Handlers in Action

Ohad Kammar
<ohad.kammar@ed.ac.uk>

SPLS
March 15
joint with Sam Lindley and Nicolas Oury

Your
Name Here

Programming with effects

ML-style effects

Implicit effects

```
# let f = fun x -> let y = ! loc in
                   loc := x + y;;
val f : int -> unit = <fun>

# let g = fun i -> if (i < 0)
                   then raise (Failure "neg.")
                   else 0;;
val g : int -> int = <fun>
```

Trivially combined

```
# let h = f(7); g(1);;
val h : int = 0
```

ML-style effects

Controlling effects: effect systems

```
                    {lookup, update}
val f : int ---------------------------> unit = <fun>

                    {raise}
val g : int ----------------> int = <fun>

val h : int ! {lookup, update, raise}
```

Rigid effects

Fixed collection of effects.

Haskell effects

Explicit effects

```
f :: Int -> State Int ()
f = \ x ->  do { y <- get;
                 put (x+y)  }

g :: Int -> ErrorT String Identity Int
g = \ i ->  if (i < 0)
            then throwError "neg."
            else return 0
```

Haskell effects

Sophisticated combination

```
f :: Int -> State Int ()
f = \ x ->  do { y <- get;
                 put (x+y)  }

g :: Monad m => Int -> ErrorT String m Int
g = \ i ->  if (i < 0)
            then throwError "neg."
            else return 0

h :: (ErrorT String (State Int)) Int
h = do { lift (f 7);
         g 0            }
```

Haskell effects

Controlling effects: Monads

```
f :: Int -> State Int ()

g :: Monad m => Int -> ErrorT String m Int

h :: (ErrorT String (State Int)) Int
```

Manual/semi-inferred, intrinsic.

Fluid effects

User defined effects.

Semantic synthesis

Layered monads and effect reification

Monads in action, Filinski, POPL'10.

Eff

Programming with algebraic effects and handlers, Bauer and Pretnar, arXiv draft, 2012.

Idea

- Operational semantics for an Eff variant.
- Inspired by Filinski.

Contribution

The λ_{eff}-calculus:

- Implicit effects.
- Trivially combined.
- Fluidity.

Non-contribution

Controlling effects: type system, effect system.

Outline

- Running example: pipes.
- λ_{eff}-calculus
- Pipes: implementation and execution.
- Summary and further work.

Running example: pipes

```
M1 = x ← random(2);
     if (x = 1)
     then put('y')
     else put('n');
     put('\n')

M2 = s ← readln;
     if (s = "y")
     then set(loc, 10);

M1 | M2
```

No concurrency!

λ_{eff}-calculus

Syntax

$$
\begin{aligned}
V &::= \mathsf{x} \mid * \mid \textbf{thunk}\ M \\
M, N &::= \textbf{return}\ V \mid \mathsf{x} \leftarrow M_1 ; M_2 \mid \textbf{force}\ V \mid \\
&\quad\ \lambda \mathsf{x} . M \mid M\ V \mid op(V)(\lambda \mathsf{x} . M) \mid \\
&\quad\ \textbf{whatever}\ V \mid \textbf{handle}\ M\ \textbf{with}\ H \\
H &::= \textbf{handler}\ \{ \textbf{return} : M_{\text{ret}} \\
&\qquad\qquad\qquad op_1 : N_1 \\
&\qquad\qquad\qquad \ldots \\
&\qquad\qquad\qquad op_n : N_n \ \}
\end{aligned}
$$

For example,	abbreviated as
$output(\text{'a'})(\lambda\_ . \textbf{return}\ *)$	$put(\text{'a'})$
$input(*)(\lambda \mathsf{c} . \textbf{return}\ \mathsf{c})$	get
$raise(\text{"neg."})(\lambda \mathsf{z} . \textbf{whatever}\ \mathsf{z})$	$raise(\text{"neg."})$

Handlers

Base case:

$$\texttt{handle return 'a'}$$
$$\texttt{with handler } \{\texttt{return}: \lambda \texttt{c}.\, put(\texttt{c})\}$$
$$\longrightarrow (\lambda \texttt{c}.\, put(\texttt{c}))\ \texttt{'a'} \longrightarrow put(\texttt{'a'})$$

Generally:

$$\texttt{handle return } V \texttt{ with handler } \{\texttt{return}: N_{\texttt{ret}} \cdots\}$$
$$\longrightarrow N_{\texttt{ret}}\ V$$

Handlers

Operations

```
H := handler {return: λx.return x
              input : λ_.λk.(force k) 'f'}

handle input(*)(λc.return c) with H
⟶(λ_.λk.(force k) 'f')
    * (thunk λc. handle return c with H)
⟶(λk.(force k) 'f')
    (thunk λc. handle return c with H)
⟶(force (thunk (λc. handle return c with H)))
    'f'
⟶handle return 'f' with H
⟶return 'f'
```

Handlers

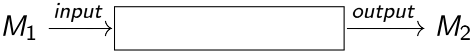
Operations

Generally, given:

$$H := \textbf{handler}\ \{\ \textbf{return} : M_{\mathrm{ret}}\quad op_1 : N_1\quad \ldots\quad op_n : N_n\ \}$$

$$\textbf{handle}\ \ op_i(V)(\lambda x.M)\ \ \textbf{with}\ \ H$$
$$\longrightarrow N_i\ \ V\ \ (\textbf{thunk}\ \ \lambda x.\textbf{handle}\ \ M\ \ \textbf{with}\ \ H)$$

Pipes



Idea

- Parametrise M_2 by a computation **producing** characters, and
- parametrise M_1 by a computation **consuming** characters,

without touching M_1, M_2.

Pipes

$$H_2 := \textbf{handler}\ \{\textbf{return}: \lambda x.\lambda prod.\ \textbf{return}\ x$$
$$\quad\quad \textit{input}: \lambda\_.\lambda k.\lambda prod.$$
$$\quad\quad\quad (\textbf{force}\ prod) * k\}$$

$$H_1 := \textbf{handler}\ \{\textbf{return}: \lambda z.\textbf{whatever}\ z$$
$$\quad\quad \textit{output}: \lambda c.\lambda k.\lambda cons.$$
$$\quad\quad\quad (\textbf{force}\ cons)\ c\ k\}$$

$$M_1 \mid M_2 := \textbf{handle}\ M_2\ \textbf{with}\ H_2$$
$$\quad (\textbf{thunk}\ \lambda\_.\textbf{handle}\ M_1\ \textbf{with}\ H_1)$$

Run $M_1 \mid M_2$ for:

$$M_1 := \textit{output}('a')(\lambda\_.M_1')$$
$$M_2 := \textit{input}(*)(\lambda c.M_2'\ c)$$

Execute

```
M1 | M2 =
handle input (*)(λc.M2' c)
with handler {return: λx.λprod. return x
              input : λ_.λk.λprod.
                        (force prod) * k}
  (thunk λ_.handle M1 with H1)
⟶
(λ_.λk.λprod.(force prod) * k})
  *
  (thunk λc.handle M2' c with H2)
  (thunk λ_.handle M1 with H1)
```

Execute

$$\longrightarrow^3$$
$$(\textbf{force}\ (\textbf{thunk}\ \lambda\_.\textbf{handle}\ M_1\ \textbf{with}\ H_1))$$
$$*$$
$$(\textbf{thunk}\ (\lambda c.\textbf{thunk}\ \textbf{handle}\ M_2'\ c\ \textbf{with}\ H_2))$$
$$\longrightarrow$$
$$(\lambda\_.\textbf{handle}\ M_1\ \textbf{with}\ H_1)$$
$$*$$
$$(\textbf{thunk}\ (\lambda c.\textbf{thunk}\ \textbf{handle}\ M_2'\ c\ \textbf{with}\ H_2))$$
$$\longrightarrow$$
$$\textbf{handle}\ M_1\ \textbf{with}\ H_1$$
$$(\textbf{thunk}\ (\lambda c.\textbf{handle}\ M_2'\ c\ \textbf{with}\ H_2))$$

Execute

$=$

```
handle output('a')(λ_.M'_1)
with handler {return: λz.whatever z
              output: λc.λk.λcons.
                        (force cons) c k}
(thunk (λc.handle M'_2 c with H_2))
⟶
(λc.λk.λcons.(force cons) c k)
'a'
(thunk λ_.handle M'_1 with H_1)
(thunk (λc.handle M'_2 c with H_2))
```

Execute

$\longrightarrow^3$

$(\textbf{force}\ (\textbf{thunk}\ (\lambda \mathtt{c} . \textbf{handle}\ M_2'\ \mathtt{c}\ \textbf{with}\ H_2)))$
$\quad \mathtt{'a'}$
$\quad (\textbf{thunk}\ \lambda \_ . \textbf{handle}\ M_1'\ \textbf{with}\ H_1)$

$\longrightarrow$

$(\lambda \mathtt{c} . \textbf{handle}\ M_2'\ \mathtt{c}\ \textbf{with}\ H_2)$
$\quad \mathtt{'a'}$
$\quad (\textbf{thunk}\ \lambda \_ . \textbf{handle}\ M_1'\ \textbf{with}\ H_1)$

$\longrightarrow$

$\textbf{handle}\ M_2'\ \mathtt{'a'}\ \textbf{with}\ H_2$
$\quad (\textbf{thunk}\ \lambda \_ . \textbf{handle}\ M_1'\ \textbf{with}\ H_1)$

$=$

$M_1' \mid (M_2'\ \mathtt{'a'})$

Zoom out

Expected behaviour

$$output(\texttt{'a'})(\lambda\_.\, M_1') \mid (input\ (\ *\ )(\lambda c.\, M_2'\ c)$$
$$\longrightarrow^*$$
$$M_1' \mid (M_2'\ \texttt{'a'})$$

Unhandled effects

$$
\begin{aligned}
H := \ \textbf{handler}\ \{\ & \textbf{return} && : M_{\mathrm{ret}} \\
& op_1 && : N_1 \\
& \ldots && \\
& op_n && : N_n\ \}
\end{aligned}
$$

If $\boxed{op} \notin \{op_1, \ldots, op_n\}$ then:

$$
\begin{aligned}
& \textbf{handle}\ \boxed{op}(V)(\lambda x.M)\ \textbf{with}\ H \\
& \quad \longrightarrow \boxed{op}(V)(\lambda x.\textbf{handle}\ M\ \textbf{with}\ H)
\end{aligned}
$$

i.e., implicit handling:

$$
\begin{aligned}
\textbf{handler}\ \{ & \ldots \\
& \boxed{op}\ :\ \lambda p.\lambda k.\boxed{op}(p)(\lambda x.(\textbf{force}\ k)\ x) \\
& \ldots \}
\end{aligned}
$$

Further work

- Type system, type inference.
- Effect reification.
- Implement as Haskell, ML, Scheme libraries?
- What is the delta? (e.g., delimited continuations? proof obligations?)

Idea

- Operational semantics for an Eff variant.
- Inspired by Filinski.

Contribution

The λ_{eff}-calculus:

- Implicit effects.
- Trivially combined.
- Fluidity.

Non-contribution

Controlling effects: type system, effect system.