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:

\[
\text{effect } \text{Tick} = \{ \text{tick} : \text{Unit} \Rightarrow \text{Unit} \} \\
\]

\[
\text{val } f : (\text{T1} \rightarrow[^*] \text{T2}) \rightarrow[^*] \text{Unit} = \ldots \\
\]

\[
\text{let } \text{cnt}_f \ g = \\
\quad \text{handle } f \ (\text{fn } x \Rightarrow \text{tick }(); \ g \ x) \ \text{with} \\
\quad | \text{tick }() \Rightarrow \text{fn } n \Rightarrow \text{resume }(); \ (n+1) \\
\quad | \text{return }_\_ \Rightarrow \text{fn } n \Rightarrow n \\
\quad \text{end } 0
\]

But what if \( g \) uses \( \text{Tick} \) 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:

```ml
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]
 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
A glimpse at the type system:

**Kinds and Types**

\[
\text{Kind } \exists \kappa \ ::= \ T \ | \ E \ | \ R \ | \ \kappa \to \kappa \\
\text{Typelike } \exists \sigma, \tau, \varepsilon, \rho \ ::= \ \alpha \ | \ \tau \ \tau \ | \ \tau \to \rho \tau \ | \ \forall \alpha :: \kappa. \ \tau \ | \ \exists \alpha :: \kappa. \ \tau \ | \ \langle \rangle \ | \ \langle \varepsilon | \rho \rangle
\]

**Judgements**

\[
\Delta, \Gamma \vdash e : \tau / \rho
\]
Typing Rule for Local Effects

\[
\begin{align*}
\Delta, \alpha = \theta & \vdash \theta \\
\Delta, \alpha = \theta; \Gamma & \vdash e : \tau / \rho \\
\Delta & \vdash \tau :: T \\
\Delta & \vdash \rho :: R \\
\Delta; \Gamma & \vdash \text{effect } \alpha = \theta \text{ in } e : \tau / \rho
\end{align*}
\]

Note that \( \tau \) and \( \rho \) cannot mention \( \alpha \)
Typing Rules for Existentials

\[
\frac{\Delta \vdash \sigma :: \kappa \quad \Delta; \Gamma \vdash e : \tau \{\sigma / \alpha\} / \rho}{\Delta; \Gamma \vdash \text{pack}(\sigma, e) \ as \ \exists \alpha :: \kappa. \tau : \exists \alpha :: \kappa. \tau / \rho}
\]

\[
\frac{\Delta; \Gamma \vdash e_1 : \exists \alpha :: \kappa. \sigma / \rho \quad \Delta, \alpha :: \kappa; \Gamma, x : \sigma \vdash e_2 : \tau / \rho}{\Delta; \Gamma \vdash \text{unpack} \ e_1 \ as \ \alpha :: \kappa, x : \sigma \ in \ e_2 : \tau / \rho}
\]

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

\[
\alpha = \beta :: \kappa. \quad \{\delta\} \in \Delta \quad \Delta \vdash \sigma :: \kappa \quad \Delta; \Gamma \vdash e : \tau_a / \langle \alpha \sigma | \rho \rangle \\
\Delta; \Gamma; \delta\{\sigma / \beta\} \vdash h : \tau_r / \rho \quad \Delta; \Gamma, x : \tau_a \vdash e_r : \tau_r / \rho \\
\Delta; \Gamma \vdash \text{handle}_{\alpha \sigma} e \{h; \text{return } x : \tau_a \Rightarrow e_r\} : \tau_r / \rho
\]

\[
\Delta \vdash \varepsilon_1 \# \varepsilon_2 \\
\Delta \vdash \langle \varepsilon_1, \varepsilon_2 | \rho \rangle \simeq \langle \varepsilon_2, \varepsilon_1 | \rho \rangle :: R
\]

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

\[ \text{handle}_{\text{Reader int}} e \{ \ldots ; \ldots \} \]

simply won’t type-check, because we cannot guarantee that \( \alpha \neq \text{Reader int} \)

Our solution is to use...
Explicit Coercions in the Core Language

\[
\begin{align*}
\Delta; \Gamma \vdash e : \tau / \rho & \quad \Delta \vdash c : \rho \triangleright \rho' \\
\Delta; \Gamma \vdash \langle c \rangle e : \tau / \rho' \\
\Delta \vdash \varepsilon :: E & \\
\Delta \vdash \uparrow \varepsilon : \rho \triangleright \langle \varepsilon | \rho \rangle & \\
\Delta \vdash \varepsilon_1 \leftrightarrow \varepsilon_2 : \langle \varepsilon_1, \varepsilon_2 | \rho \rangle \triangleright \langle \varepsilon_2, \varepsilon_1 | \rho \rangle & \\
\Delta \vdash c : \rho \triangleright \rho' & \\
\Delta \vdash \varepsilon : \langle \varepsilon | \rho \rangle \triangleright \langle \varepsilon | \rho' \rangle & \\
\Delta \vdash c_1 : \rho_1 \triangleright \rho_2 & \quad \Delta \vdash \rho_1 \simeq \rho_2 :: R & \quad \Delta \vdash c_2 : \rho_2 \triangleright \rho_3 \\
\Delta \vdash c_1 \cdot c_2 : \rho_1 \triangleright \rho_3 &
\end{align*}
\]
Conclusions

**Implementation:** a simple language with ML-style module system built on top of existentials ([https://bitbucket.org/pl-uwr/helium](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.