquad \Gamma, x{:}\,\iota \vdash E_3 : \sigma}{\Gamma \vdash \mathsf{case}_\sigma(E_1, E_2, x.\,E_3) : \sigma}
$$

$$
\frac{\Gamma, f{:}\,\sigma_1 \to \sigma_2, x{:}\,\sigma_1 \vdash E : \sigma_2}{\Gamma \vdash \mathsf{fun}\ f(x^{\,\sigma_1})^{\,\sigma_2}.\,E : \sigma_1 \to \sigma_2}
$$

$$
\frac{\Gamma \vdash E_1 : \sigma_1 \to \sigma_2 \qquad \Gamma \vdash E_2 : \sigma_1}{\Gamma \vdash E_1\,(E_2) : \sigma_2}
$$

Ex: $\mathsf{add} \equiv \lambda x.\,\mathsf{fun}\ a(y).\,\mathsf{case}(y, x, y'.\,\mathsf{s}\,(a\,(y')))$.

$\cdot \vdash \mathsf{add} : \iota \to \iota \to \iota$.

Operational semantics of TPL

Call-by-value, "big-step" semantics.

A subset of terms are called *values*:

$$V ::= \mathsf{z} \mid \mathsf{s}\,V \mid \mathsf{fun}\ f(x).\,E$$

For closed E and V, define the relation $E \Downarrow V$

$$\frac{}{\mathsf{z} \Downarrow \mathsf{z}} \qquad \frac{E \Downarrow V}{\mathsf{s}\,E \Downarrow \mathsf{s}\,V} \qquad \frac{}{\mathsf{fun}\ f(x).\,E \Downarrow \mathsf{fun}\ f(x).\,E}$$

$$\frac{E_1 \Downarrow \mathsf{z} \qquad E_2 \Downarrow V}{\mathsf{case}(E_1, E_2, x.\,E_3) \Downarrow V} \qquad \frac{E_1 \Downarrow \mathsf{s}\,V' \qquad E_3\{V'/x\} \Downarrow V}{\mathsf{case}(E_1, E_2, x.\,E_3) \Downarrow V}$$

$$\frac{E_1 \Downarrow \mathsf{fun}\ f(x).\,E_1' \qquad E_2 \Downarrow V_2 \qquad E_1'\{(\mathsf{fun}\ f(x).\,E_1')/f, V_2/x\} \Downarrow V}{E_1\,(E_2) \Downarrow V}$$

Denotational semantics of TPL

The idea of denotational semantics is to assign meanings to program phrases in a *compositional* way, so that the meaning of a phrase is independent of its context.

Every well-typed program has a meaning; model non-termination explicitly.

Interpret a well-formed type σ as a cpo $\mathcal{V}[\![\sigma]\!]$:

$$\begin{aligned} \mathcal{V}[\![\iota]\!] &= \mathbf{N} \\ \mathcal{V}[\![\sigma_1 \to \sigma_2]\!] &= \mathcal{V}[\![\sigma_1]\!] \to \mathcal{V}[\![\sigma_2]\!]_\perp \end{aligned}$$

Ex: $\mathcal{V}[\![\iota \to \iota]\!] = \mathbf{N} \to \mathbf{N}_\perp$. "Partial functions."

Interpret a well-formed type assignment $\Gamma = (x_1 : \sigma_1, \ldots, x_n : \sigma_n)$ as a product cpo $\mathcal{V}[\![\Gamma]\!]$:

$$\mathcal{V}[\![\Gamma]\!] = \prod_{x \in \mathrm{dom}\,\Gamma} \mathcal{V}[\![\Gamma(x)]\!]$$

(where $\mathrm{dom}\,\Gamma = \{x_1, \ldots, x_n\}$ and $\Gamma(x_i) = \sigma_i$). An element of $\mathcal{V}[\![\Gamma]\!]$ is simply a map from variable names to elements of the appropriate cpos.

Semantics of TPL terms

Meaning of well-typed term $\Gamma \vdash E : \sigma$ is a cont. function $\mathcal{V}[\![E]\!] : \mathcal{V}[\![\Gamma]\!] \to \mathcal{V}[\![\sigma]\!]_\perp$,

$$\begin{array}{rcl}
\mathcal{V}[\![x]\!]\,\rho & = & \uparrow(\rho\,x) \\
\mathcal{V}[\![\mathsf{z}]\!]\,\rho & = & \uparrow 0 \\
\mathcal{V}[\![\mathsf{s}\,E]\!]\,\rho & = & (\lambda n.\uparrow(n+1))^\dagger\,(\mathcal{V}[\![E]\!]\,\rho) \\
\mathcal{V}[\![\mathsf{case}(E_1, E_2, x.\,E_3)]\!]\,\rho & = & \phi^\dagger\,(\mathcal{V}[\![E_1]\!]\,\rho) \text{ where } \begin{cases} \phi(0) = \mathcal{V}[\![E_2]\!]\,\rho \\ \phi(n+1) = \mathcal{V}[\![E_3]\!]\,(\rho[x \mapsto n]) \end{cases} \\
\mathcal{V}[\![\mathsf{fun}\ f\,(x^{\,\sigma_1})^{\,\sigma_2}.\,E]\!]\,\rho & = & \uparrow(\mathsf{fix}_{\mathcal{V}[\![\sigma_1]\!] \to \mathcal{V}[\![\sigma_2]\!]_\perp}\,(\lambda\phi.\,\lambda a.\,\mathcal{V}[\![E]\!]\,(\rho[f \mapsto \phi][x \mapsto a]))) \\
\mathcal{V}[\![E_1\,(E_2)]\!]\,\rho & = & (\lambda\phi.\,\phi^\dagger\,(\mathcal{V}[\![E_2]\!]\,\rho))^\dagger\,(\mathcal{V}[\![E_1]\!]\,\rho)
\end{array}$$

($\uparrow$ is inclusion $A \to A_\perp$; $f^\dagger : A_\perp \to B$ is strict extension of $f : A \to B$, B ptd.)

Ex (since $\mathsf{fix}(\lambda x.\,a) = a$): $\mathcal{V}[\![\lambda x.\,E]\!]\,\rho = \uparrow(\lambda a.\,\mathcal{V}[\![E]\!]\,(\rho[x \mapsto a]))$

A simple effect: exceptions

Extend syntax of TPL terms as follows:

$$E ::= \cdots \mid \mathsf{fail} \mid \mathsf{try}\ E_1\ \mathsf{else}\ E_2$$

with corresponding typing rules:

$$\frac{}{\Gamma \vdash \mathsf{fail} : \sigma} \qquad \frac{\Gamma \vdash E_1 : \sigma \qquad \Gamma \vdash E_2 : \sigma}{\Gamma \vdash \mathsf{try}\ E_1\ \mathsf{else}\ E_2 : \sigma}$$

These correspond to a limited use of SML's exceptions:

	`exception Fail;`
fail	`raise Fail`
try E_1 else E_2	E_1 `handle Fail =>` E_2

Operational semantics of exceptions

For the operational semantics, define the set of *results*, $R ::= V \mid \mathsf{fail}$.

And then the evaluation relation $E \Downarrow R$ as follows:

$$\frac{}{\mathsf{z} \Downarrow \mathsf{z}} \qquad \frac{E \Downarrow V}{\mathsf{s}\,E \Downarrow \mathsf{s}\,V} \qquad \frac{E \Downarrow \mathsf{fail}}{\mathsf{s}\,E \Downarrow \mathsf{fail}} \qquad \frac{}{\mathsf{fun}\ f(x).\,E \Downarrow \mathsf{fun}\ f(x).\,E}$$

$$\frac{E_1 \Downarrow \mathsf{z} \qquad E_2 \Downarrow R}{\mathsf{case}(E_1, E_2, x.\,E_3) \Downarrow R} \qquad \frac{E_1 \Downarrow \mathsf{s}\,V' \qquad E_3\{V'/x\} \Downarrow R}{\mathsf{case}(E_1, E_2, x.\,E_3) \Downarrow R} \qquad \frac{E_1 \Downarrow \mathsf{fail}}{\mathsf{case}(E_1, E_2, x.\,E_3) \Downarrow \mathsf{fail}}$$

$$\frac{E_1 \Downarrow \mathsf{fun}\ f(x).\,E_1' \qquad E_2 \Downarrow V_2 \qquad E_1'\{(\mathsf{fun}\ f(x).\,E_1')/f, V_2/x\} \Downarrow R}{E_1\,(E_2) \Downarrow R}$$

$$\frac{E_1 \Downarrow \mathsf{fail}}{E_1\,(E_2) \Downarrow \mathsf{fail}} \qquad \frac{E_1 \Downarrow \mathsf{fun}\ f(x).\,E_1' \qquad E_2 \Downarrow \mathsf{fail}}{E_1\,(E_2) \Downarrow \mathsf{fail}}$$

$$\frac{}{\mathsf{fail} \Downarrow \mathsf{fail}} \qquad \frac{E_1 \Downarrow V}{\mathsf{try}\ E_1\ \mathsf{else}\ E_2 \Downarrow V} \qquad \frac{E_1 \Downarrow \mathsf{fail} \qquad E_2 \Downarrow R}{\mathsf{try}\ E_1\ \mathsf{else}\ E_2 \Downarrow R}$$

Denotational semantics of exceptions

New semantic function $\mathcal{V}_{\mathrm{e}}[\![-]\!]$.

Interpretation of types:

$$\begin{aligned} \mathcal{V}_{\mathrm{e}}[\![\iota]\!] &= \mathbf{N} \\ \mathcal{V}_{\mathrm{e}}[\![\sigma_1 \to \sigma_2]\!] &= \mathcal{V}_{\mathrm{e}}[\![\sigma_1]\!] \to (\mathcal{V}_{\mathrm{e}}[\![\sigma_2]\!] + 1)_\perp \end{aligned}$$

Interpretation of type assignments pointwise as before, $\mathcal{V}_{\mathrm{e}}[\![\Gamma]\!] = \prod_{x \in \mathrm{dom}\,\Gamma} \mathcal{V}_{\mathrm{e}}[\![\Gamma(x)]\!]$.

Define value-inclusion unction $\xi : A \to (A+1)_\perp$ by

$$\xi a = \uparrow(1, a)$$

When $f : A \to (B+1)_\perp$, define its *failure-extension* $f^\diamond : (A+1)_\perp \to (B+1)_\perp$ by

$$\begin{aligned} f^\diamond \perp &= \perp \\ f^\diamond(\uparrow(1,a)) &= f\,a \\ f^\diamond(\uparrow(2,*)) &= \uparrow(2,*) \end{aligned}$$

DS of exceptions (terms)

Interpret typed term $\Gamma \vdash E : \sigma$ as cont. function $\mathcal{V}_{\mathrm{e}}[\![E]\!] : \mathcal{V}_{\mathrm{e}}[\![\Gamma]\!] \to (\mathcal{V}_{\mathrm{e}}[\![\sigma]\!] + 1)_\perp$:

$$
\begin{array}{rcl}
\mathcal{V}_{\mathrm{e}}[\![x]\!]\,\rho & = & \xi\,(\rho\,x) \\
\mathcal{V}_{\mathrm{e}}[\![\mathsf{z}]\!]\,\rho & = & \xi\,0 \\
\mathcal{V}_{\mathrm{e}}[\![\mathsf{s}\,E]\!]\,\rho & = & (\lambda n.\,\xi\,(n+1))^{\diamond}\,(\mathcal{V}_{\mathrm{e}}[\![E]\!]\,\rho) \\
\mathcal{V}_{\mathrm{e}}[\![\mathsf{case}(E_1, E_2, x.\,E_3)]\!]\,\rho & = & \phi^{\diamond}\,(\mathcal{V}_{\mathrm{e}}[\![E_1]\!]\,\rho)\ \text{where}\ \begin{cases}\phi(0) = \mathcal{V}_{\mathrm{e}}[\![E_2]\!]\,\rho \\ \phi(n+1) = \mathcal{V}_{\mathrm{e}}[\![E_3]\!]\,(\rho[x \mapsto n])\end{cases} \\
\mathcal{V}_{\mathrm{e}}[\![\mathsf{fun}\ f\,(x^{\,\sigma_1})^{\,\sigma_2}.\,E]\!]\,\rho & = & \xi\,(\mathsf{fix}_{\mathcal{V}_{\mathrm{e}}[\![\sigma_1]\!] \to (\mathcal{V}_{\mathrm{e}}[\![\sigma_2]\!]+1)_\perp}\,(\lambda\phi.\,\lambda a.\,\mathcal{V}_{\mathrm{e}}[\![E]\!]\,(\rho[f \mapsto \phi][x \mapsto a]))) \\
\mathcal{V}_{\mathrm{e}}[\![E_1\,(E_2)]\!]\,\rho & = & (\lambda\phi.\,\phi^{\diamond}\,(\mathcal{V}_{\mathrm{e}}[\![E_2]\!]\,\rho))^{\diamond}\,(\mathcal{V}_{\mathrm{e}}[\![E_1]\!]\,\rho) \\
\mathcal{V}_{\mathrm{e}}[\![\mathsf{fail}]\!]\,\rho & = & \uparrow(2, *) \\
\mathcal{V}_{\mathrm{e}}[\![\mathsf{try}\ E_1\ \mathsf{else}\ E_2]\!]\,\rho & = & \phi^{\dagger}\,(\mathcal{V}_{\mathrm{e}}[\![E_1]\!]\,\rho)\ \text{where}\ \begin{cases}\phi((1,a)) = \xi\,a \\ \phi((2,*)) = \mathcal{V}_{\mathrm{e}}[\![E_2]\!]\,\rho\end{cases}
\end{array}
$$

Monads

Definition 1. *A* monad $\mathbf{T} = (T, \eta, -^*)$ *in the semantics consists of the following assignments: [Technically, this is called a Kleisli triple]*

- *To any cpo A, a cpo TA. The elements of TA represent* computations *of elements from A.*
- *To any cpo A, a function $\eta_A : A \to TA$. The element $\eta\, a$ represents the* effect-free *computation of a.*
- *To any cpos A and B, and function $f : A \to TB$, a function $f^* : TA \to TB$.*
  For a computation t, $f^*\, t$ represents the sequence *of computations consisting of evaluating t to a value a, followed by evaluation of $f\, a$.*

These must satisfy the following equations:

$$f^* \circ \eta_A = f \qquad \eta_A^* = \mathsf{id}_{TA} \qquad f^* \circ g^* = (f^* \circ g)^*$$

(+ additional technicalities: continuity, strength).

We further say that a monad is computational *if for any A, TA is pointed, and for any f, f^* is strict.*

Monadic denotational semantics

Let $\mathbf{T} = (T, \eta, -^*)$ be a monad. We then define the *monadic denotational semantics* of TPL as follows:

Interpret a type as a cpo:

$$\begin{aligned} \mathcal{V}_T[\![\iota]\!] &= \mathbf{N} \\ \mathcal{V}_T[\![\sigma_1 \to \sigma_2]\!] &= \mathcal{V}_T[\![\sigma_1]\!] \to T\mathcal{V}_T[\![\sigma_2]\!] \end{aligned}$$

Extend pointwise to typing assignments Γ as usual.

Interpret term $\Gamma \vdash E : \sigma$ as continuous function $\mathcal{V}_T[\![E]\!] : \mathcal{V}_T[\![\Gamma]\!] \to T\mathcal{V}_T[\![\sigma]\!]$. (Same as before, only with TA for $A_\perp$, etc.)

Note that by instantiating this generic semantics with the lifting monad $(\text{-}_\perp, \uparrow, \text{-}^\dagger)$, we obtain exactly the original semantics of TPL.

But in the framework, we can also give meanings to various *extensions* of TPL, to express actual effectful computations.

Example: exceptions

For any cpo X, the *exception monad* is given by:

$$
\begin{aligned}
TA &= (A+X)_\perp \\
\eta a &= \uparrow(1,a) \\
f^* t &= \begin{cases} \perp & \text{if } t = \perp \\ f\,a & \text{if } t = \uparrow(1,a) \\ \uparrow(2,x) & \text{if } t = \uparrow(2,x) \end{cases}
\end{aligned}
$$

(This slightly generalizes our previous notion of exceptions, where we took $X = 1$.)

Let σ_{exn} be some base type of exception names in TPL. Then we can define the extensions:

$$
\frac{\Gamma \vdash E : \sigma_{exn}}{\Gamma \vdash \textsf{raise}\ E : \sigma}
\qquad
\frac{\Gamma \vdash E_1 : \sigma \qquad \Gamma, x{:}\,\sigma_{exn} \vdash E_2 : \sigma}{\Gamma \vdash E_1\ \textsf{handle}\ x \Rightarrow E_2 : \sigma}
$$

Taking the monadic semantics with $X = \mathcal{V}[\![\sigma_{exn}]\!]$, we can define

$$
\begin{aligned}
\mathcal{V}_T[\![\textsf{raise}\ E]\!]\,\rho &= (\lambda e.\uparrow(2,e))^*\,(\mathcal{V}_T[\![E]\!]\,\rho) \\
\mathcal{V}_T[\![E_1\ \textsf{handle}\ x \Rightarrow E_2]\!]\,\rho &= \phi^\dagger\,(\mathcal{V}_T[\![E_1]\!]\,\rho) \\
&\quad \text{where } \begin{cases} \phi((1,a)) = \uparrow(1,a) \\ \phi((2,e)) = \mathcal{V}_T[\![E_2]\!]\,\rho[x \mapsto e] \end{cases}
\end{aligned}
$$

Example: state

For any cpo S, the *state monad* is given by:

$$\begin{aligned} TA &= S \to (A \times S)_\perp \\ \eta\, a &= \lambda s.\, \uparrow(a, s) \\ f^* t &= \lambda s.\, (\lambda(a, s').\, f\, a\, s')^\dagger\, (t\, s) \end{aligned}$$

Let $r_1, \ldots, r_n$ be a finite set of *cell names*, and define TPL extensions

$$\frac{\Gamma \vdash E : \iota}{\Gamma \vdash r_i := E : \iota} \qquad \frac{}{\Gamma \vdash !r_i : \iota}$$

Take $S = \mathbf{N}^{\{1,\ldots,n\}}$. Then we can give semantics of the new constructs:

$$\begin{aligned} \mathcal{V}_T[\![r_i := E]\!]\,\rho &= (\lambda a.\, \lambda s.\, \uparrow(0, s[i \mapsto a]))^*\, (\mathcal{V}_T[\![E]\!]\,\rho) \\ \mathcal{V}_T[\![!r_i]\!]\,\rho &= \lambda s.\, \uparrow(s\, i, s) \end{aligned}$$

(We can also give meaning to cells of non-base type in this style, but that needs a reflexive domain S).

Example: dynamic scope

When D is any cpo, the *data monad* is given by

$$
\begin{aligned}
TA &= D \to A_\perp \\
\eta\, a &= \lambda d.\, {\uparrow} a \\
f^* t &= \lambda d.\, (\lambda a.\, f\, a\, d)^\dagger\, (t\, d)
\end{aligned}
$$

Let $v_1, \ldots, v_n$ be a finite set of *dynamic variables*. Then we can extend TPL by

$$
\frac{\Gamma \vdash E_1 : \iota \qquad \Gamma \vdash E_2 : \sigma}{\Gamma \vdash \text{with } v_i = E_1 \text{ do } E_2 : \sigma} \qquad \frac{}{\Gamma \vdash ?v_i : \iota}
$$

Taking $D = \mathbf{N}^{\{1,\ldots,n\}}$, we can give semantics of the new constructs:

$$
\begin{aligned}
\mathcal{V}_T[\![\text{with } v_i = E_1 \text{ do } E_2]\!]\, \rho \;&=\\
&(\lambda a.\, \lambda d.\, \mathcal{V}_T[\![E_2]\!]\, \rho\, d[i \mapsto a])^*\, (\mathcal{V}_T[\![E_1]\!]\, \rho) \\
\mathcal{V}_T[\![?v_i]\!]\, \rho \;&=\; \lambda d.\, {\uparrow}(d\, i)
\end{aligned}
$$

Note: commutative (unlike state). Ex: can check that $\mathcal{V}_T[\![(\lambda x.\, \lambda y.\, E)\, E_1\, E_2]\!] = \mathcal{V}_T[\![(\lambda y.\, \lambda x.\, E)\, E_2\, E_1]\!]$

Status

Many notions of “effectful functional programming” can be captured by a monadic semantics for a suitable monad.

But: the monad “lives” at the level of the semantics, not as a programmer-definable entity. Approaches:

- Pick some set of effects and stick to them (Scheme, ML): inflexible.
- Write everything in monadic style (Haskell): awkward.
- Use a monadic interpreter or pre-translation: inefficient; substantial effort for practical languages.

Two challenges:

- How can we add arbitrary user-defined effects to a standard CBV language, “as if” they were built in?
- How can we do it efficiently, to avoid paying the inclusion and extension cost at every computation step?

Effect-TPL

First step: keep more explicitly track of effect-nature of expressions. Keep expression syntax, but refine types:

$$
\begin{aligned}
e &::= \mathsf{n} \mid \mathsf{p} \\
\sigma &::= \iota \mid \sigma_1 \xrightarrow{e} \sigma_2
\end{aligned}
$$

(n is "no effect"; p is "partiality".)

Subeffect judgement, $e \le e'$:

$$
\frac{}{e \le e} \qquad \frac{e \le e' \qquad e' \le e''}{e \le e''} \qquad \frac{}{\mathsf{n} \le \mathsf{p}}
$$

Subtype judgement, $\sigma \le \sigma'$:

$$
\frac{}{\iota \le \iota} \qquad \frac{\sigma_1' \le \sigma_1 \qquad e \le e' \qquad \sigma_2 \le \sigma_2'}{\sigma_1 \xrightarrow{e} \sigma_2 \le \sigma_1' \xrightarrow{e'} \sigma_2'}
$$

(note that subtyping only affects annotations on the arrows.)

Effect-TPL typing II

Typing judgement: $\Gamma \vdash E : \sigma / e$:

$$\frac{\Gamma(x) = \sigma}{\Gamma \vdash x : \sigma / e} \qquad \frac{}{\Gamma \vdash \mathsf{z} : \iota / e} \qquad \frac{\Gamma \vdash E : \iota / e}{\Gamma \vdash \mathsf{s}\, E : \iota / e}$$

$$\frac{\Gamma \vdash E_1 : \iota / e \qquad \Gamma \vdash E_2 : \sigma / e \qquad \Gamma, x{:}\, \iota \vdash E_3 : \sigma / e}{\Gamma \vdash \mathsf{case}_{\sigma / e}(E_1, E_2, x.\, E_3) : \sigma / e}$$

$$\frac{\Gamma, x{:}\, \sigma_1 \vdash E : \sigma_2 / e'}{\Gamma \vdash \lambda x^{\,\sigma_1}.\, E : (\sigma_1 \xrightarrow{e'} \sigma_2) / e}$$

$$\frac{\Gamma, f{:}\, \sigma_1 \xrightarrow{e'} \sigma_2, x{:}\, \sigma_1 \vdash E : \sigma_2 / e'}{\Gamma \vdash \mathsf{fun}\ f(x^{\,\sigma_1})^{\,\sigma_2 / e'}.\, E : (\sigma_1 \xrightarrow{e'} \sigma_2) / e}\,(\mathsf{p} \le e')$$

$$\frac{\Gamma \vdash E_1 : (\sigma_1 \xrightarrow{e} \sigma_2) / e \qquad \Gamma \vdash E_2 : \sigma_1 / e}{\Gamma \vdash E_1\,(E_2) : \sigma_2 / e}$$

$$\frac{\Gamma \vdash E : \sigma / e \qquad \sigma \le \sigma' \qquad e \le e'}{\Gamma \vdash E : \sigma' / e'}$$

Note: typing derivations are not unique: can always take $e = \mathsf{p}$ everywhere, but often more precise typings are possible.

Denotational semantics of Effect-TPL

We assign meanings to terms based on their typing derivations. (Must check later that this is coherent.)

To every notion of effect e, associate a monad (computational if $\mathsf{p} \leq e$):

$$\begin{aligned} \mathcal{E}[\![\mathsf{n}]\!] &= \mathbf{Id} \\ \mathcal{E}[\![\mathsf{p}]\!] &= \mathbf{Lift} \end{aligned}$$

We write T_e, η_e, -^{*_e} for the components of $\mathcal{E}[\![e]\!]$.

To every type σ, a cpo $\mathcal{E}[\![\sigma]\!]$:

$$\begin{aligned} \mathcal{E}[\![\iota]\!] &= \mathbf{N} \\ \mathcal{E}[\![\sigma_1 \xrightarrow{e} \sigma_2]\!] &= \mathcal{E}[\![\sigma_1]\!] \to T_e\mathcal{E}[\![\sigma_2]\!] \end{aligned}$$

To every subeffecting derivation $e \leq e'$ and cpo A, a function $\mathcal{E}[\![e \leq e']\!]_A : T_e A \to T_{e'} A$:

$$\begin{aligned} \mathcal{E}[\![e \leq e]\!] &= \mathsf{id}_{T_e A} \\ \mathcal{E}[\![e \leq e'']\!] &= \mathcal{E}[\![e' \leq e'']\!] \circ \mathcal{E}[\![e \leq e']\!] \\ \mathcal{E}[\![\mathsf{n} \leq \mathsf{p}]\!] &= \uparrow_A \end{aligned}$$

(actually, a *monad morphism* from $\mathcal{E}[\![e]\!]$ to $\mathcal{E}[\![e']\!]$.)

DS of Effect-TPL II

To every subtyping derivation $\sigma \leq \sigma'$, a continuous *coercion* function $\mathcal{E}[\![\sigma \leq \sigma']\!] : \mathcal{E}[\![\sigma]\!] \to \mathcal{E}[\![\sigma']\!]$:

$$\mathcal{E}[\![\iota \leq \iota]\!] = \lambda n.\, n$$

$$\mathcal{E}[\![\sigma_1 \xrightarrow{e} \sigma_2 \leq \sigma_1' \xrightarrow{e'} \sigma_2']\!] = \lambda f.\, \mathcal{E}[\![\sigma_2/e \leq \sigma_2'/e']\!] \circ f \circ \mathcal{E}[\![\sigma_1' \leq \sigma_1]\!]$$

where

$$\mathcal{E}[\![\sigma/e \leq \sigma'/e']\!] = (\eta_{e'} \circ \mathcal{E}[\![\sigma \leq \sigma']\!])^{*e'} \circ \mathcal{E}[\![e \leq e']\!]$$

To every typing derivation $\Gamma \vdash E : \sigma/e$ a continuous function $\mathcal{E}[\![E]\!]^e : \mathcal{E}[\![\Gamma]\!] \to T_e\mathcal{E}[\![\sigma]\!]$:

$$\begin{aligned}
\mathcal{E}[\![x]\!]^e\rho &= \eta_e\,(\rho\,x) \\
\mathcal{E}[\![\mathsf{z}]\!]^e\rho &= \eta_e\,0 \\
\mathcal{E}[\![\mathsf{s}\,E]\!]^e\rho &= (\lambda n.\,\eta_e\,(n+1))^{*e}\,(\mathcal{E}[\![E]\!]^e\,\rho) \\
\mathcal{E}[\![\mathsf{case}(E_1, E_2, x.\,E_3)]\!]^e\rho &= \phi^{*e}\,(\mathcal{E}[\![E_1]\!]^e\rho) \text{ where } \phi = \cdots \\
\mathcal{E}[\![\lambda x.\,E]\!]^e\rho &= \eta_e\,(\lambda a.\,\mathcal{E}[\![E]\!]^{e'}\rho[x \mapsto a]) \\
\mathcal{E}[\![\mathsf{fun}\ f(x).\,E]\!]^e\rho &= \eta_e\,(\mathsf{fix}\,(\lambda\phi.\,\lambda a.\,\mathcal{E}[\![E]\!]^{e'}\rho[\cdots])) \\
\mathcal{E}[\![E_1\,(E_2)]\!]^e\rho &= (\lambda\phi.\,\phi^{*e}\,(\mathcal{E}[\![E_2]\!]^e\,\rho))^{*e}\,(\mathcal{E}[\![E_1]\!]^e\,\rho) \\
\mathcal{E}[\![E]\!]^{e'}\rho &= \mathcal{E}[\![\sigma/e \leq \sigma'/e']\!]\,(\mathcal{E}[\![E]\!]^e\rho)
\end{aligned}$$

Defining monads in Effect-TPL

Recall that the monadic semantics required special-purpose semantic equations for monad-specific operations.

Want a *uniform* notation for all effects expressible in a given monadic semantics.

Definition. *We say that a monad* $\mathbf{T} = (T, \eta, \text{-}^*)$ *is* definable *in Effect-TPL if there exists a type constructor comp*(-) *and term families*

$$\begin{aligned} \mathit{unit}_\sigma &: \sigma \xrightarrow{\mathsf{n}} \mathit{comp}(\sigma) \\ \mathit{ext}_{\sigma_1,\sigma_2} &: (\sigma_1 \xrightarrow{\mathsf{n}} \mathit{comp}(\sigma_2)) \xrightarrow{\mathsf{n}} (\mathit{comp}(\sigma_1) \xrightarrow{\mathsf{n}} \mathit{comp}(\sigma_2)) \end{aligned}$$

such that

$$\begin{aligned} \mathcal{E}[\![\mathit{comp}(\sigma)]\!] &= T\mathcal{E}[\![\sigma]\!] \\ \mathcal{E}[\![\mathit{unit}_\sigma]\!]^{\mathsf{n}}\rho &= \eta_{\mathcal{E}[\![\sigma]\!]} \\ \mathcal{E}[\![\mathit{ext}_{\sigma_1,\sigma_2}]\!]^{\mathsf{n}}\rho &= \lambda f^{\,\mathcal{E}[\![\sigma_1]\!] \to T\mathcal{E}[\![\sigma_2]\!]}.\, f^* \end{aligned}$$

(We will assume that Effect-TPL has been extended with product and sum types, with the straightforward semantics.)

Adding effects: Monadic reflection

Let $\mathbf{T}$ be a definable computational monad. Then we can extend Effect-TPL to get $\mathbf{T}$-Effect-TPL.

New effect:

$$e ::= \cdots \mid \mathsf{t}$$

New terms:

$$E ::= \cdots \mid \mu(E) \mid [E]$$

New subeffect and typing rules:

$$\frac{}{\mathsf{p} \leq \mathsf{t}}$$

$$\frac{\Gamma \vdash E : \mathit{comp}(\sigma)/\mathsf{n}}{\Gamma \vdash \mu(E) : \sigma/\mathsf{t}} \qquad \frac{\Gamma \vdash E : \sigma/\mathsf{t}}{\Gamma \vdash [E] : \mathit{comp}(\sigma)/\mathsf{n}}$$

$\mu(E)$ *reflects* the data representation E into an "active" effect.

$[E]$ *reifies* the effectful expression E into a "passive" datum.

Semantics of $\mathbb{T}$-Effect-TPL

Of the new effect:

$$\mathcal{E}[\![\mathsf{t}]\!] = (T, \eta, \text{-}^*)$$

Of the new subeffect rule:

$$\mathcal{E}[\![\mathsf{p} \leq \mathsf{t}]\!] = \lambda p.\, (\lambda a.\, \eta\, a)^\dagger\, p$$

Of the new operations:

$$\begin{aligned} \mathcal{E}[\![\mu(E)]\!]^{\mathsf{t}}\rho &= \mathcal{E}[\![E]\!]^{\mathsf{n}}\rho \\ \mathcal{E}[\![\texttt{[}E\texttt{]}]\!]^{\mathsf{n}}\rho &= \mathcal{E}[\![E]\!]^{\mathsf{t}}\rho \end{aligned}$$

(Remember that $\mathcal{E}[\![\sigma/\mathsf{t}]\!]$ and $\mathcal{E}[\![\mathit{comp}(\sigma)/\mathsf{n}]\!]$ are the same cpo.)

In particular, $\texttt{[}\mu(E)\texttt{]} = E$ and $\mu(\texttt{[}E\texttt{]}) = E$ are valid source-level equivalences.

(Later, will consider a different semantics for the language, where reflection and reification have more interesting denotations.)

Example: exceptions II

The monad:

$$\begin{aligned} \mathit{comp}(\sigma) &= 1 \xrightarrow{\mathsf{p}} (\sigma + \sigma_{exn}) \\ \mathit{unit}\ a &= \lambda().\ \mathsf{inl}\ a \\ \mathit{ext}\ f\ t &= \lambda().\ \mathsf{scase}(t\,(), a.\ f\ a, e.\ \mathsf{inr}\ e) \end{aligned}$$

(In practice, can omit the thunkification, because f^* must be strict; include it here for uniformity.)

Operation definitions:

$$\begin{aligned} \mathsf{raise}\ E &\equiv (\lambda e.\, \mu(\lambda().\ \mathsf{inr}\ e))\, E \\ E_1\ \mathsf{handle}\ x \Rightarrow E_2 &\equiv \mathsf{scase}([E_1]\,(), a.\, a, x.\, E_2) \end{aligned}$$

Example: state II

Let υ be a type of variable names (e.g., strings). Take $\sigma_{\mathsf{st}} = \upsilon \xrightarrow{\mathsf{n}} \iota$, and define the functions

$$\begin{aligned} \textit{lookup} &: \sigma_{\mathsf{st}} \times \upsilon \xrightarrow{\mathsf{n}} \iota \\ \textit{update} &: \sigma_{\mathsf{st}} \times \upsilon \times \iota \xrightarrow{\mathsf{n}} \sigma_{\mathsf{st}} \end{aligned}$$

in the straightforward way.

The monad:

$$\begin{aligned} \textit{comp}(\sigma) &= \sigma_{\mathsf{st}} \xrightarrow{\mathsf{p}} (\sigma \times \sigma_{\mathsf{st}}) \\ \textit{unit}\; a &= \lambda s.\, (a, s) \\ \textit{ext}\; f\, t &= \lambda s.\, (\lambda(a, s').\, f\, a\, s')\, (t\, s) \end{aligned}$$

The operations:

$$\begin{aligned} v := E &\equiv (\lambda a.\, \mu(\lambda s.\, (\mathsf{z}, \textit{update}(s, v, a))))\, E \\ !v &\equiv \mu(\lambda s.\, (\textit{lookup}(s, v), s)) \\ \mathbf{run}\, E &\equiv [E]\, (\lambda v.\, \mathsf{z}) \end{aligned}$$

Example: dynamic scope II

Monad:

$$\begin{aligned} \textit{comp}(\sigma) &= (\upsilon \xrightarrow{\mathsf{n}} \iota) \xrightarrow{\mathsf{p}} \sigma \\ \textit{unit}\, a &= \lambda d.\, a \\ \textit{ext}\, f\, t &= \lambda d.\, f\,(t\, d)\, d \end{aligned}$$

Operations:

$$\begin{aligned} &\mathsf{with}\ v = E_1\ \mathsf{do}\ E_2 \ \equiv \\ &\quad (\lambda a.\, \mu(\lambda d.\, [E_2]\,(\textit{update}\,(d, v, a))))\, E_1 \\ &\qquad\qquad ?v \ \equiv \ \mu(\lambda d.\, \textit{lookup}\,(d, v)) \end{aligned}$$

Implementing a monadic semantics

Main goals of a formal semantics:

- To completely and unambiguously specify the result of running any well-formed program.
- To facilitate manual and automated reasoning about programs and program fragments.
- Optionally: to serve as the basis of an implementation. (e.g.. the ML Kit)

Unfortunately, the latter two goals are often in conflict: a "naive" realization of a clear and modular specification is often prohibitively inefficient.

Ex: simple semantics of exceptions (operational or denotational) specifies a time-consuming tag-check at every computation step. Even programs which rarely use exceptions are paying a substantial price to support them.

Idea: shift the cost to the actual effect-causing operations.

Changing representations

Let O be a "sufficiently large" cpo, so that for any A of interest there exists functions $\mathsf{in}_A : A \to O$ and $\mathsf{out}_A : O \to A_\perp$, with $\mathsf{out}_A \circ \mathsf{in}_A = \uparrow$.

Let $\mathbf{T} = (T, \eta, \text{-}^*)$ be any monad, and $\mathbf{U}$ be the continuation monad with TO-answers, i.e., $UA = (A \to TO) \to TO$. Then $i_A : TA \to UA$ given by

$$i_A\, t = \lambda\kappa.\, \kappa^*\, t$$

(actually, a monad morphism from $\mathbf{T}$ to $\mathbf{U}$) encodes T-computations as U-computations.

Ex: Let $\mathbf{T}$ be the exception monad, $TA = (A+X)_\perp$:

$$\begin{aligned} i\,\perp &= \lambda\kappa.\,\perp \\ i\,(\uparrow(1,a)) &= \lambda\kappa.\,\kappa\,a \\ i\,(\uparrow(2,e)) &= \lambda\kappa.\uparrow(2,e) \end{aligned}$$

Moreover, the function $j_A : UA \to TA$ given by

$$j_A\, u = (\eta_A^\dagger \circ \mathsf{out}_A)^*\,(u\,(\eta_O \circ \mathsf{in}_A))$$

recovers T-representations from U-representations.

Monads and continuations

More formally:

Definition. *Let $\mathbf{T}$ be a monad, and take $\mathbf{K}$ to be the continuation monad with answer type TO. Then we define the variant $\mathbf{T}$-Effect-TPL semantics $\mathcal{E}'_T[\![\text{-}]\!]$ as the $\mathbf{K}$-Effect semantics, i.e.,*

$$\mathcal{E}'_T[\![\mathsf{t}]\!] = \mathbf{K}$$

extended with

$$\begin{aligned}\mathcal{E}'_T[\![\mu^T(E)]\!]^{\mathsf{t}}\rho &= \lambda\kappa.\,\kappa^*\,(\mathcal{E}'_T[\![E]\!]^{\mathsf{n}}\,\rho)\\ \mathcal{E}'_T[\![[E]^T]\!]^{\mathsf{n}}\rho &= (\eta^\dagger \circ \mathsf{out})^*\,(\mathcal{E}'_T[\![E]\!]^{\mathsf{t}}\,\rho\,(\eta \circ \mathsf{in}))\end{aligned}$$

Theorem. *For a complete program $\cdot \vdash E : \iota/\mathsf{p}$, $\mathcal{E}_T[\![E]\!]^{\mathsf{p}}\bullet =_{\mathbf{N}_\perp} \mathcal{E}'_T[\![E]\!]^{\mathsf{p}}\bullet$.*

Consequence: can embed $\mathbf{T}$-Effect-TPL in $\mathbf{K}$-Effect-TPL by defining:

$$\begin{aligned}\mu^T(E) &\equiv \mu^K(\lambda k.\,\mathit{ext}\,k\,E)\\ [E]^T &\equiv \mathit{ext}\,(\mathit{unit} \circ \mathsf{out})\,([E]^K\,(\mathit{unit} \circ \mathsf{in}))\end{aligned}$$

(Recall that $\mathbf{K}$-semantics of $\mu^K(\text{-})$ and $[\text{-}]^K$ is trivial.)

Summary of the simulation

Three major steps:

- Show that the any monadic effect `t` can be simulated by the *universal* effect k (general continuations).

  Consequence: only need to implement a *fixed* effect, others definable.

- Show that all three effects (n, p, k) can be simulated by a single effect cs of *control and state*.

  Consequence: don't need the effect-annotations on arrows for program execution; all correct typings give same result.

- Show that the necessary cs-operations can be expressed in terms of standard `call/cc` and `ref` operations.

  Consequence: we can do it in SML/NJ!

Proofs mainly by (semantic) logical relations.

What do we have?

A model for programming with first-class monadic effects:

- Programmer defines triple $(\mathit{comp}(\text{-}), \mathit{unit}, \mathit{ext})$ (parameterized type, two polymorphic functions.)
- System supplies $\mu(\text{-})/\texttt{[-]}$ for that triple.
- *Contract:* if the supplied triple satisfies the monad laws, then the newly-defined $\mu(\text{-})/\texttt{[-]}$ correctly implement monadic reflection and reification.
- Programmer can use standard, monadic semantics for reasoning about programs. E.g., exploit commutativity of the dynamic-binding monad.
- System can use variant, continuation/state-based semantics to harness efficient implementations of call/cc and state.

Representing control in SML/NJ

```
signature CONTROL =
  sig
    type ans
    val reflect : ((’a -> ans) -> ans) -> ’a
    val reify : (unit -> ’a) -> (’a -> ans) -> ans
  end;

functor Control (type ans) : CONTROL =
  struct
    open SMLofNJ (* ’a cont, callcc, throw *)

    val mc : ans cont ref =
        ref (callcc (fn c =>
               (callcc (fn c’ => throw c c’);
                raise Fail "impossible")))
    fun abort x = throw (!mc) x
    fun reset t =
        let val m = !mc
            val r = callcc (fn c =>
                      (mc := c; abort (t ())))
        in mc := m; r end

    type ans = ans
    fun reflect h =
        callcc (fn c =>
          abort (h (fn v => reset (fn () =>
                                throw c v))))
    fun reify t k = reset (fn () => k (t ()))
  end;
```

Representing monadic effects

Simulating monad reflection with control operators:

```
signature MONAD =
  sig
    type 'a T
    val unit : 'a -> 'a T
    val ext : ('a -> 'b T) -> 'a T -> 'b T
  end;

signature RMONAD =
  sig
    structure M : MONAD
    val reflect : 'a M.T -> 'a
    val reify : (unit -> 'a) -> 'a M.T
  end;

functor Represent (M : MONAD) : RMONAD =
  struct
    structure CR = Control (type ans = exn M.T)

    structure M = M
    fun reflect t = CR.reflect (fn k => M.ext k t)
    fun reify (m : unit -> 'a) =
        let exception In of 'a
        in M.ext (fn (In x) => M.unit x)
             (CR.reify m (M.unit o In))
        end
  end;
```

Example: nondeterminism

```
structure ListMonad : MONAD =
  struct
    type 'a T = 'a list
    fun unit x = [x]
    fun ext f [] = []
      | ext f (h::t) = f h @ ext f t
  end;

structure ListRep = Represent (ListMonad);

val pick = ListRep.reflect
val results = ListRep.reify;
> val pick = fn : 'a list -> 'a
> val results = fn : (unit -> 'a) -> 'a list

results (fn ()=>let val n = pick [2,3,5]
                    val s = pick ["a","b"]
                in (makestring n) ^ s end);
> val it = ["2a","2b","3a","3b","5a","5b"]
```

(In fact, monadic reflection strictly generalizes "monad comprehensions".)

Compactness of implementation somewhat obscures its signficicance: can reuse both the *code* and its *correctness proof* for a whole spectrum of effects.

Summary

Practical approach to programming with monads in ML-like languages:

- Convenient: can write programs in direct style (but integrates naturally with explicitly effect-passing fragments).
- Embeddable: no translation/interpretation phase; inherits resources of host language (standard library, type checker, modules, etc.)
- Declarative: precise characterization of effects; allows, e.g., case-based reasoning about exceptions.
- Efficient: programs run at full native speed, even for complex monads. [Modulo fixed (and small) overhead of supporting call/cc.]

Conclusions & future work

Some (perhaps) surprising conclusions:

- When properly integrated, monads are just as useful in ML-like languages as in Haskell-like ones.

- Full first-class continuations ("the goto of functional programming") allow us to program *more* declaratively.

Things to look at:

- Extend proofs to recursive types in source language (esp. for specifying recursive monads, such as higher-order state). No big surprises expected.

- Layering effects: define multi-effect language by hierarchy of monads; simulate with call/cc + one meta^n-continuation per level. [Code exists and seems to work; sketches of proofs]

- Practical effect-type systems (for documentation and static checking purposes); integration with polymorphism.