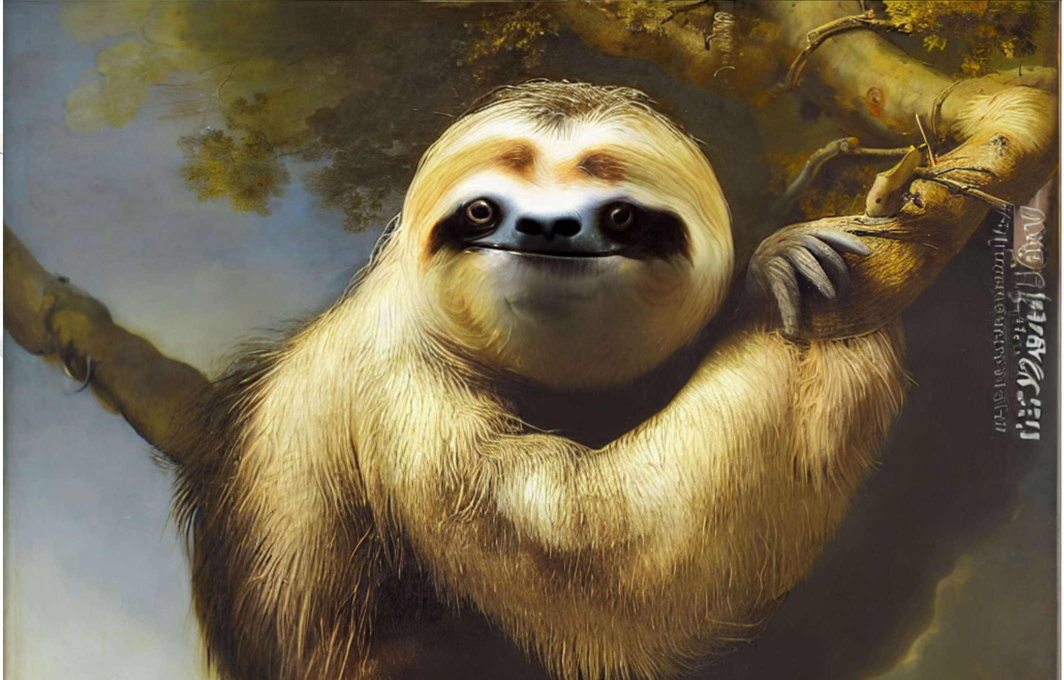
Getting recursive definitions off their bottoms

Joachim @nomeata Breitner

March 17, 2023

BOB, Berlin



Let’s tie a knot!

A famous example

```
fibs :: [Integer]
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

A famous example

```
fibs :: [Integer]
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

A programming puzzle

```
import qualified Data.Map as M

type Graph = M.Map Int [Int]

transitive :: Graph -> Graph
transitive g = ...
```

Let’s step through it

```
transitive graph1

graph1 = M.fromList [(1,[3]),(2,[1,3]),(3,[] )]

transitive g = M.map S.toList reaches
  where
    reaches = M.mapWithKey f g
    f v vs = S.insert v (S.unions [ reaches M.! v’ | v’ <- vs ]))
```

Let's step through it

```
M.map S.toList reaches
  where
    reaches = M.mapWithKey f g
    f v vs = S.insert v (S.unions [ reaches M.! v' | v' <- vs ]))
    g = M.fromList [(1,[3]),(2,[1,3]),(3,[] )]
```

Let's step through it

```
M.map S.toList reaches
  where
    reaches = M.mapWithKey f (M.fromList [(1,[3]),(2,[1,3]),(3,[] )])
    f v vs = S.insert v (S.unions [ reaches M.! v' | v' <- vs ]))
```

Let's step through it

```
M.map S.toList reaches
  where
    reaches = M.fromList [(1,f 1 [3]),(2,f 2 [1,3]),(3,f 3 [] )]
    f v vs = S.insert v (S.unions [ reaches M.! v' | v' <- vs ]))
```

Let's step through it

```
M.map S.toList reaches
  where
    reaches = M.fromList [(1,s1),(2,s2),(3,s3)]
    f v vs = S.insert v (S.unions [ reaches M.! v' | v' <- vs ]))
    s1 = f 1 [3]
    s2 = f 2 [1,3]
    s3 = f 3 [ ]
```

Let's step through it

```
M.map S.toList reaches
  where
    reaches = M.fromList [(1,s1),(2,s2),(3,s3)]

    s1 = S.insert 1 (S.unions [ reaches M.! v' | v' <- [3] ])
    s2 = S.insert 2 (S.unions [ reaches M.! v' | v' <- [1,3] ])
    s3 = S.insert 3 (S.unions [ reaches M.! v' | v' <- [ ]  ])
```

Let's step through it

```
M.map S.toList reaches
  where
    reaches = M.fromList [(1,s1),(2,s2),(3,s3)]

    s1 = S.insert 1 (S.unions [ reaches M.! 3 ])
    s2 = S.insert 2 (S.unions [ reaches M.! 1, reaches M.! 3 ])
    s3 = S.insert 3 (S.unions [] )
```

Let's step through it

```
M.map S.toList reaches
  where
    reaches = M.fromList [(1,s1),(2,s2),(3,s3)]

    s1 = S.insert 1 (S.unions [ s3 ])
    s2 = S.insert 2 (S.unions [ s1, s3 ])
    s3 = S.insert 3 (S.unions [] )
```

Let's step through it

```
M.map S.toList reaches
  where
    reaches = M.fromList [(1,s1),(2,s2),(3,s3)]

    s1 = S.insert 1 (S.unions [ s3 ])
    s2 = S.insert 2 (S.unions [ s1, s3 ])
    s3 = S.fromList [3]
```

Let's step through it

```haskell
M.map S.toList reaches
  where
    reaches = M.fromList [(1,s1),(2,s2),(3,s3)]

    s1 = S.fromList [1,3]
    s2 = S.insert 2 (S.unions [ s1, s3 ])
    s3 = S.fromList [3]
```

Let's step through it

```
M.map S.toList reaches
  where
    reaches = M.fromList [(1,s1),(2,s2),(3,s3)]

    s1 = S.fromList [1,3]
    s2 = S.fromList [1,2,3]
    s3 = S.fromList [3]
```

So far so good...

A vicious cycle

```
transitive graph2

graph2 = M.fromList [(1,[2,3]),(2,[1,2,3]),(3,[3])]

transitive g = M.map S.toList reaches
  where
    reaches = M.mapWithKey f g
    f v vs = S.insert v (S.unions [ reaches M.! v' | v' <- vs ]))
```

A vicious cycle

```
M.map S.toList reaches
  where
    reaches = M.fromList [(1,s1),(2,s2),(3,s3)]

    s1 = S.insert 1 (S.unions [ s2, s3 ])
    s2 = S.insert 2 (S.unions [ s1, s3 ])
    s3 = S.insert 3 (S.unions [] )
```

A vicious cycle

```
M.map S.toList reaches
  where
    reaches = M.fromList [(1,s1),(2,s2),(3,s3)]

    s1 = S.insert 1 (S.unions [ s2, s3 ])
    s2 = S.insert 2 (S.unions [ s1, s3 ])
    s3 = S.insert 3 (S.unions [] )
```

This does not work . . . could it?

The set API

```haskell
import Data.Set as S
data Set a
S.insert  :: Ord a => a -> Set a -> Set a
S.unions  :: Ord a => [Set a] -> Set a
```

The set API

```
import Data.Set as S
data Set a
S.insert  :: Ord a => a -> Set a -> Set a
S.unions  :: Ord a => [Set a] -> Set a

import Data.Recursive.Set as RS
data RSet a
RS.insert :: Ord a => a -> RSet a -> RSet a
RS.unions :: Ord a => [RSet a] -> RSet a
RS.get    :: RSet a -> Set a
```

The set API

```haskell
import Data.Set as S
data Set a
S.insert  :: Ord a => a -> Set a -> Set a
S.unions  :: Ord a => [Set a] -> Set a

import Data.Recursive.Set as RS
data RSet a
RS.insert :: Ord a => a -> RSet a -> RSet a
RS.unions :: Ord a => [RSet a] -> RSet a
RS.get    :: RSet a -> Set a
```

Let's try!

It worked!

(And there are more examples, but not today. . .)

https://hackage.haskell.org/package/rec-def

Solves every set of equations!

```
RS.mk           :: Set a -> RSet a
RS.insert       :: Ord a => a -> RSet a -> RSet a
RS.delete       :: Ord a => a -> RSet a -> RSet a
RS.union        :: Ord a => RSet a -> RSet a -> RSet a
RS.intersection :: Ord a => RSet a -> RSet a -> RSet a
RS.member       :: Ord a => a -> RSet a -> RBool
RB.&&           :: RBool -> RBool -> RBool
```

Solves every set of equations!

```
RS.mk           :: Set a -> RSet a
RS.insert       :: Ord a => a -> RSet a -> RSet a
RS.delete       :: Ord a => a -> RSet a -> RSet a
RS.union        :: Ord a => RSet a -> RSet a -> RSet a
RS.intersection :: Ord a => RSet a -> RSet a -> RSet a
RS.member       :: Ord a => a -> RSet a -> RBool
RB.&&           :: RBool -> RBool -> RBool
```

... because we do not have:

```
RS.difference   :: Ord a => RSet a -> RSet a -> RSet a
RB.not          :: RBool -> RBool
```

So how does it work?

Breaking down the problem

1. A monadic “propagator”
   (declare cells, declare relationships, solves, read values)
2. The pure wrapping
3. Some issues we gloss over today

Breaking down the problem

1. A monadic "propagator"
   (declare cells, declare relationships, solves, read values)
2. The pure wrapping
3. Some issues we gloss over today

Our (simplified) goal:

```
data RSet a
insert :: a -> RSet a -> RSet a
get    :: RSet a -> Set a
```

The propagator – the API

```
data Cell a
newC    :: IO (Cell a)

insertC :: Ord a => Cell a -> a -> Cell a -> IO ()







getC    :: Cell a -> IO (Set a)
```

The propagator – a naive(!) implementation

```
data Cell a = C (IORef (Set a)) (IORef [IO ()])
newC    :: IO (Cell a)
newC = C <$> newIORef S.empty <*> newIORef []
insertC :: Ord a => Cell a -> a -> Cell a -> IO ()
insertC (C s0 ws0) x (C s1 ws1) = do
  let update = do
        new <- S.insert x <$> readIORef s1
        old <- readIORef s0
        unless (old == new) $ do
          writeIORef s0 new
          readIORef ws0 >>= sequence_
  modifyIORef ws1 (update :)
  update
getC    :: Cell a -> IO (Set a)
getC (C s1 _) = readIORef s1
```

The pure wrapper – the API

```
data RSet a

insert :: Ord a => a -> RSet a -> RSet a

get :: RSet a -> Set a
```

The pure wrapper – let's get dirty

```haskell
unsafePerformIO :: IO a -> a
```

A thunking data structure

```haskell
data DoOnce

later :: IO () -> IO DoOnce

doNow :: DoOnce -> IO ()
```

A thunking data structure

```haskell
data DoOnce = DoOnce (IO ()) (IORef Bool)

later :: IO () -> IO DoOnce
later act = DoOnce act <$> newIORef False

doNow :: DoOnce -> IO ()
doNow (DoOnce act done) = do
    is_done <- readIORef done
    unless is_done $ do
        writeIORef done True
        act
```

The pure wrapper – a naive(!) implementation

```haskell
data RSet a = RSet (Cell a) DoOnce

insert :: Ord a => a -> RSet a -> RSet a
insert x r2 = unsafePerformIO $ do
  c1 <- newC
  todo <- later $ do
      let (RSet c2 todo2) = r2
      insertC c1 x c2
      doNow todo2
  return (RSet c1 todo)

get :: RSet a -> Set a
get (RSet c todo) = unsafePerformIO $ do
  doNow todo >> getC c
```

Simplified for your viewing pleasure

- Other data types

  `RBool` with `(RB.&&)` etc.

- Mixing different data types

  ```
  RS.member :: Ord a => a -> RSet a -> RBool
  ```

- Concurrency and reentrancy issues (`unsafePerformIO`!)
- Space leaks (watchers!)

Is this still Haskell?

Queasy about unsafePerformIO?

that is why the function is unsafe.
However "unsafe" is not the same as "wrong". It simply means that the programmer, not the compiler, must undertake the proof obligation that the program's semantics is unaffected [. . .]

"Stretching the Storage Manager: Weak Pointers and Stable Names in Haskell"
Simon Peyton Jones, Simon Marlow, and Conal Elliott

Is this still Haskell?

- Type safety ✓
- Independence of evaluation order ✓

  (At least if the *ascending chain conditions* holds, else unclear.)
- Equational reasoning ✓

```
let x = E1[x] in E2[x]   ≡   let x = E1[x] in E2[E1[x]]
let x = E1[x] in E2[x]   ≡   let x = E1[y]; y = E1[x] in E2[x]
```

Is this still Haskell?

- Type safety ✓
- Independence of evaluation order ✓

  (At least if the *ascending chain conditions* holds, else unclear.)
- Equational reasoning ✓

```
let x = E1[x] in E2[x]   ≡   let x = E1[x] in E2[E1[x]]
let x = E1[x] in E2[x]   ≡   let x = E1[y]; y = E1[x] in E2[x]
```

- Lambda lifting ✗

```
let x = E1[x,e] in E2[x]   ≢   let x y = E1[x y,y] in E2[x e]
```

  Transformations that break *sharing* can prevent termination!

  (So far: can only increase costs, but otherwise unobservable)

Is this still Haskell?

- Type safety ✓
- Independence of evaluation order ✓

  (At least if the *ascending chain conditions* holds, else unclear.)
- Equational reasoning ✓

  ```
  let x = E1[x] in E2[x]   ≡   let x = E1[x] in E2[E1[x]]
  let x = E1[x] in E2[x]   ≡   let x = E1[y]; y = E1[x] in E2[x]
  ```
- Lambda lifting ✗

  ```
  let x = E1[x,e] in E2[x]   ≢   let x y = E1[x y,y] in E2[x e]
  ```

  Transformations that break *sharing* can prevent termination!

  (So far: can only increase costs, but otherwise unobservable)

…and how to prove it?

Summary

- Laziness is key to describing recursive problems declaratively.
- Let us allow more partial orders than Haskell's "normal one"!
- Open question: Is this still pure, and how to prove it?
- Not discussed today:
  The `let x = x` problem, thread safety, avoiding leaks, performance.

Thank you for your attention!

Backup slides

Theory

All involved functions must be *monotone*

For sets:

$$\text{If } s_1 \subseteq s_2 \text{ then } f(s_1) \subseteq f(s_2).$$

For `Bool`:

$$\text{If } b_1 \leq b_2 \text{ then } f(b_1) \leq f(b_2).$$

where `False` $\leq$ `True`.

Finding the least fixed-point

Let X be partially ordered by $\sqsubseteq$, $\bot \in X$ be its least element, and $f : X \to X$ be a continuous function (i.e. $x \sqsubseteq y \implies f(x) \sqsubseteq f(y)$).

Finding the least fixed-point

Let X be partially ordered by $\sqsubseteq$, $\bot \in X$ be its least element, and $f : X \to X$ be a continuous function (i.e. $x \sqsubseteq y \implies f(x) \sqsubseteq f(y)$).

Then the sequence

$$\bot \sqsubseteq f(\bot) \sqsubseteq f(f(\bot)) \sqsubseteq \dots$$

either diverges (all elements are different), or eventually finds a least fixed-point $x \in X$ of f, where

$$x = f(x).$$

Finding the least fixed-point

Let X be partially ordered by $\sqsubseteq$, $\bot \in X$ be its least element, and $f : X \to X$ be a continuous function (i.e. $x \sqsubseteq y \implies f(x) \sqsubseteq f(y)$).

Then the sequence

$$\bot \sqsubseteq f(\bot) \sqsubseteq f(f(\bot)) \sqsubseteq \dots$$

either diverges (all elements are different), or eventually finds a least fixed-point $x \in X$ of f, where

$$x = f(x).$$

If X has the *Ascending Chain Condition* (i.e. no infinite chain $x_0 \sqsubset x_1 \sqsubset \dots$ exists), then the fixed-point will always be found.

It worked!

Let's try another example. . .

A small programming language

```
type Var = String

data Exp
  = Var Var
  | Lam Var Exp
  | App Exp Exp
  | Throw
  | Catch Exp
  | Let Var Exp Exp
```

A small analysis

```
canThrow1 :: Exp -> Bool
canThrow1 = go M.empty where
    go :: M.Map Var Bool -> Exp -> Bool
    go env (Var v)         = env M.! v
    go env Throw           = True
    go env (Catch e)       = False
    go env (Lam v e)       = go (M.insert v False env) e
    go env (App e1 e2)     = go env e1 || go env e2
    go env (Let v e1 e2) = go env' e2 where
        env_bind = M.fromList [ (v, go env e1) ]
        env' = M.union env_bind env
```

Let's add recursion

```
data Exp
  ...
  | LetRec [(Var, Exp)] Exp
```

Let’s add recursion

```
data Exp
  ...
  | LetRec [(Var, Exp)] Exp

canThrow1 :: Exp -> Bool
canThrow1 = go M.empty where
    go :: M.Map Var Bool -> Exp -> Bool
    ...
    go env (LetRec binds e) = go env’ e where
        env_bind = M.fromList [ (v, go env’ e) | (v,e) <- binds ]
        env’ = M.union env_bind env
```

Let's add recursion

```
data Exp
  ...
  | LetRec [(Var, Exp)] Exp

canThrow1 :: Exp -> Bool
canThrow1 = go M.empty where
    go :: M.Map Var Bool -> Exp -> Bool
     ...
    go env (LetRec binds e) = go env' e where
        env_bind = M.fromList [ (v, go env' e) | (v,e) <- binds ]
        env' = M.union env_bind env
```

Again, this fails with cyclic values

```
λ> someVal = Lam "y" (Var "y")
λ> prog = LetRec [("x", App (Var "x") someVal), ("y", Throw)] (Var "x")
λ> canThrow1 prog
^CInterrupted.
```

Data.Recursive.Bool to the rescue!

```
λ> someVal = Lam "y" (Var "y")
λ> prog = LetRec [("x", App (Var "x") someVal), ("y", Throw)] (Var "x")
λ> canThrow1 prog
^CInterrupted.
λ> canThrow2 prog
False
```

Data.Recursive.Bool API

```
import Data.Recursive.Bool as RB

RB.true  :: RBool
RB.false :: RBool
RB.&&    :: RBool -> RBool -> RBool
RB.||    :: RBool -> RBool -> RBool
RB.and   :: [RBool] -> RBool
RB.or    :: [RBool] -> RBool
...
RB.get   :: RBool -> Bool
```

JFR: The full code

```
canThrow2 :: Exp -> Bool
canThrow2 = RB.get . go M.empty where
    go :: M.Map Var RBool -> Exp -> RBool
    go env (Var v)         = env M.! v
    go env Throw           = RB.false
    go env (Catch e)       = RB.true
    go env (Lam v e)       = go (M.insert v RB.false env) e
    go env (App e1 e2)     = go env e1 RB.|| go env e2
    go env (Let v e1 e2) = go env' e2 where
        env_bind = M.singleton v (go env e1)
        env' = M.union env_bind env
    go env (LetRec binds e) = go env' e where
        env_bind = M.fromList [ (v, go env' e) | (v,e) <- binds ]
        env' = M.union env_bind env
```