Talking to Frank

Craig McLaughlin

The University of Edinburgh

March 26, 2019

Joint work with Lukas Convent, Sam Lindley and Conor McBride

What's Frank?

Frank:

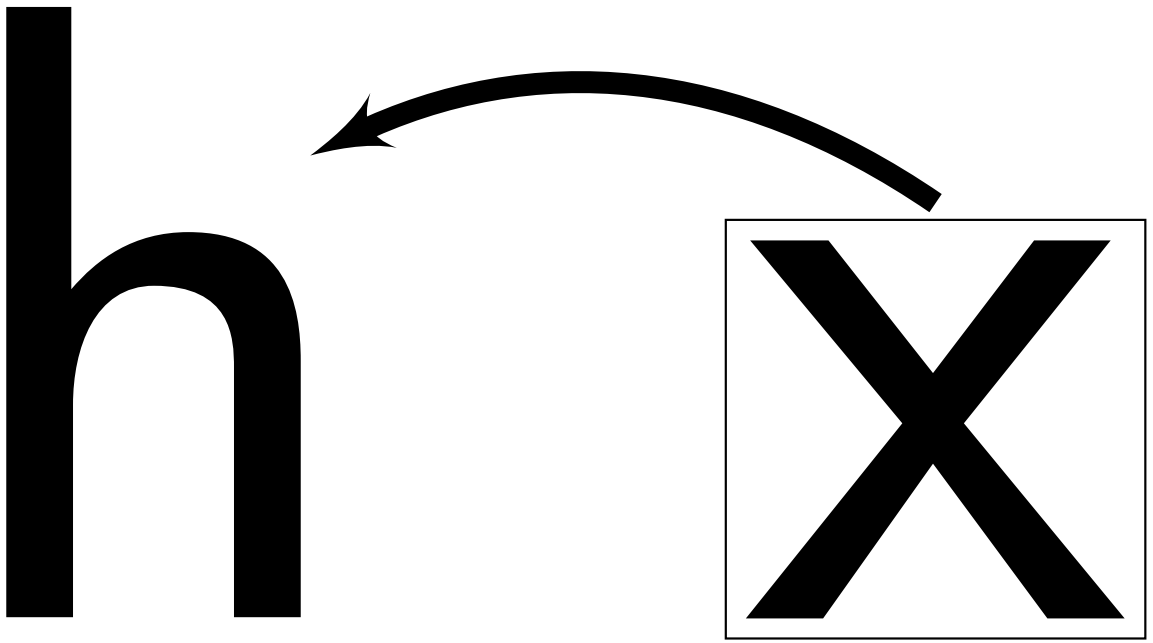
- strict functional language
- effects as collections of *commands* (effect operations)

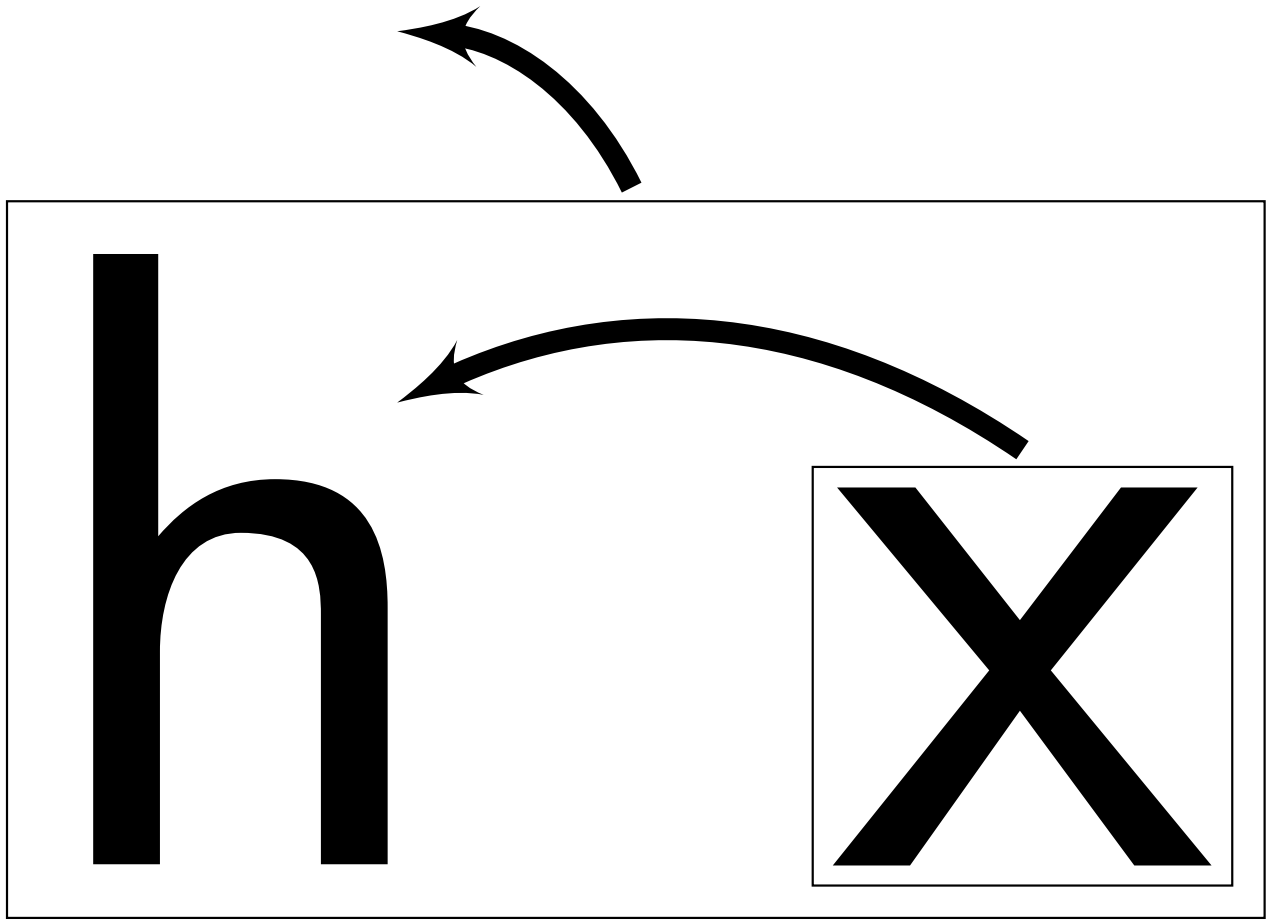
Novelties:

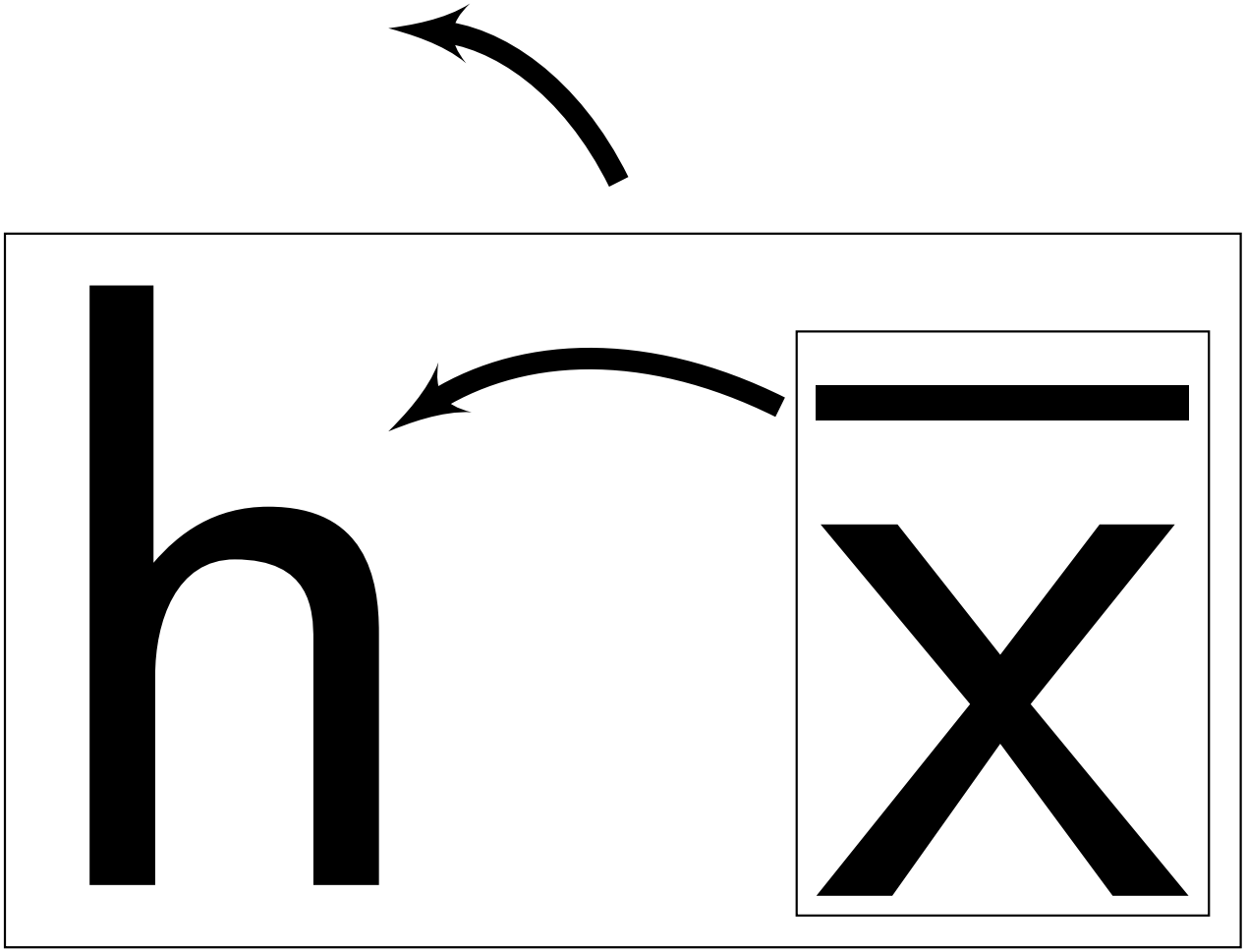
- effect type system for statically tracking effects
- effect handling arising from **generalising function application**

Implementation:

`https://www.github.com/frank-lang/frank` — **try today!**


h
X


h
X


h
x

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] ⟹ [2,3,4]
```

Example: Declaring Effects in Frank

```
interface Abort = abort X : X

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

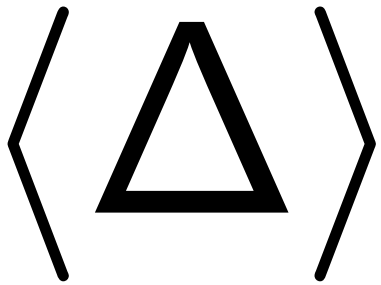
```
state : S -> <Read S,Write S>X  -> X

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)
```



"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`!

Do be do a demo

Demo

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
- Adaptors provide general rewiring of effects in the ambient ability

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
- Adaptors provide general rewiring of effects in the ambient ability

`https://www.github.com/frank-lang/frank`

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!