Higher-Order Programming is an Effect

Oleg Kiselyov

HOPE
September 3, 2017
Outline

► Introduction and Conclusion

Extensible Interpreter Problem

Extensible Interpreter Solution

Lambda as an Effect

Conclusions
Summary

Problem
Interaction of higher-order programming with Effects
Summary

Solution

- Interaction of higher-order programming with Effects
History

- Frege (1879)
History

- Frege (1879)
- Carl Adam Petri (1939, at the age of 13)
History

- Frege (1879)
- Carl Adam Petri (1939, at the age of 13)
- Hewitt, 1976
- Fifth Generation Project, 1982-1992
- Robin Milner, end of 1970s
- Kohei Honda, end of 1980s
- Cartwright/Felleisen, 1994
- The first version of this code, 2006
- Extensible-effects, 2013
History

- Frege (1879)
- Carl Adam Petri (1939, at the age of 13)
- Hewitt, 1976
- Fifth Generation Project, 1982-1992
- Robin Milner, end of 1970s
- Kohei Honda, end of 1980s
- Cartwright/Felleisen, 1994

History

- Frege (1879)
- Carl Adam Petri (1939, at the age of 13)
- Hewitt, 1976
- Fifth Generation Project, 1982-1992
- Robin Milner, end of 1970s
- Kohei Honda, end of 1980s
- Cartwright/Felleisen, 1994
- The first version of this code, 2006

History

- Frege (1879)
- Carl Adam Petri (1939, at the age of 13)
- Hewitt, 1976
- Fifth Generation Project, 1982-1992
- Robin Milner, end of 1970s
- Kohei Honda, end of 1980s
- Cartwright/Felleisen, 1994
- The first version of this code, 2006
- Extensible-effects, 2013

Outline

Introduction and Conclusion

- Extensible Interpreter Problem

Extensible Interpreter Solution

Lambda as an Effect

Conclusions
The Central Problem

Stable Denotations
Unstable denotations doomed the denotational semantics

or,

Extensible Interpreters
Extensible Interpreters

type \text{exp} = \text{Int of int} \mid \text{Inc of exp}

let rec eval : exp \to int = function
\mid \text{Int n} \to n
\mid \text{Inc e} \to \text{eval e} + 1

denotational semantics (eval as the semantic function)
Non-Extensible Interpreters

\[
\text{type } \text{exp} = \text{Int of } \text{int} \mid \text{Inc of } \text{exp}
\]

\[
\text{let rec } \text{eval} : \text{exp } \rightarrow \text{int} \\
    = \text{function} \\
    | \text{Int } n \rightarrow n \\
    \ldots
\]
Non-Extensible Interpreters

type exp =  Int of int  |  Inc of exp
  |  Equal of exp * exp  |  If of exp * exp * exp

let rec eval : exp -> (int + bool)
  = function
  | Int n       -> Left n
...

Non-Extensible Interpreters

\[
\text{type } \text{exp} = \text{Int of } \text{int} \mid \text{Inc of } \text{exp} \\
\quad \mid \text{Equal of } \text{exp} \ast \text{exp} \mid \text{If of } \text{exp} \ast \text{exp} \ast \text{exp} \\
\quad \mid \text{Exc of } \text{string}
\]

\[
\text{let rec eval : exp } \rightarrow (\text{int } + \text{ bool } + \text{ exc}) \\
\quad = \text{ function} \\
\quad \mid \text{Int } n \quad \rightarrow \text{ Left?? } n \\
\quad ...
\]
Non-Extensible Interpreters

\[
\texttt{type exp = \text{Int of int | Inc of exp}} \\
\quad \quad \text{| Equal of exp \star exp | If of exp \star exp \star exp} \\
\quad \quad \text{| Exc of string} \\
\quad \quad \text{| Get | Put of exp}
\]

\[
\text{let rec eval : exp } \rightarrow \text{ (state } \rightarrow \text{ (int } + \text{ bool } + \text{ exc, state))} \\
\quad = \text{ function} \\
\quad \quad \text{ | Int n \rightarrow fun s } \rightarrow \text{ (Left?? n, s)}
\]

\ldots
Non-Extensible Interpreters

\[
\text{type } \text{exp} = \text{Int of int} \mid \text{Inc of exp} \\
\mid \text{Equal of exp} \ast \text{exp} \mid \text{If of exp} \ast \text{exp} \ast \text{exp} \\
\mid \text{Exc of string} \\
\mid \text{Get} \mid \text{Put of exp} \\
\mid \text{Var of vname} \mid \text{Lam of vname} \ast \text{exp} \mid \text{App of exp} \ast \text{exp}
\]

\[
\text{type } \text{env} = \text{vname} \to \text{v}??
\]

\[
\text{let rec eval : exp } \to \text{ (env } \to \text{ state } \to \text{ (v } + \text{ exc, state))}
\]

\[
= \text{ function}
\mid \text{Int n } \to \text{ fun env } \to \text{ fun s } \to \text{ (Left (Left n), s)}
\]

\[
= \text{ function}
\mid \text{Int n } \to \text{ fun env } \to \text{ fun s } \to \text{ (Left (Left n), s)}
\]

\[
= \text{ function}
\mid \text{Int n } \to \text{ fun env } \to \text{ fun s } \to \text{ (Left (Left n), s)}
\]

\[
\text{let rec eval : exp } \to \text{ (env } \to \text{ state } \to \text{ (v } + \text{ exc, state))}
\]

\[
= \text{ function}
\mid \text{Int n } \to \text{ fun env } \to \text{ fun s } \to \text{ (Left (Left n), s)}
\]

A knotty problem
Towards Extensible Interpreters

type exp = Int of int | Inc of exp

let rec eval : {int → int} → exp → int
  = fun {inj} → function
  | Int n → inj n
  ...

Towards Extensible Interpreters

```
type exp = Int of int | Inc of exp | Equal of exp * exp | If of exp * exp * exp | Exc of string

let rec eval : (int \rightarrow (v + exc)) \rightarrow exp \rightarrow (v + exc)
  = fun {inj} \rightarrow function
  | Int n \rightarrow inj n
  ...
```
Towards Extensible Interpreters

\[
\text{type exp = Int of int | Inc of exp} \\
| \quad \text{Equal of exp * exp | If of exp * exp * exp} \\
| \quad \text{Exc of string} \\
| \quad \text{Get | Put of exp}
\]

\[
\text{let rec eval : (int } \rightarrow \omega) \rightarrow \exp \rightarrow \\
(\text{(state } \rightarrow (v + exc, state)) \text{ as } \omega)
\]

\[
= \text{ fun inj } \rightarrow \text{ function} \\
| \quad \text{Int n } \rightarrow \text{ inj n}
\]

...
Further Towards Extensible Interpreters

\[
\text{type } \text{exp} = \text{Int of int} \mid \text{Inc of exp}
\]

\[
\text{let rec eval : \{\ldots\} } \rightarrow \text{exp} \rightarrow \text{int} = \text{fun \{inj,prj\} } \rightarrow \text{function} \\
\mid \text{Int } n \rightarrow \text{inj } n \\
\mid \text{Inc } e \rightarrow \text{match } \text{prj (eval } e) \text{ with} \\
\quad | \text{Left } n \rightarrow \text{inj } (n+1) \\
\quad | \text{Right } e \rightarrow ???
\]
Further Towards Extensible Interpreters

\[ \text{type } \text{exp} = \text{Int of int} \mid \text{Inc of exp} \]
\[ \mid \text{Equal of exp} \ast \text{exp} \mid \text{If of exp} \ast \text{exp} \ast \text{exp} \]
\[ \mid \text{Exc of string} \]
\[ \mid \text{Get} \mid \text{Put of exp} \]

\[ \text{let rec eval : (int} \rightarrow \omega) \rightarrow \text{exp} \rightarrow \]
\[ \quad ((\text{state} \rightarrow (v + \text{exc}, \text{state})) \text{ as } \omega) \]
\[ = \text{fun } \{\text{inj, prj}\} \rightarrow \text{function} \]
\[ \mid \text{Int n} \rightarrow \text{inj n} \]
\[ \mid \text{Inc e} \rightarrow \text{match prj (eval e) with} \]
\[ \quad \text{Left n} \rightarrow \text{inj (n+1)} \]
\[ \quad \text{Right e} \rightarrow ??? \]
Outline

Introduction and Conclusion

Extensible Interpreter Problem

► Extensible Interpreter Solution

Lambda as an Effect

Conclusions
Extensible Interpreters

```
type v = ..
type v + = VInt of int

type e = ..
type e + = Error of string

type c = V of v | FX of e ∗ (v → c)

let rec eval : exp → c = function
  | Int n → V (VInt n)
  | Inc e →
    let rec inc = function
      | V (VInt n) → V (VInt (n + 1))
      | FX (e, k) → FX (e, fun x → k (inc x))
in  inc (eval e)
```
Extensible Interpreters

\[
\text{type } c = \text{ V of } v \mid \text{ FX of } e \ast (v \rightarrow c)
\]

\[
\text{let rec lift : } c \rightarrow (v \rightarrow c) \rightarrow c = \text{ fun } c \ k \rightarrow \text{ match } c \text{ with }
\]
\[
| \text{ V } x \rightarrow k \ x
\]
\[
| \text{ FX } (e,k1) \rightarrow \text{ FX } (e, \text{ fun } x \rightarrow \text{ lift } (k1 \ x) \ k)
\]

\[
\text{let rec eval : exp } \rightarrow c = \text{ function }
\]
\[
| \text{ Int } n \rightarrow \text{ V } (\text{ VInt } n)
\]
\[
| \text{ Inc } e \rightarrow \text{ lift } (\text{ eval } c) @@ \text{ function }
\]
\[
| \text{ V } (\text{ VInt } n) \rightarrow \text{ V } (\text{ VInt } (n+1))
\]
let send: e → c = fun e → FX(e,(fun x → V x))

type e + = EGet | EPut of v

let rec eval : exp → c = function
| Get → send EGet
| Put e → lift (eval e) @@ fun v → send (EPut v)
State

let rec eval : exp → c
  = function
  | Get       → send EGet
  | Put e → lift (eval e) @@ fun v → send (EPut v)

let rec observe : v → c → c = fun state → function
  | V x → V x
  | FX (EGet,k) → observe state (k state)
  | FX (EPut state,k) → observe state (k state)
  | FX (e,k) → FX (e, fun x → observe state (k x))
Outline

Introduction and Conclusion

Extensible Interpreter Problem

Extensible Interpreter Solution

▶ Lambda as an Effect

Conclusions
Lambda as State?

```ocaml
type exp =  Int of int | Inc of exp
    | Equal of exp * exp | If of exp * exp * exp
    | Exc of string
    | Get | Put of exp
    | Var of vname | Lam of vname * exp | App of exp * exp

type env =  vname -> v??
let rec eval : exp -> (env -> state -> (v + exc, state))
...
```
type \( v \) \( + = \) VFun of (\( v \to c \))

\texttt{type } e \texttt{ + = EVar of } \texttt{vname } | \texttt{EClosure of } \texttt{vname } \ast \texttt{d}

\begin{verbatim}
let rec eval : exp \to c = function
  | Var v \to send (EVar v)
  | Lam (v,body) \to send (EClosure (v, eval body))
  | App (e1,e2) \to lift2 (eval e1) (eval e2) @@ function
      (VFun f,x) \to f x
\end{verbatim}
let rec observe : env → c → c = fun env → function
| V x → V x
| FX (EVar var,k) → lookup var env @@
  fun v → observe env (k v)
| FX (EClosure (var,body), k) →
  let v = VFun (fun x → observe ((var,x):: env) body) in
  observe env (k v)
| FX (e,k) → FX (e, fun x → observe env (k x))
Outline

Introduction and Conclusion

Extensible Interpreter Problem

Extensible Interpreter Solution

Lambda as an Effect

► Conclusions
Higher-Order Programming is an Effect

Oleg Kiselyov

HOPE
September 3, 2017

\[ \Gamma \vdash M : Te_T \implies M : Te_T \]
Extending interpreters, mix-and-match features in any combination


using Haskell

Detailed (1h38m) talk
YouTube; using Haskell

This workshop’s page
OCaml code with many comments
“Here’s the issue. It looks as though the future of scaling is lots of processors, running slower than typical desktops, with things turned down or off as much as possible, so you won’t be able to pull the Parallela/Epiphany trick of always being able to access another chip’s local memory. Any programming model that relies on large flat shared address spaces is out; message passing that copies stuff is going to be much easier to manage than passing a pointer to memory that might be powered off when you need it; anything that creates tight coupling between the execution orders of separate processors is going to be a nightmare.”

“In sequential computing, the value of a variable at a given time point is a well-defined notion. On the other hand, I had thought in studying the properties of Guarded Horn Clauses (GHC) that the value of a variable observed at some time point and place would not necessarily be a well-defined notion. I thought that the model (or more specifically, the memory model) of concurrency we wanted to establish should allow the observation of the value of a variable to take time and the value of a variable to be transmitted asynchronously to each occurrence of the variable.”

Kazunori Ueda: The Hard-Won Lessons of the Fifth Generation Computer Project, 2017
Conventional Computational Model no longer sufficed

At the workshop on Functional and Logic Programming held in Trento, Italy in December 1986, I asked if there was a theory that made clear distinction between variables and occurrences of variables. Per Martin-Loef responded that Jean-Yves Girard of the University of Paris 7 was considering Linear Logic. Linear Logic, published immediately after that, had no direct connection to asynchrony I was thinking about...”

Kazunori Ueda: The Hard-Won Lessons of the Fifth Generation Computer Project, 2017