Higher-Order Programming *is* an Effect

Oleg Kiselyov

HOPE
September 3, 2017

Outline

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

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

`http://okmij.org/ftp/Computation/`*`having-effect.html`*

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

`http://okmij.org/ftp/Computation/`*`having-effect.html`*

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

`http://okmij.org/ftp/Computation/`*`having-effect.html`*

Outline

The Central Problem

Stable Denotations

Unstable denotations doomed the denotational semantics

or,

Extensible Interpreters

Extensible Interpreters

```
type exp =  Int of int  |  Inc  of exp

let  rec  eval  :  exp → int
 =  function
 |  Int  n    → n
 |  Inc  e    → eval  e  +  1
```

denotational semantics (eval as the semantic function)

Non-Extensible Interpreters

```
type exp =  Int of int  |  Inc  of exp
```

```
let  rec  eval  :  exp → int
 =  function
 |  Int  n          → n
   ...
```

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

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

```
let  rec  eval  :  exp → (int  +  bool +  exc)
 =  function
 |  Int  n          → Left?? n
  ...
```

Non-Extensible Interpreters

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

```
let  rec  eval  :  exp → (state  → (int  +  bool +  exc, state))
 =  function
 |  Int  n          → fun s  → (Left?? n,s)
  ...
```

Non-Extensible Interpreters

```
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))
 =  function
 |  Int  n          → fun env → fun s  → (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 → (v + exc)) → exp → (v + exc)
 = 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
    |  Get  |  Put of exp
```

```
let  rec  eval  :  (int  → ω) → exp →
                   ((state  → (v  +  exc, state))  as  ω)
 =  fun inj → function
 |  Int  n          → inj  n
  ...
```

Further Towards Extensible Interpreters

```
type exp = Int of int | Inc of exp
```

```
let rec eval : {...} → exp → int
 = fun {inj, prj} → function
 | Int n   → inj n
 | Inc e   → match prj (eval e) with
        | Left n  → inj (n+1)
        | Right e → ???
```

Further Towards Extensible Interpreters

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

```
let  rec  eval  :  (int  → ω) → exp →
                   ((state  → (v  +  exc, state))  as  ω)
 =  fun {inj, prj} → function
 |  Int  n          → inj  n
 |  Inc  e     → match prj (eval  e)  with
          |  Left  n   →  inj  (n+ 1)
          |  Right  e  →  ???
```

Outline

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

```
type c =  V of v | FX of e * (v → c)

let rec  lift : c → (v → c) → c =  fun c k → match c with
 | V x          → k x
 | FX (e,k1) → FX (e, fun x →  lift  (k1 x)  k)


let rec eval  : exp → c
 =  function
 | Int  n   → V (VInt  n)
 | Inc  e   →  lift  (eval  c)  @@ function
        | V (VInt  n) → V (VInt  (n+ 1))
```

State

```
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

Lambda as State?

```
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))
  ...
```

Lambda

```
type v += VFun of (v → c)
type e += EVar of vname | EClosure of vname * d
```

```
let rec eval : exp → c
 = function
  | Var v          → send (EVar v)
  | Lam (v,body) → send (EClosure (v, eval body))
  | App (e1,e2)  →  lift2 (eval e1) (eval e2) @@ function
                        (VFun f,x) → f x
```

Lambda

```
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

Higher-Order Programming *is* an Effect

Oleg Kiselyov

HOPE
September 3, 2017

$$\Gamma \vdash M : Te\tau \qquad \Longrightarrow \qquad M : Te\tau$$

Details

Extending interpreters, mix-and-match features in any combination

`http://okmij.org/ftp/Computation/having-effect.html`

using Haskell

Detailed (1h38m) talk

YouTube; using Haskell

This workshop's page

OCaml code with many comments

Conventional Computational Model no longer suffices

"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."
Richard A. O'Keefe. Haskell-Cafe, October 28, 2016

Conventional Computational Model no longer sufficed

"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