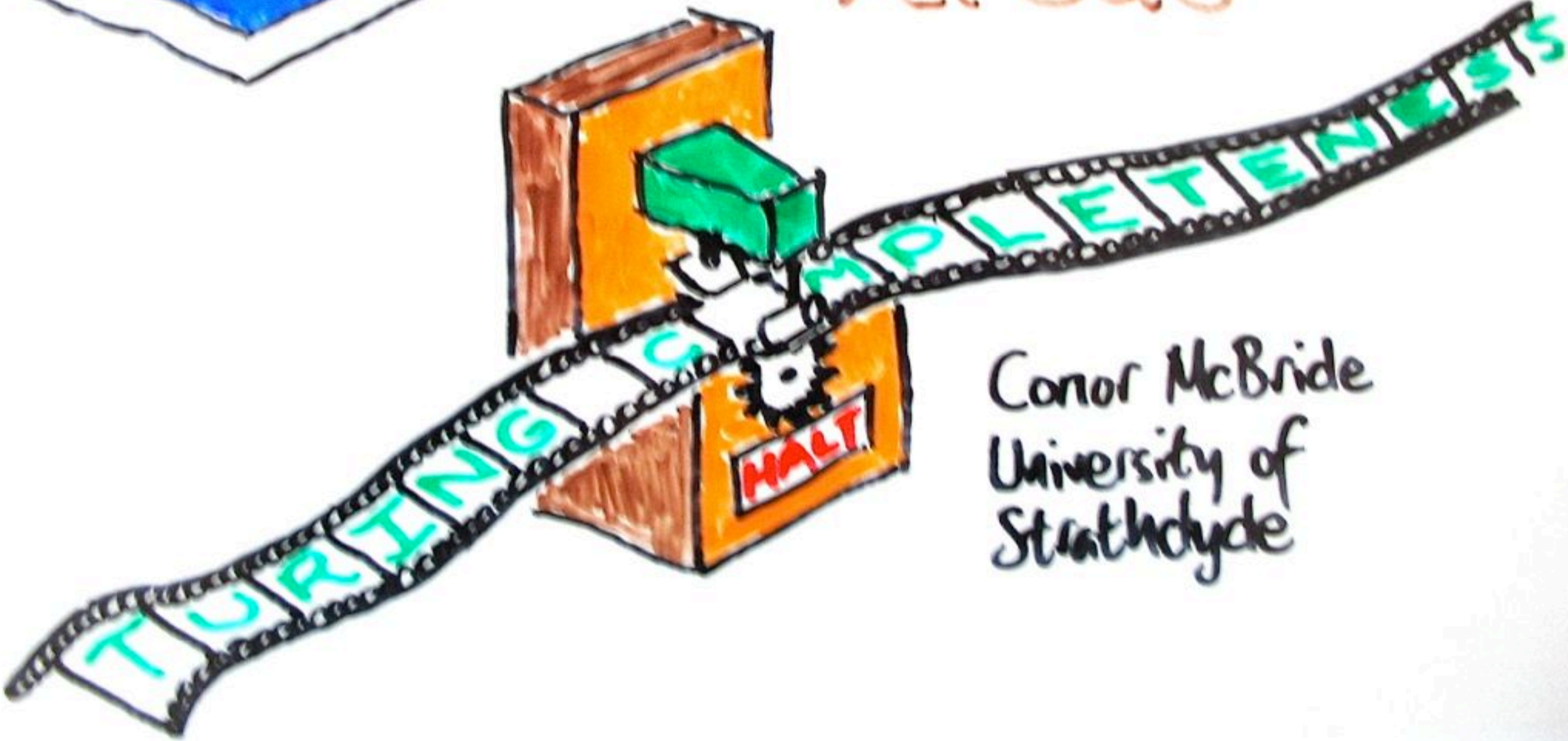
TOTALITY versus TURING COMPLETENESS



Conor McBride
University of Strathclyde

XKCD 386


ARE YOU COMING TO BED?
I CAN'T THIS IS IMPORTANT.
WHAT?
SOMEONE IS WRONG ON THE INTERNET.

David Turner on

Total Functional Programming

(+) "Strong Church-Rosser Property"

but "two obvious disadvantages"

(−) "Our programming language is no longer Turing complete!"

(−) "If all programs terminate, how do we write an operating system?"

This talk:

BULLISH DEFENCE OF TOTAL PROGRAMMING

David Turner on

Total Functional Programming

(+) "Strong Church-Rosser Property"

but "two obvious disadvantages"

(–) "Our programming language is no longer Turing complete!"

(–) "If all programs terminate, how do we write an operating system?"

This talk:

BULL SHI
DEFENCE
OF TOTAL
PROGRAMMING

David Turner on

Total Functional Programming

(+) "Strong Church-Rosser Property"

but "two obvious disadvantages"

(−) "Our programming language is no longer Turing complete!"

(−) "If all programs terminate, how do we write an operating system?"

This talk:

BULLSHI DEFENCE OF TOTAL COPROGRAMMING

Turner rightly identifies codata and total coprogramming as a means to write operating systems, etc.

Capretta rightly notes that codata, and specifically

$$\nu Y.\ Y + X,$$

addresses Turing completeness, also. There are total models of partiality.

What is total programming?

- there are expressions and types
- some expressions are values
- expressions reduce unless they are values
- every reduction sequence starting from a well typed expression is finite
- a well typed expression eventually reduces to exactly one value

What about codata? e.g.

$$\mathsf{codata}\ \mathrm{Stream}\ X$$
$$\ni (x : X) :> (xs : \mathrm{Stream}\ X)$$

What about codata ? e.g.

codata Stream X

$\ni (x : X) :> (xs : \mathrm{Stream}\ X)$

coconstructors make patterns, but not values

What about codata? e.g.

codata Stream X

$\ni (x : X) :> (xs : \mathrm{Stream}\ X)$

coconstructors make patterns, but not values

$$\frac{f : S \to X \times S \qquad s : S}{\mathrm{unfold}\ f\ s : \mathrm{Stream}\ x}$$

$$\left(\begin{array}{l}\text{case unfold } f\ s \text{ of} \\ \quad x :> xs \mapsto e[x, xs]\end{array}\right)$$

$$\rightsquigarrow$$

$$\left(\begin{array}{l}\text{let } (x, s') = f\ s \\ \text{in } e[x, \text{unfold } f\ s']\end{array}\right)$$

How to trace an evolving system

evolve : Config

trace : Config → Stream Config
trace = unfold (dup · evolve)

The system might be a Turing machine.

How to trace an evolving system

evolve : Config → Config

trace : Config → Stream Config
trace = unfold (dup · evolve)

The system might be a Turing machine.

BUT YOU WON'T LET
ME WRITE

eventually : (Config → Bool) →
Stream Config → Bool
eventually happy (now :> later)
= happy now orelse
eventually happy later

That's right! I won't. Instead...

```
codata Delay X
  ∋ ret (x : X)
  | wait (y : Delay X)

eventually : (Config → Bool) →
    Stream Config → Delay Bool
eventually happy (now :> later)
  | happy now = ret true
  | true      =
    wait (eventually happy later)
```

That's right! I won't. Instead...

one man went to mow, went to mow a monad

```
codata Delay X
  ∋ ret (x : X)
  | wait (y : Delay X)

eventually : (Config → Bool) →
    Stream Config → Delay Bool
eventually happy (now :> later)
  | happy now = ret true
  | true      =
    wait (eventually happy later)
```

That's right! I won't. Instead...

one man went to mow, went to mow a monad

codata Delay X
  ∋ ret (x : X)
  | wait (y : Delay X)

eventually : (Config → Bool) →
  Stream Config → Delay Bool
eventually happy (now :> later)
  | happy now = ret true
  | true =
    wait (eventually happy later)

UGH! MONADS!

convergence: data about codata

$$\mathsf{data}\ (c : \mathsf{Delay}\ X) \Downarrow (x : X)\ \mathsf{where}$$
$$\mathsf{ret}\ x' \Downarrow x \ni \mathsf{stop}\ (q : x' \equiv x)$$
$$\mathsf{wait}\ c' \Downarrow x \ni \mathsf{go}\ (v : c' \Downarrow x)$$

or define

$$\mathsf{data}\ \mathsf{Conv}\ (c : \mathsf{Delay}\ X)\ \mathsf{where}$$
$$\mathsf{Conv}\ (\mathsf{ret}\ x) \ni \mathsf{now}$$
$$\mathsf{Conv}\ (\mathsf{wait}\ c') \ni \mathsf{later}\ (v : \mathsf{Conv}\ c')$$

and give

$$\mathsf{conv}\ (c : \mathsf{Delay}\ X)(v : \mathsf{Conv}\ c) : X$$
$$\mathsf{conv}\ (\mathsf{ret}\ x)\ \mathsf{now} = x$$
$$\mathsf{conv}\ (\mathsf{wait}\ c')(\mathsf{later}\ v) =$$
$$\mathsf{conv}\ c'\ v$$

and prove

$$c \Downarrow x \rightleftarrows (v : \mathsf{Conv}\ c) \times \mathsf{conv}\ c\ v \equiv x$$

Cave, @emptor! Savvy @emptor!

- interactive systems have episodic evaluation
- types can document possible interaction with the environment, e.g. requesting more electricity
- types document risk
- to argue for undocumented risk is to insist on ignorance of safety

Cave, @emptor!
Savvy @emptor!

- interactive systems have episodic evaluation
- types can document possible interaction with the environment, e.g. requesting more electricity
- types document **risk**
- to argue for undocumented risk is to insist on ignorance of safety

BUT IT MIGHT TAKE "FOREVER" IF NOT FOREVER

Cave, @emptor!
Savvy @emptor!

- interactive systems have episodic evaluation
- types can document possible interaction with the environment, e.g. requesting more electricity
- types document risk
- to argue for undocumented risk is to insist on ignorance of safety

UGH! MONADS!

- functors from signatures

$$[f_1 : S_1 \to T_1 ; \ldots ; f_n : S_n \to T_n]\ X$$

$$\ni\ f_i\ (s : S_i) \blacktriangleleft (k : T_i \to X)$$

"after one interaction, X"

- functors from signatures

$$[f_1 : S_1 \to T_1 ; \ldots ; f_n : S_n \to T_n]\, X$$
$$\ni f_i\,(s : S_i) \blacktriangleleft (k : T_i \to X)$$

"after one interaction, X"

- free monads from signatures

$$\text{data } F^{*}X \ni \mathbf{ret}\ (x : X)$$
$$\mid \{c : F(F^{*}X)\}$$

"eventually, after interaction, deliver an X"

- functors from signatures

$$[f_1 : S_1 \to T_1 ; \ldots ; f_n : S_n \to T_n]\ X$$
$$\ni f_i\ (s : S_i) \blacktriangleleft (k : T_i \to X)$$

"after one interaction, X"

- completely iterative monads from signatures

$$\text{codata } F^\infty X \ni \mathsf{ret}\ (x : X)$$
$$\mid \{c : F(F^\infty X)\}$$

"always ready to interact or deliver an X"

- functors from signatures

$$[f_1 : S_1 \to T_1 ; \ldots ; f_n : S_n \to T_n]\ X$$
$$\ni f_i\,(s : S_i) \blacktriangleleft (k : T_i \to X)$$

"after one interaction, X"

- completely iterative monads from signatures

codata $F^{\infty} X \ni \mathsf{ret}\ (x : X)$
$\quad\quad\mid \{c : F(F^{\infty} X)\}$

"always ready to interact or deliver an X"

Delay X =

$[$  $? : 1 \to 1]^{\infty}\ X$

map

$$\mathsf{map} : (X \to Y) \to [\Sigma]X \to [\Sigma]Y$$

$$\mathsf{map}\ g\ (f\ s \blacktriangleleft k) = f\ s \blacktriangleleft (g \cdot k)$$

bind

$$(\gg=) : F^{*}X \to (X \to F^{*}Y) \to F^{*}Y$$

$$\mathsf{ret}\ x \gg= g = g\ x$$

$$\{c\} \gg= g = \{\mathsf{map}\ (\gg= g)\ c\}$$

(aside: it's not obvious why these are guarded for any old F, but it is obvious when $F = [\Sigma]$)

map

$$map : (X \to Y) \to [\Sigma]X \to [\Sigma]Y$$

$$map\ g\ (f\ s \blacktriangleleft k) = f\ s \blacktriangleleft (g \cdot k)$$

bind

$$(\gg\!=) : F^{\infty}X \to (X \to F^{\infty}Y) \to F^{\infty}Y$$

$$ret\ x \gg\!= g = g\ x$$

$$\{c\} \gg\!= g = \{map\ (\gg\!= g)\ c\}$$

(aside: it's not obvious why these are guarded for any old F, but it is obvious when $F = [\Sigma]$)

plan eff (FRANK - Dec 2007)

- to construct 'Haskell' computations

  $$\Gamma \vdash c : [\Sigma]^{*} X,$$

  elaborate 'ML' expressions

  $$\Gamma \vdash_{\Sigma} e : X$$

  with suitable monadic plumbing

idea: separate values from computations
(following Levy)
track signatures in comp types
check signature inclusion
when forcing a thunk

eff general recursion

- to write general recursive

$$g : S \to \qquad\qquad T$$

eff general recursion

- to write general recursive

$$g : S \to [g : S \to T]^{*} T$$

interact with an oracle

eff general recursion

- to write general recursive

$$g : S \to [g : S \to T]^* T$$

interact with an oracle

- how to run it?

choose a homomorphism respecting the equivalence

$$(\!|g\ s \blacktriangleleft k|\!) \sim g\ s \gg= k$$

(suitably closed)

the  semantics

$$\mathrm{run} : [g : S \to T]^{*}\, X \to \mathrm{Delay}\; X$$

$$\mathrm{run}\;(\mathrm{ret}\; x) = \mathrm{ret}\; x$$

$$\mathrm{run}\;\{g\; s \blacktriangleleft k\} =$$

$$50p \to \{\mathrm{run}\;(g\; s) \gg\!= (\mathrm{run} \cdot k)\}$$

guarded?

the 

semantics

$$\mathsf{run} : [g : S \to T]^{*}\, X \to \mathsf{Delay}\, X$$

$$\mathsf{run}\ (\mathsf{ret}\ x) = \mathsf{ret}\ x$$

$$\mathsf{run}\ (g\ s \blacktriangleleft k) =$$

$$50p \to \{\cancel{\mathsf{run}\ (g\ s) \gg\!= (\mathsf{run} \cdot k)}\}$$

guarded?

$$\mathsf{after}\ (g\ s)\ k$$

$$\mathsf{after} : [g : S \to T]^{*}\, X \to (X \to [g : S \to T]^{*}\, Y) \to \mathsf{Delay}\, Y$$

$$\mathsf{after}\ (\mathsf{ret}\ x)\ l = \mathsf{run}\ (l\ x)$$

$$\mathsf{after}\ (g\ s \blacktriangleleft k)\ l =$$

$$\{\mathsf{after}\ (g\ s)\ ((\gg\!= l) \cdot k)\}$$

the Graph semantics

$$\mathrm{data}\ \mathsf{Graph}\ (s, t : S \times T)$$
$$\ni \mathbf{node}\ (n : [\![g\ s]\!]\ \mathsf{Graph}\ t)$$

where

$$[\![(c : [g : S \to T]^{*} X)]\!]$$
$$(G : S \times T \to \mathsf{Set})$$
$$(x : X)$$
$$: \mathsf{Set}$$

$$[\![\mathbf{ret}\ x']\!]\ G\ x \;=\; x' \equiv x$$

$$[\![(g\ s \blacktriangleleft k)]\!]\ G\ x \;=$$
$$(t : T) \times G\,(s, t) \times [\![k\ t]\!]\ G\ x$$

& prove $\mathsf{Graph}\ (s, t) \rightleftarrows \mathrm{run}\ (g\ s) \Downarrow t$

the Bove-Capretta semantics (sketch)

- the Bove-Capretta method
  define by induction-recursion

  $$\left.\begin{array}{l}\text{data } \mathrm{Dom}\ (s : S) \\ \check{g}\ (s : S)(d : \mathrm{Dom}\ s) : T\end{array}\right\}\text{mutual}$$

  show $(s : S) \to \mathrm{Dom}\ s$ (*)
- Dybjer & Setzer have a **coding** for (indexed) I-R definitions
- so, automate Bove-Capretta (apart from (*)) by computing a D-S **code** from a $[g : S \to T]^* T$

variations

check : $Cxt \times Tm \to$
$[$check : $Cxt \times Tm \to Ty$
; abort : $1 \to 0$
$]^* \, Ty$

describes a typechecker

so, trivially extending $[\![..]\!]$
the Graph construction gives
a syntax-directed rule system

- I've used just this method to define a Set:Set type theory and its evaluator in a total type theory

conclusion

- codata make total languages Turing complete
- the user decides how to interact with an unfold
- do anything you like, making weak promises
- do some things with strong promises

- you can write an interpreter for a total language in itself
- you just can't prove it converges
- let's build a ramified hierarchy of total languages!