Abstracting Algebraic Effects

Maciej Piróg

(joint work with Dariusz Biernacki, Piotr Polesiuk, and Filip Sieczkowski)

`bitbucket.org/pl-uwr/helium`

Abstractions for programming with algebraic effects and handlers

- Local effects
- Existential effects

Local Effects

Counting the number of times `f` uses its argument:

```
effect Tick = { tick : Unit => Unit }

...

val f : (T1 ->[r] T2) ->[r] Unit = ...

let cnt_f g =
  handle f (fn x => tick (); g x) with
  | tick ()  => fn n => resume () (n+1)
  | return _ => fn n => n
  end 0
```

But what if `g` uses `Tick` as its effect? We're in trouble!

Local Effects

```
val f : (T1 ->[r] T2) ->[r] Unit = ...

let cnt_f g =
  effect Tick = { tick : Unit => Unit } in
  handle f (fn x => tick (); g x) with
  | tick () => fn n => resume () (n+1)
  | return _ => fn n => n
  end 0
```

Now `Tick` is local, which means that `g` cannot know about it. We're safe!

Existential Effects / ML-style Module System

A signature based on SML/OCaml's UREF:

```
type Set : type -> type

effect UF : type -> effect

val new    : a ->[UF a] Set a
val find   : Set a ->[UF a] a
val union  : (a -> a ->[r] a) -> Set a -> Set a ->[UF a, r] Unit
val withUF : (Unit ->[UF a, r] b) ->[RE, r] b
```

Trouble is Coming

Slogan: Use whatever operations you want, the handler will know its own:

```
effect Reader s = { ask : Unit => s }
effect State  s = { get : Unit => s
                  ; put : s    => Unit }

handle ask () + get () with
  | ask ()   => resume 1
end
```

...the expression has the type `Int / [State Int]`

```
(* signature *)
effect E
val my_ask : Unit ->[E] Int
val my_handle : (Unit ->[E,r] a) ->[r] a

(* module M *)
effect E = Reader Int
let my_ask = ask
let my_handle t = handle t () with
 | ask () => resume 1
end

(* user code *)
handle
  handle ask () + M.my_ask () with
  | ask () => resume 5
  end
with M.my_handle
```

If we simply erase the type and module information, how will the handler know its own?!?

A glimpse at the type system:

Kinds and Types

$$\begin{aligned}\text{Kind} \ni \kappa &::= \mathsf{T} \mid \mathsf{E} \mid \mathsf{R} \mid \kappa \to \kappa \\ \text{Typelike} \ni \sigma, \tau, \varepsilon, \rho &::= \alpha \mid \tau\ \tau \mid \tau \to_\rho \tau \mid \forall\, \alpha :: \kappa.\ \tau \mid \exists\, \alpha :: \kappa.\ \tau \mid \langle\rangle \mid \langle \varepsilon | \rho \rangle\end{aligned}$$

Judgements

$$\Delta, \Gamma \vdash e : \tau / \rho$$

Typing Rule for Local Effects

$$\frac{\Delta, \alpha = \theta \vdash \theta \qquad \Delta, \alpha = \theta; \Gamma \vdash e : \tau \;/\; \rho \qquad \Delta \vdash \tau :: \mathsf{T} \qquad \Delta \vdash \rho :: \mathsf{R}}{\Delta; \Gamma \vdash \textbf{effect}\ \alpha = \theta\ \textbf{in}\ e : \tau \;/\; \rho}$$

Note that τ and ρ cannot mention α

Typing Rules for Existentials

$$\frac{\Delta \vdash \sigma :: \kappa \qquad \Delta; \Gamma \vdash e : \tau\{\sigma / \alpha\} \; / \; \rho}{\Delta; \Gamma \vdash \textbf{pack}(\sigma, e) \; \textbf{as} \; \exists \alpha :: \kappa.\, \tau : \exists \alpha :: \kappa.\, \tau \; / \; \rho}$$

$$\frac{\Delta; \Gamma \vdash e_1 : \exists \alpha :: \kappa.\, \sigma \; / \; \rho \qquad \Delta, \alpha :: \kappa; \Gamma, x : \sigma \vdash e_2 : \tau \; / \; \rho \qquad \Delta \vdash \tau :: \mathsf{T}}{\Delta; \Gamma \vdash \textbf{unpack} \; e_1 \; \textbf{as} \; \alpha :: \kappa \textbf{,} \; x : \sigma \; \textbf{in} \; e_2 : \tau \; / \; \rho}$$

But these are the usual rules for existentials! So what is going on?

Typing Rule for Handlers

$$\frac{\dfrac{\alpha = \overline{\beta :: \kappa}.\ \{\overline{\delta}\} \in \Delta \qquad \overline{\Delta \vdash \sigma :: \kappa}}{\Delta; \Gamma; \delta\{\overline{\sigma} / \overline{\beta}\} \vdash h : \tau_r \ / \ \rho} \qquad \Delta; \Gamma \vdash e : \tau_a \ / \ \langle \alpha\ \overline{\sigma} | \rho \rangle \qquad \Delta; \Gamma, x : \tau_a \vdash e_r : \tau_r \ / \ \rho}{\Delta; \Gamma \vdash \textbf{handle}_{\alpha\ \overline{\sigma}}\ e\ \{\overline{h}; \textbf{return}\ x : \tau_a \Rightarrow e_r\} : \tau_r \ / \ \rho}$$

$$\frac{\Delta \vdash \varepsilon_1 \# \varepsilon_2}{\Delta \vdash \langle \varepsilon_1, \varepsilon_2 | \rho \rangle \simeq \langle \varepsilon_2, \varepsilon_1 | \rho \rangle :: \mathsf{R}}$$

Note that:

- You can handle only known effects (obviously!)
- You cannot freely swap variables in rows (only known effects)

This Means That...

Given an expression

$$e : \tau \ / \ \langle \alpha, \text{Reader int} \rangle,$$

the expression

$$\textbf{handle}_{\text{Reader int}} \ e \ \{\ldots \ \textbf{;} \ \ldots\}$$

simply won't type-check, because we cannot guarantee that $\alpha \neq$ Reader int

Our solution is to use...

Explicit Coercions in the Core Language

$$\frac{\Delta;\Gamma \vdash e : \tau \;/\; \rho \qquad \Delta \vdash c : \rho \triangleright \rho'}{\Delta;\Gamma \vdash \langle c \rangle\, e : \tau \;/\; \rho'}$$

$$\frac{\Delta \vdash \varepsilon :: \mathsf{E}}{\Delta \vdash \uparrow\!\varepsilon : \rho \triangleright \langle \varepsilon | \rho \rangle} \qquad \frac{}{\Delta \vdash \varepsilon_1 \leftrightarrow \varepsilon_2 : \langle \varepsilon_1, \varepsilon_2 | \rho \rangle \triangleright \langle \varepsilon_2, \varepsilon_1 | \rho \rangle}$$

$$\frac{\Delta \vdash c : \rho \triangleright \rho'}{\Delta \vdash \varepsilon : c : \langle \varepsilon | \rho \rangle \triangleright \langle \varepsilon | \rho' \rangle}$$

$$\frac{\Delta \vdash c_1 : \rho_1 \triangleright \rho_2 \qquad \Delta \vdash \rho_1 \simeq \rho_2 :: \mathsf{R} \qquad \Delta \vdash c_2 : \rho_2 \triangleright \rho_3}{\Delta \vdash c_1 \cdot c_2 : \rho_1 \triangleright \rho_3}$$

Conclusions

Implementation: a simple language with ML-style module system built on top of existentials (`https://bitbucket.org/pl-uwr/helium`). The programmer doesn't see the coercions; they are introduced by the compiler during type-checking, before the types are erased.

Rows don't give us anything for free now.

Open question: What is the right programmer-level interface to manage effects.