The essence of Frank programming

Craig McLaughlin

The University Of Edinburgh

November 8, 2016

Joint work with Sam Lindley and Conor McBride (**POPL 2017**)

The What, Why, and Whence of Effectful Computations

What:

- example effects include I/O, State, Exceptions

The What, Why, and Whence of Effectful Computations

What:

- example effects include I/O, State, Exceptions

Why:

- real world software interacts with its environment
- desire: effects performed only where intended

The What, Why, and Whence of Effectful Computations

What:

- example effects include I/O, State, Exceptions

Why:

- real world software interacts with its environment
- desire: effects performed only where intended

Whence:

- solution: specify effects in types
- algebraic theory of effects & handlers
- effects are collection of abstract symbols with types (*commands*)
  e.g `Abort` effect with command `aborting`

✓ algebraic effects compose

✓ effect handler: command interpreter

What's Frank?

Frank:

- strict functional language
- effects as collections of *commands*, as before

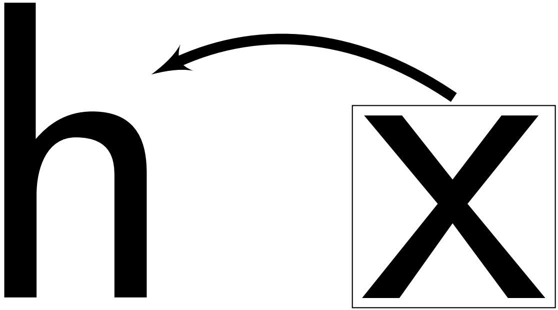
Novelties:

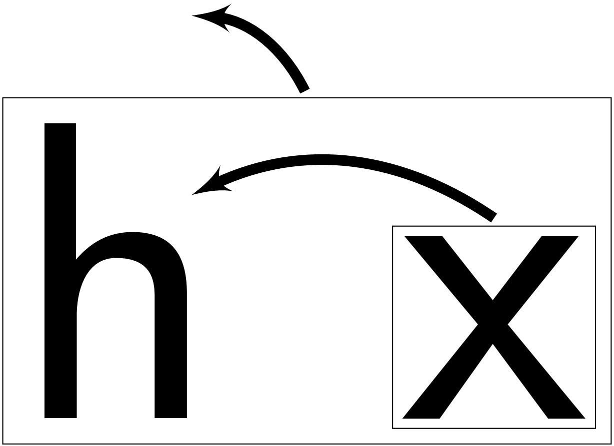
- effect type system for statically tracking effects
- handlers arise by generalising a familiar construct...

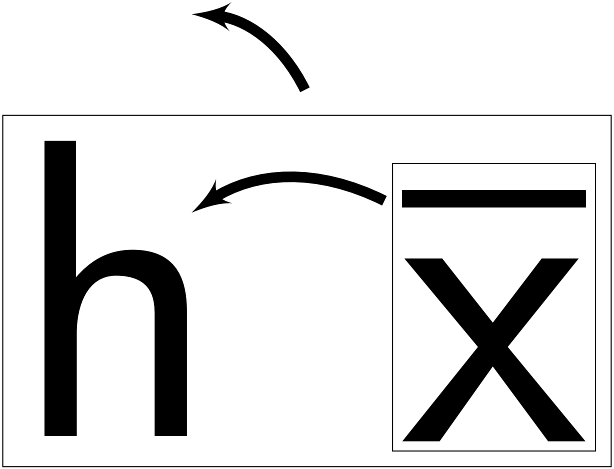
Experimental implementation:
`https://www.github.com/cmcl/frankjnr`, **try today!**

$f\ x$

h x


h
x


h
x


h
x̄

Functional "Hello World" in Frank

```
map : {X -> Y} -> List X -> List Y

map f nil        = nil
map f (x :: xs) = f x :: map f xs
```

Functional "Hello World" in Frank

```
map : {X -> Y} -> List X -> List Y

map f nil         = nil
map f (x :: xs) = f x :: map f xs

map {n -> n+1} [1,2,3]
  -- result: [2,3,4]
```

Example: Declaring Effects in Frank

```
interface Abort = aborting : Zero

interface Write X = tell : X -> Unit

interface Read X = ask : X
```

Example: Writing a List

```
interface Write X = tell : X -> Unit

writeList : List X -> [Write X]Unit

writeList xs = map tell xs; unit
```

Example: Writing a List

```
interface Write X = tell : X -> Unit

writeList : List X -> [Write X]Unit

writeList xs = map tell xs; unit
```

$$[\Sigma]$$

"Hi, I'm an *ability*.
The environment must be *able* to *handle* effects declared in Σ"

Example: Interpreting Read and Write

```
state : S -> <Read S,Write S>X  -> X

state _ x = x
state s <ask -> k> = state s (k s)
state _ <tell s -> k> = state s (k unit)
```

Example: Interpreting Read and Write

```
state : S -> <Read S,Write S>X  -> X

state _ x = x
state s <ask -> k> = state s (k s)
state _ <tell s -> k> = state s (k unit)
```

$$\langle \Delta \rangle$$

"Hi, I'm an *adjustment*.
The effects declared in Δ must be handled locally."

Desugaring The Type of Map

```
map : {X -> Y} -> List X -> List Y
```

desugars to

$\langle\iota\rangle$`{`$\langle\iota\rangle$`X -> `$[\varepsilon]$`Y} -> `$\langle\iota\rangle$`List X -> `$[\varepsilon]$`List Y`

Desugaring The Type of Map

```
map : {X -> Y} -> List X -> List Y
```

desugars to

$\langle\iota\rangle$`{`$\langle\iota\rangle$`X -> `$[\varepsilon]$`Y} -> `$\langle\iota\rangle$`List X -> `$[\varepsilon]$`List Y`

Aside for Haskell programmers:

We've got something that's equivalent to both `map` and `mapM`!

Catching More Precisely

```
catch : <Abort>X -> {X} -> X

catch x _   =   x
catch <aborting -> _> h   =   h!
```

Catching More Precisely

```
catch : <Abort>X -> {X} -> X

catch x _  =  x
catch <aborting -> _> h  =  h!
```

```
catchError :: -- Haskell
   MonadError () m => m a -> (() -> m a) -> m a
```

Imprecise typing (`() -> m a`) permits alternative to throw errors!

Multihandler Example: Preliminaries

```
on : X -> {X -> Y} -> Y

on x f = f x

abort   : [Abort]X

abort! = on aborting! {}
```

Multihandler Example: Pipe

```
pipe:⟨Write X⟩Unit -> ⟨Read X⟩Y -> [Abort]Y

pipe <tell x -> w> <ask -> r> =
  pipe (w unit) (r x)

pipe <_> y = y

pipe unit <_> = abort!
```

That's Frank!

Conclusions:

- Application generalises to account for both functions & handlers
- Effect type system: effects tracked and pushed inwards
- Convenient syntactic sugars: rarely need specify effect variables

That's Frank!

Conclusions:

- Application generalises to account for both functions & handlers
- Effect type system: effects tracked and pushed inwards
- Convenient syntactic sugars: rarely need specify effect variables

One more thing...did I mention there's an implementation?!

`https://www.github.com/cmcl/frankjnr`, **try today!**

Questions?

Tread softly because you tread on my dreams

—William Butler Yeats

Backup Slides

Application Typing Rule

$$\frac{\Gamma\,[\Sigma]\!\vdash\; m \Rightarrow \{\overline{\langle\Delta\rangle A \rightarrow}\;[\Sigma']B\} \qquad \Sigma = \Sigma' \qquad \overline{\Gamma\,[\Sigma \oplus \Delta]\!\vdash\; n : A}}{\Gamma\,[\Sigma]\!\vdash\; m\;\overline{n} \Rightarrow B}$$

Definition (Normal Forms)

If $\Gamma\ [\Sigma]\!\vdash\ n : A$ then we say that n is normal with respect to Σ if it is either a value w or of the form $\mathcal{E}[c\ \overline{w}]$ where $c : \overline{A} \to B \in \Sigma$ and $c \notin HC(\mathcal{E})$.