Fearless programming and reasoning with infinities

Alexander Gryzlov
Research programmer
IMDEA Software Institute

Functional Programming Madrid
09/04/2024

Agenda

- Totality, partiality and fixed points
- Infinite data
- Interfaces and objects
- Automata
- The road ahead

Part 1

Totality, partiality and fixed points

Purely functional programming

= programming and reasoning with referentially transparent higher-order functions

Having effects explicit simplifies reasoning

Purity

Very useful for reasoning about the intention and correctness

Types classify well-behaved programs, but we inevitably lose some programs

So there's a quest to regain expressivity by making type system more powerful

A crucial step is to unify types and programs

Totality

Pushing type-based reasoning further gives dependent types

We can program the type-checker itself

```
data Vec (A : 𝒰) : ℕ → 𝒰 where
  []  : Vec A zero
  _∷_ : A → Vec A n → Vec A (suc n)

data Format = Number Format
            | Str Format
            | Lit String Format
            | End

PrintfType : Format → 𝒰
PrintfType (Number fmt) = (i : Int) → PrintfType fmt
PrintfType (Str fmt) = (str : String) → PrintfType fmt
PrintfType (Lit str fmt) = PrintfType fmt
PrintfType End = String

printfFmt : (fmt : Format) → (acc : String) → PrintfType fmt
printfFmt (Number fmt) acc = λ i → printfFmt fmt (acc ++ show i)
printfFmt (Str fmt) acc = λ str → printfFmt fmt (acc ++ str)
printfFmt (Lit lit fmt) acc = printfFmt fmt (acc ++ lit)
printfFmt End acc = acc
```

Totality

This however means that purity is not enough

Functions have to be total: defined everywhere

Otherwise the typechecking crashes or fails

Hanging terms = inconsistent

```
{-# TERMINATING #-}
void : ⊥
void = void

oops : 2 + 2 = 5
oops = absurd void
```

Semantics

- mathematical function: idealized pairing of input to outputs such that the output is uniquely determined by input (denotational)
- algorithmic function: a tree of instructions for manipulating abstract automata
- denotational view mostly ignores time, however operational view is typically quite low-level

Safety and liveness

Correctness properties typically split into

- safety (nothing bad ever happens)
- liveness (something good eventually happens)

Totality also has two aspects:

- defined input (no crashing)
- producing output (no hanging)

Safety reasoning

Partial functions violating safety = some arguments are not handled

Typically modelled with `Maybe`/`Either`

Violation of liveness = non-termination

The operational/temporal aspect (e.g. complexity) is generally hard to reason with in FP

Essentially, functions can drop and duplicate data in unrestricted fashion

Non-termination

Temporal aspects are typically "invisible"

How to model endless computation?

Total programming usually restricts to terminating functions

Too narrow, cannot reason about interactive programs

Need to model control flow in the type system

Non-termination hacks

We can try adding hacks:

1. construct individual terminating steps
2. make a small unsafe function that spins the steps
3. alternatively, add number of steps and then unsafely generate an infinite number

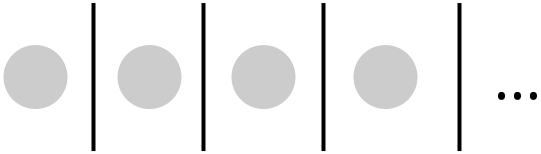
```
data Fuel = Dry | More Fuel

limit : ℕ → Fuel
limit  zero   = Dry
limit (suc n) = More (limit n)

{-# TERMINATING #-}
forever : Fuel
forever = More forever
```

Not very satisfactory, we should have a formal solution

Type-level time

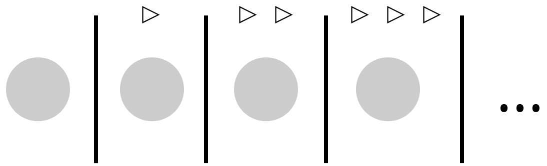
A natural way of reasoning about time is to split it into steps/ticks on some global clock

The flow of time should be unidirectional



A thunk calculus

- Let us introduce a special type constructor $\triangleright$
- $\triangleright A$ is "A, but available one step later"
- Essentially a type-level thunk $() \Rightarrow A$
- Can also be thought of as staging
- The program generates a new program that runs after the first one and so on



Structure of later

```
next : A →  ▷A
ap :  ▷(A → B) →  ▷A →  ▷B
```

It's an applicative functor

(will denote ap by ⊛)

Functorial structure

```
map : (A → B) → ▹ A → ▹ B
map f a▹ = next f ⊛ a▹
```

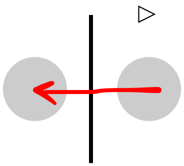
- We can derive a functorial action (will denote map by ⍉)
- All the laws hold definitionally (by symbolic computation)

Not a monad

There is no monadic structure

`flatten : ▹ ▹ A → ▹ A`

This ensures that the temporal structure is preserved

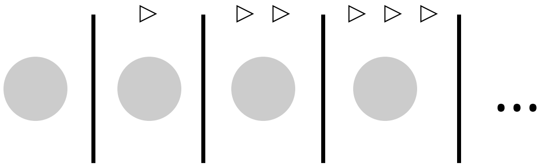


For an arbitrary type there's also typically no

`▹ A → A`

Guarded recursion

- We can schedule computations, what now?
- Terminating → Productive
- Every recursive call is "guarded" by a thunk, giving back control
- Infinite / streaming computations, servers, OSs



Guarded recursion

```
fix: (▹ A → A) → A
```

- A form of Y-combinator
- Postulated definition, unfolding made propositional
- Can be safely erased down to the usual fixpoint

Ticked type theory

- We'll use Agda proof assistant for interactive examples
- Guarded modality is encoded as a function from `Ticks` of a modal type 𝕋
- A proof of an elapsed time step
- Can encode `next`, `ap` and `map`

Ticked cubical type theory

- Technically we're also using the cubical mode of Agda
- No higher equalities / quotients / univalence
- Equality is encoded as a function from a continuous interval 𝕀
- The interval is allowed to "time-travel"
- We can reason about the future in the present

Logical justification

- A flavour of provability modality
- 1933 - Gödel's analysis of `S4`
- 1950s - Löb's axiom: `□(□A → A) → □A`
- We're using the strong Löb's version: `(□A → A) → A`
- 1960s - `GL` system
- 1970s - intuitionistic systems, fixed points
- 2000s - links to formal semantics and category theory

Nakano's approximation

- Nakano, [2000] "A modality for recursion"
- Initially denoted ●
- Continued by many authors, most notably:
- Atkey-McBride'2013
- A series of papers by Birkedal and coauthors in 2010s
- Nowadays symbolized by a right triangle ▷
- Viewed as a special type constructor in a type system

Programming with ▷

So, to recap we have essentially 4 new constructs:

`▷, next, ap/⊛, fix`

(+ `map/⍉` and some proof machinery)

What can we write?

Part 2

Infinite data

Which infinite types make sense?

Fitting the fix

Previously, I've said that for an arbitrary type there typically is no

▹ A → A

However, that is the type of function we need:

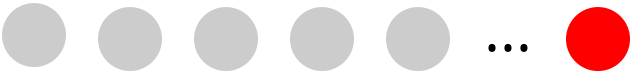
```
fix: (▹ A → A) → A
```

We can construct such types with ▹

Partiality effect

- Recall the motivation of having a non-termination effect
- We can express it with two constructors + a guard
- Many names: `L(ift)`, `Event`, `Delay`

```
data Part (A : 𝒰) : 𝒰 where
  now   : A → Part A
  later : ▹ Part A → Part A
```



Partiality functor

```
map-body : (A → B)
         → ▹ (Part A → Part B)
         →    Part A → Part B

map-body f m▹ (now a)   = now (f a)
map-body f m▹ (later p) = later (m▹ ⊛ p)

map : (A → B) → Part A → Part B
map f = fix (map-body f)
```

Mapping a function = waiting until the end and applying it

Partiality applicative

```
pure : A → Part A
pure = now

ap-body : ▹ (Part (A → B) → Part A → Part B)
          →  Part (A → B) → Part A → Part B
ap-body a▹ (now f)    (now x)    = now (f x)
ap-body a▹ (now f)    (later x▹) = later (a▹ ⊛ next (now f) ⊛ x▹)
ap-body a▹ (later f▹) (now x)    = later (a▹ ⊛ f▹ ⊛ next (now x))
ap-body a▹ (later f▹) (later x▹) = later (a▹ ⊛ f▹ ⊛ x▹)

ap : Part (A → B) → Part A → Part B
ap = fix ap-body
```

Unwind both structures "in parallel"

Partiality monad

```
flatten-body : ▹ (Part (Part A) → Part A)
             →   Part (Part A) → Part A
flatten-body f▹ (now p)    = p
flatten-body f▹ (later p▹) = later (f▹ ⊛ p▹)

flatten : Part (Part A) → Part A
flatten = fix flatten-body
```

- Essentially an arbitrary sequence of nested ▹'s
- Reassociating → a monad
- `▹▹▹(▹▹A) = ▹▹▹▹▹A`

Indexed partiality monad

Can be made more graphic by indexing with steps

```
mapᵈ : (A → B) → Delayed A n → Delayed B n

apᵈ : Delayed (A → B) m
    → Delayed A n
    → Delayed B (max m n)
```

runs "in parallel"

```
_>>=ᵈ_ : Delayed A m
       → (A → Delayed B n)
       → Delayed B (m + n)
```

runs sequentially

Encoding applicative via bind changes complexity!

Partiality effect

```
never : Part ⊥
never = fix later

collatz-body : ▹ (ℕ → Part ⊤) → ℕ → Part ⊤
collatz-body c▹ 1 = now tt
collatz-body c▹ n =
  if even n then later (c▹ ⊛ next (n ÷2))
            else later (c▹ ⊛ next (suc (3 · n)))

collatz : ℕ → Part ⊤
collatz = fix collatz-body
```

Wraps potentially non-terminating computations

Conaturals

```
data ℕ∞ : 𝒰 where
  ze : ℕ∞
  su : ▹ ℕ∞ → ℕ∞

infty : ℕ∞
infty = fix su

+-body : ▹ (ℕ∞ → ℕ∞ → ℕ∞) → ℕ∞ → ℕ∞ → ℕ∞
+-body a▹    ze         ze      = ze
+-body a▹ x@(su _)      ze      = x
+-body a▹    ze      y@(su _)   = y
+-body a▹   (su x▹)   (su y▹)   =
      su (next (su (a▹ ⊛ x▹ ⊛ y▹)))

_+_ : ℕ∞ → ℕ∞ → ℕ∞
_+_ = fix +-body
```

Unary numbers extended with numerical infinity

`≅ Part⊤`

Conatural subtraction

```
∸-body : ▹ (ℕ∞ → ℕ∞ → Part ℕ∞) → ℕ∞ → ℕ∞ → Part ℕ∞
∸-body s▹    ze      _      = now ze
∸-body s▹ x@(su _)   ze     = now x
∸-body s▹   (su x▹) (su y▹) = later (s▹ ⊛ x▹ ⊛ y▹)

_∸_ : ℕ∞ → ℕ∞ → Part ℕ∞
_∸_ = fix ∸-body

∸-infty : infty ∸ᶜ infty ≡ never
...
```

(Saturating) subtraction is partial:

∞ ∸ ∞ never terminates

Co/free monad

Partiality is just an instantiation of the free monad with the ▷ functor

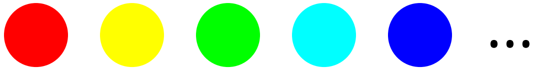
```
data Free (F : 𝒰 → 𝒰) (A : 𝒰) : 𝒰 where
  Pure : A → Free F A
  Roll : F (Free F A) → Free F A

data Cofree (F : 𝒰 → 𝒰) (A : 𝒰) : 𝒰 where
  Cof : A → F (Cofree F A) → Cofree F A
```

- `Free` monad is an F-branching tree with data on the leaves
- `Cofree` comonad is a tree with data at the branches
- What do we get by instantiating `Cofree` with ▷ ?

Streams

```
data Stream (A : 𝒰) : 𝒰 where
  cons : A → ▹ Stream A → Stream A
```

- Another classical infinite structure
- An inductive list with a delayed tail and no empty case
- A lazy linear producer of values



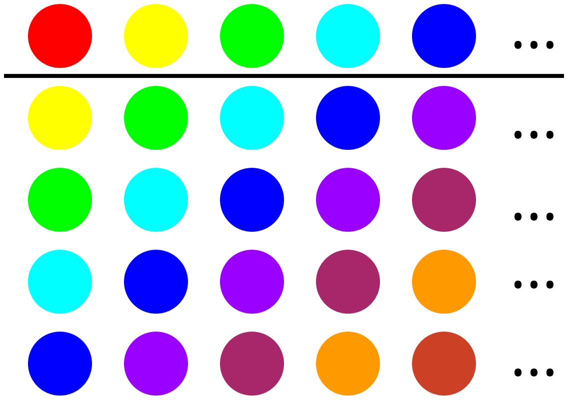
Stream functions

```
headˢ : Stream A → A
headˢ (cons x _) = x

tail▹ˢ : Stream A → ▹ Stream A
tail▹ˢ (cons _ xs▹) = xs▹

repeatˢ : A → Stream A
repeatˢ a = fix (cons a)

mapˢ-body : (A → B)
          → ▹ (Stream A → Stream B)
          →    Stream A → Stream B
mapˢ-body f m▹ as = cons (f (headˢ as)) (m▹ ⊛ (tail▹ˢ as))

mapˢ : (A → B) → Stream A → Stream B
mapˢ f = fix (mapˢ-body f)

natsˢ-body : ▹ Stream ℕ → Stream ℕ
natsˢ-body n▹ = cons 0 (mapˢ suc ⍉ n▹)

natsˢ : Stream ℕ
natsˢ = fix natsˢ-body
```

Stream comonad

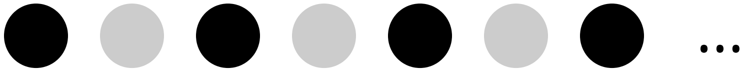
```
extractˢ : Stream A → A
extractˢ = headˢ

duplicateˢ-body : ▹ (Stream A → Stream (Stream A))
                →    Stream A → Stream (Stream A)
duplicateˢ-body d▹ s@(cons _ t▹) = cons s (d▹ ⊛ t▹)

duplicateˢ : Stream A → Stream (Stream A)
duplicateˢ = fix duplicateˢ-body
```

```
extract = head

duplicate = tails
```



Causality

```
stutter : Stream A → Stream A
stutter = fix λ d▹ s →
  cons (headˢ s) (next (cons (headˢ s) (d▹ ⊛ tail▹ˢ s)))

-- everyother : Stream A → Stream A
-- everyother = fix λ e▹ s →
--    cons (headˢ s) (e▹ ⊛ tail▹ˢ (tail▹ˢ s {!!}))
```



- We can define `stutter` but not `everyother`
- Violates casualty
- (can be defined with clocks, however)

Folds and numbers

- We can define other familiar functions
- `foldr`, `scan`, `zipWith`, `interleave`
- numerical streams

```
fibˢ-body : ▹ Stream ℕ → Stream ℕ
fibˢ-body f▹ =
  cons 0 ((λ s → cons 1 $ (zipWithˢ _+_ s) ⍉ (tail▹ˢ s)) ⍉ f▹)

fibˢ : Stream ℕ
fibˢ = fix fibˢ-body

primesˢ-body : ▹ Stream ℕ → Stream ℕ
primesˢ-body p▹ = cons 2 ((mapˢ suc ∘ scanl1ˢ _·_) ⍉ p▹)

primesˢ : Stream ℕ
primesˢ = fix primesˢ-body
```

Part 3

Objects and interfaces

Let's look at the definition of the stream again

```
data Stream (A : 𝒰) : 𝒰 where
  cons : A → ▹ Stream A → Stream A
```

A datatype with a single constructor is essentially a *record*

Iterator

```
record Stream (A : 𝒰) : 𝒰 where
  constructor cons
  field
    hd  : A
    tl▹ : ▹ Stream A
```

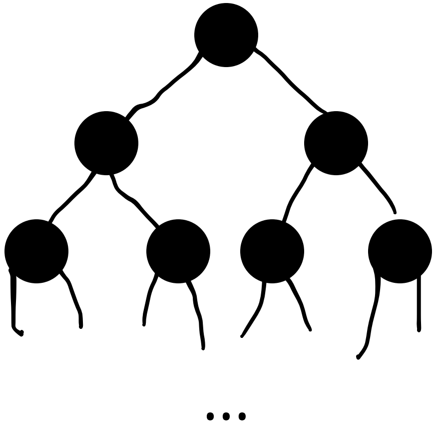
We can treat the stream as an iterator object with two methods:

1. reading the head value
2. advancing by one step

Branching iterator

Can be generalized to an infinite binary tree

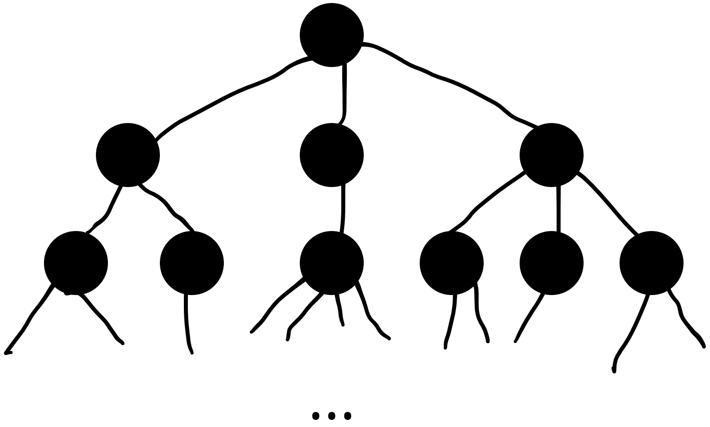
```
data Tree∞ (A : 𝒰) : 𝒰 where
  node : A → ▹ Tree∞ A → ▹ Tree∞ A
       → Tree∞ A

record Tree∞ (A : 𝒰) : 𝒰 where
  constructor node
  field
    val : A
    l▹  : ▹ Tree∞ A
    r▹  : ▹ Tree∞ A
```



Branching iterator

Or a rose tree with arbitrary branching

```
data RTree (A : 𝒰) : 𝒰 where
  rnode : A → List (▹ RTree A) → RTree A

record RTree (A : 𝒰) : 𝒰 where
  constructor rnode
  field
    val : A
    ch▹ : List (▹ RTree A)
```



Terminating iterators

Multiple constructors makes this harder

```
data Colist (A : 𝒰) : 𝒰 where
  cnil  : Colist A
  ccons : A → ▹ Colist A → Colist A

record Colist0 (A : 𝒰) : 𝒰 where
  constructor ccons0
  field
    hd  : Maybe A
    tl▹ : ▹ Colist0 A

record Colist1 (A : 𝒰) : 𝒰 where
  constructor ccons1
  field
    hd    : A
    emp? : Bool
    tl▹   : ▹ Colist1 A
```

Set interface

How would we represent an infinite set of ℕ?

(a finite set is usually some search structure `RedBlackTree ℕ`)

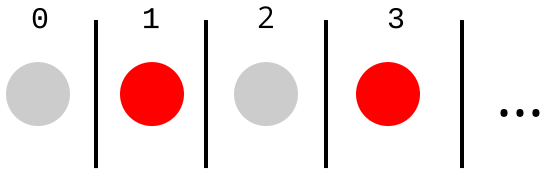
Typically via a function `ℕ → Bool`

We can also use `Stream Bool`

Generally, `Stream A ≅ ℕ → A`

Tabulation of a function

However, this is not very efficient



Set interface

Instead, we can encode a set interface as
a recursive guarded record

```
record Setℕ : 𝒰 where
  constructor mkSet
  field
    emp? : Bool
    has? : ℕ → Bool
    ins  : ℕ → ▹ Setℕ
    uni  : ▹ Setℕ → Part Setℕ
```

(actual implementation a bit more technical)

Strict positivity

- Guarded recursion has two general areas of application:
  1. Working with potentially infinite data structures
  2. Encoding non-strictly-positive recursive types

- **Strictly positive** type appears to the left of 0 arrows
- A syntactic approximation of monotonicity

Strict positivity

- **Strictly positive** type appears to the left of 0 arrows
- Another syntactic approximation
- **Positive** type appears in even positions
- **Negative** type appears in odd positions

```
data Expr : 𝒰 → 𝒰 where
  Foo : ((Expr a → Expr a) → Expr b) → Expr (a → b)
          ^^^^^^----------------------- positive occurrence
                   ^^^^^^-------------- negative occurrence
                             ^^^^^^---- strictly positive occurrence
```

A finite set object

```
finiteSet-body : ▹ (List ℕ → Setℕ) → List ℕ → Setℕ
finiteSet-body f▹ l =
  mkSet (empty? l)
        (λ n → elem? n l)
        (λ n → f▹ ⊛ next (n ∷ l))
        (λ x▹ → later ((λ x →
           foldrP (λ n z →
              later (now ⍉ (z .ins n))) x l) ⍉ x▹))

finiteSet : List ℕ → Setℕ
finiteSet = fix finiteSet-body
```

Carries around the search structure (here a `List`)

An infinite set object

```
evensUnion-body : ▹ (Setℕ → Setℕ) → Setℕ → Setℕ
evensUnion-body e▹ s =
  mkSet false
        (λ n → even n or s .has? n)
        (λ n → e▹ ⊛ s .ins n)
        (λ x▹ → later ((λ f →
           mapᵖ f (s .uni x▹)) ⍉ e▹))

evensUnion : Setℕ → Setℕ
evensUnion = fix evensUnion-body
```

Delegates to the parameter

Objects with IO

This idea can be extended to state and effects

Encode IO as a form of a partiality monad and abstract over methods

```
record IOInt : 𝒰 (ℓsuc 0ℓ) where
  field
    Command : 𝒰
    Response : Command → 𝒰

data IO (I : IOInt) (A : 𝒰) : 𝒰 where
  bnd : (c : Command I) (f : Response I c → ▹ IO I A) → IO I A
  ret : (a : A) → IO I A

record Interface : 𝒰 (ℓsuc 0ℓ) where
  field
    Method : 𝒰
    Result : Method → 𝒰

record IOObj (Io : IOInt) (I : Interface) : 𝒰 where
  field
    mth : (m : Method I) → IO Io (Result I m × IOObj Io I)
```

Part 4

Automata

Let's go back to the idea of function tabulation

`Stream A ≅ ℕ → A`

What is the infinite tree isomorphic to?

```
data Tree∞ (A : 𝒰) : 𝒰 where
  node : A → ▹ Tree∞ A → ▹ Tree∞ A
       → Tree∞ A
```

Word consumers

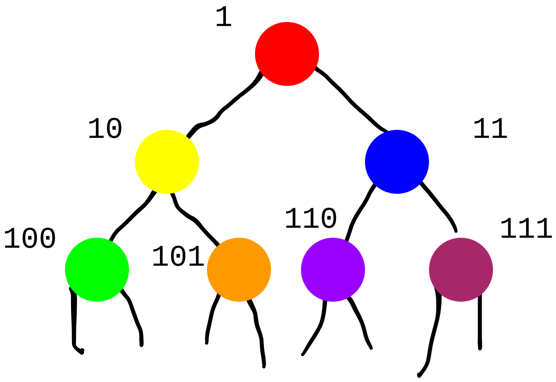
`Tree∞ A ≅ ℕ₂ → A`

(the type of binary numbers)

There's general construction to tabulate

`T → A` into some `F A`

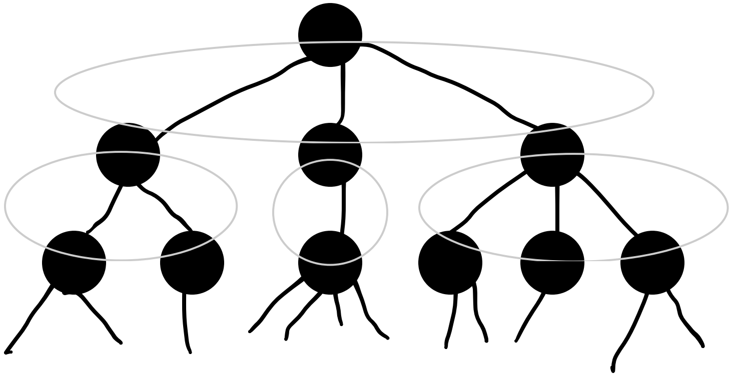
where the structure of F mirrors that of T



...

Tries

We can think of structures as infinite tries whose branching factor is determined by T



...

Word automata

This idea can be generalized even further, to tabulated polymorphic functions:

```
data Stream (A : 𝒰) : 𝒰 where
  cons : A → (T → ▹ Stream A) → Stream A

data Tree∞ (A : 𝒰) : 𝒰 where
  cons : A → (Bool → ▹ Tree∞ A) → Tree∞ A

data Moore (X A : 𝒰) : 𝒰 where
  mre : A → (X → ▹ Moore X A) → Moore X A
```

Word automata

```
data Moore (X A : 𝒰) : 𝒰 where
  mre : A → (X → ▹ Moore X A) → Moore X A
```

`Moore X A ≅ List X → A`

Deterministic Moore automaton, common special case is

`Moore X Bool ≅ List X → Bool`

which is typically called a *recognizer*

Automata operations

```
pure : B → Moore A B
pure b = fix (pure-body b)

map : (B → C)
    → Moore A B → Moore A C
...

ap : Moore A (B → C) → Moore A B → Moore A C
..

zipWith : (B → C → D)
        → Moore A B → Moore A C → gMoore A D
zipWith f = ap ∘ map f

cat : Moore A B → Moore B C → Moore A C
...
```

Regular expressions

```
Lang : 𝒰 → 𝒰
Lang A = Moore A Bool

∅ : Lang A
∅ = pure false

ε : Lang A
ε = mre true λ _ → ∅

char : A → Lang A
char a = Mre false λ x →
           if ⌊ x ≟ a ⌋ then ε else ∅

compl : Lang A → Lang A
compl = map not

_∪_ : Lang A → Lang A → Lang A
_∪_ = zipWith _or_

_∩_ : Lang A → Lang A → Lang A
_∩_ = zipWith _and_
```

Mealy automata

```
data Mealy (X A : 𝒰) : 𝒰 where
  mly : (X → A × ▹ Mealy X A) → Mealy X A

data Moore (X A : 𝒰) : 𝒰 where
  mre : A → (X → ▹ Moore X A) → Moore X A
```

`Mealy X A ≅ Stream X → Stream A`

transducer automaton

Resumptions

```
data Res (I O A : 𝒰) : 𝒰 where
  ret  : A → Res I O A
  cont : (I → O × ▹ Res I O A) → Res I O A
```

- A Mealy automaton that possibly terminates
- a combination of a partiality and state monad

Coroutines

Passing control back and forth via thunks is conceptually programming with coroutines

Can be compiled into patterns of communication between consumer and producer automata

```
data Consume (A B : 𝒰) : 𝒰 where
  end  : B → Consume A B
  more : (A → ▹ Consume A B) → Consume A B

pipe-body : ▹ (Stream A → Consume A B → Part B)
          →    Stream A → Consume A B → Part B
pipe-body p▹ _           (end x)   = now x
pipe-body p▹ (cons h t▹) (more f▹) = later (p▹ ⊛ t▹ ⊛ f▹ h)

pipe : Stream A → Consume A B → Part B
pipe = fix pipe-body
```

Part 5

The road ahead

Where to go next?

Clocked type theory

- We can work "under" thunks but not remove them
- Delayedness never decreases
- Weaker than proper coinduction
- Can be extended by a constant ☐ modality or clock variables to allow forcing "completed" thinks
- `force : (∀ κ → ▹ κ (A κ)) → ∀ κ → A κ`
- Controlled violation of causality
- E.g. we can write `everyother` function on streams

Bird's algorithm

- aka `replaceMin`
- later generalized to value recursion (`MonadFix`) in Haskell
- given a binary tree with data in leaves, replaces all values with a minimum in a single pass

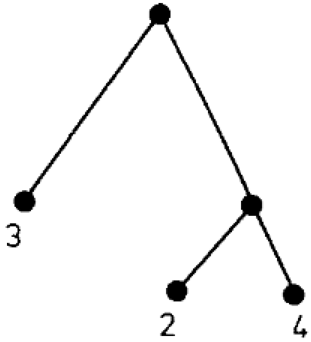


Fig. 1

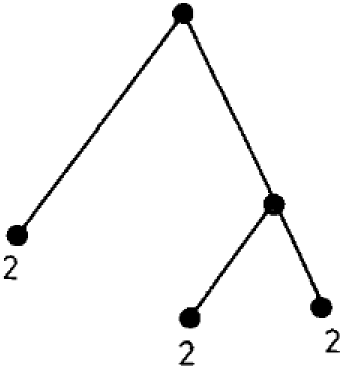


Fig. 2

Bird's algorithm

Classical form is somewhat weird

```haskell
replaceMin :: Tree -> Tree

replaceMin t =
  let (r, m) = rmb (t, m) in r
  where
  rmb :: (Tree, Int) -> (Tree, Int)
  rmb (Leaf x, y) = (Leaf y, x)
  rmb (Node l r, y) =
    let (l',ml) = rmb (l, y)
        (r',mr) = rmb (r, y)
     in
   (Node l' r', min ml mr)
```

Guarded decomposition

- We can decompose this in two temporal phases
- Compute the minimum and construct the thunk
- Then run the thunk
- Uses the feedback combinator

-
  ```
  feedback : (▹ A → B × A) → B
  feedback f = fst (fix (f ∘ ▹map snd))
  ```
- Inserts intermediate data between steps
- Cannot run the thunk without clocks

Continuations

- We can reason about control-flow based algorithms
- Harper's algorithm for matching on regexps with continuations
- Hofmann's algorithm for tree BFS

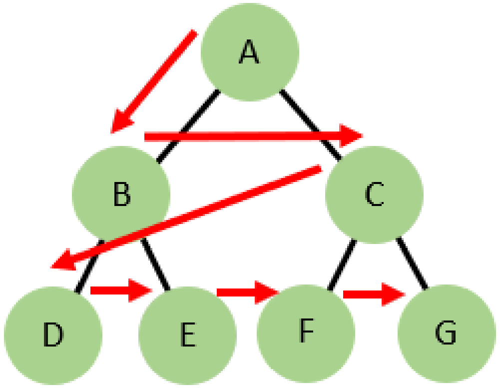
Hoffman's algorithm

- Typically done with queues
- Hoffman invented a purely functional continuation-based algorithm in 1993
- Requires a intermediate (non-strictly) positive datatype

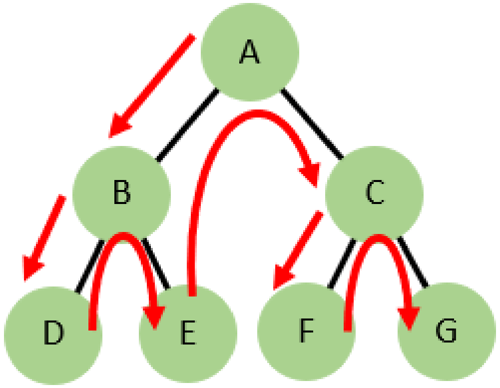
```
data RouF (A : 𝒰) (R▹ : ▹ 𝒰) : 𝒰 where
  overRF : RouF A R▹
  nextRF : ((▸ R▹ → ▹ Colist A) → Colist A) → RouF A R▹

Rou : 𝒰 → 𝒰
Rou A = fix (RouF A)
```

Breadth-first traversal

- Another form of a binary tree, data on both leaves and nodes
- Compute a breadth-first traversal



breadth-first traversal



depth-first traversal

Stream calculus

exact real numbers, series
stream differential equations

$$\frac{1}{1-X} = 1 + X + X^2 + X^3 + \cdots$$

Search algorithms

- sequential topology
- Tychonoff's theorem

```
→c-searchable' : (ds : is-discrete X) → searchable X
               → ((p , d) : d-predicate (Stream X))
               → (δ : ℕ) → δ is-u-mod-of p on (closenessˢ ds)
               → Σ[ s₀ ∶ Stream X ] (Σ (Stream X) p → p s₀)
```

Vs coinduction

- Coinductive mechanisms are more liberal
- Productivity checker is purely syntactic
- Spend a few hours on a proof, get shot down
- Guarded constructions are type-directed

Conclusion

- A principled way to work with non-termination
- A common theme is overcoming syntactic approximations
- Thunks, streams, partiality
- Non-strictly positive datatypes
- Synthetic topology and domain theory
- Concurrency models (quotienting & cubical gizmos)

Working repos

- https://github.com/clayrat/guarded-cm
- https://github.com/clayrat/guarded-termination
- https://github.com/clayrat/guarded-objects
- https://github.com/clayrat/guarded-automata
- https://github.com/clayrat/guarded-search
- https://github.com/clayrat/logrel-guarded

Literature

- Nakano, [2000] "A modality for recursion"
- Artemov, Beklemishev, [2004] "Provability logic"
- https://agda.readthedocs.io/en/latest/language/guarded.html
- Atkey, McBride, [2013] "Productive Coprogramming with Guarded Recursion"
- Bird, [1984] "Using circular programs to eliminate multiple traversals of data"
- Berger, Matthes, Setzer, [2019] "Martin Hofmann's Case for Non-Strictly Positive Data Types"
- Clouston, Bizjak, Grathwohl, Birkedal, [2016] "The guarded lambda-calculus: Programming and reasoning with guarded recursion for coinductive types"
- Paviotti, Mogelberg, Birkedal, [2015] "A model of PCF in Guarded Type Theory"

Contacts

- http://clayrat.github.io/
- https://software.imdea.org/~aliaksandr.hryzlou/
- https://www.linkedin.com/in/alexgryzlov/
- https://twitter.com/clayrat/