Molog

Typed Functional Logic Programming in Haskell

Adam C. Foltzer
Galois, Inc.
@acfoltzer

Very funny!

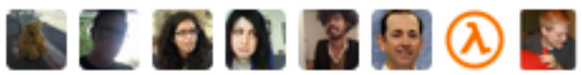
fogus
@fogus

Following

Who's up for KanrenConf?!

Reply Retweeted Favorited More

5 RETWEETS	3 FAVORITES



7:17 AM - 27 Dec 12

- Why typed?
- Straightforward miniKanren embedding
- Checking logic programs with Haskell
- Seemingly-imperative programs
- Lots of challenges

I went to IU...

```
mbp% cat a9.scm
;; Adam C. Foltzer
;; H311 Assignment 9
;; Spring 2009

;; Part I

;; 1. (5)

;; Although called with run2, there is only one answer. The second
;; expression in the inner conde fails, making the first line of the
;; outer conde fail. Most importantly, (== 6 q) always fails.

;; 2. ((_.0 _.1) (5 5) (_.0 (_.0)) (_.0 _.0))

;; The first answer, (_.0 _.1), is the result of the [fail succeed]
;; line, since x and y are not unified with anything in this case, but
;; are still distinct variables. (5 5) is the straightforward result of
;; [(== 5 x) (== y x)] since both x and y wind up associated with
;; 5. (_.0 (_.0)) is the result of [(== `(,x) y)], with x staying
;; fresh, and y being associated with a singleton list containing the
;; value of x. (_.0 _.0) is similar to the first case, except [(== x
;; y)] unifies a single fresh value over both variables. The result of
;; [succeed] would be the same as the first answer, so it is not
;; included.

;; 3. diverges

;; If the second line of the anyo conde is reached, the program
;; diverges.  The first run attempt fails thanks to fail, so the next
;; time through, it diverges.
```

Molog

```
appendo :: Unifiable a
        => Term (List a)
        -> Term (List a)
        -> Term (List a)
        -> Molog ()
appendo xs ys out =
  conde [ do xs  ==@ Nil
             out === ys
        , do x   <- fresh
             xs' <- fresh
             res <- fresh
             xs  ==@ Cons x xs'
             out ==@ Cons x res
             append xs' ys res
        ]
```

So, why typed?

Conor McBride
@pigworker

The question, as ever, is "What are types for, or are they only against?". If types were only an inhibition mechanism, I wouldn't bother.

Chung-chieh Shan
@ccshan

@pigworker Yes, types support program inference by concentrating the (probability) distribution of programs on those that make sense.

Conor McBride
@pigworker

@ccshan I like to think of types as warping our gravity, so that the direction we need to travel becomes "downhill".

So, why Haskell?

Straightforward translation

Straightforward translation

...goals

```
type Goal = Subst → [Subst]
```

Straightforward translation

...goals

```
type Goal = Subst → [Subst]
```

...substitutions

```
type Subst = [(Var, Val)]
```

Straightforward translation

...goals

```
type Goal = Subst → [Subst]
```

...substitutions

```
type Subst = [(Var, Val)]
```

...variables

```
type Var = Int
```

Straightforward translation

...goals

```
type Goal = Subst → [Subst]
```

...substitutions

```
type Subst = [(Var, Val)]
```

...variables

```
type Var = Int
```

... and values

```
data Val = Symbol String
         | Pair Val Val
```

Straightforward translation

```
type Goal = Subst → [Subst]
          ≃ Subst → [((), Subst)]
```

```
type Goal′ a = Subst → [(a, Subst)]
             = StateT Subst [] a
```

Challenge: Extending the domain

```
data Val = Symbol String
         | Number Int
         | Pair Val Val
```

Data types à la carte

```
type MyVal = Fix (String´
                  :+: Int´
                   :+: Pair´
                  )
```

Wouter Swierstra 2008

Using the type system

```
val 5 === val “hello”
```

Typed logic variables

```
type IORef a
```

```
newIORef 5 === newIORef “hello”
```

Koen Claessen, Peter Ljunglöf 2000

Backtracking is tricky

```
writeRef ref newVal k = do
  oldVal <- readIORef
  writeIORef ref newVal
  ans <- k ()
  writeIORef ref oldVal
  return ans
```

IntMaps (like vectors)

```
type IntMap a ≃ [(Int, a)]
```

Oleg's permission

```
unsafeCoerce :: a -> b

type Heap = IntMap Any

writeRef  ≃ insert . unsafeCoerce

readRef   ≃ unsafeCoerce . lookup
```

Challenge: Garbage Collection

Lightweight Monadic Regions?
(Oleg Kiselyov & Chung-chieh Shan 2008)

Wrapping it in a burrito

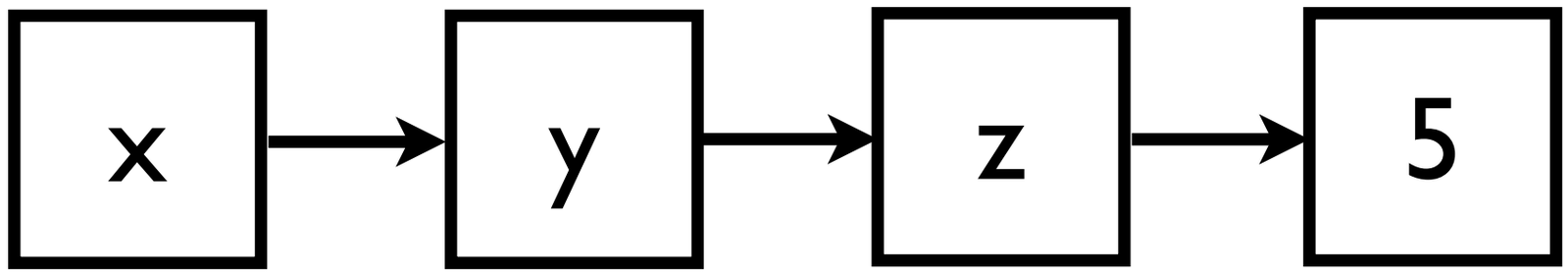
```
-- Kiselyov, Shan, Friedman & Sabry 2005
import Control.Monad.Logic

type STT m a = StateT (IntMap Any) m a
instance Monad m => Monad (STT m)


type Molog a = STT Logic a
instance Monad Molog
```

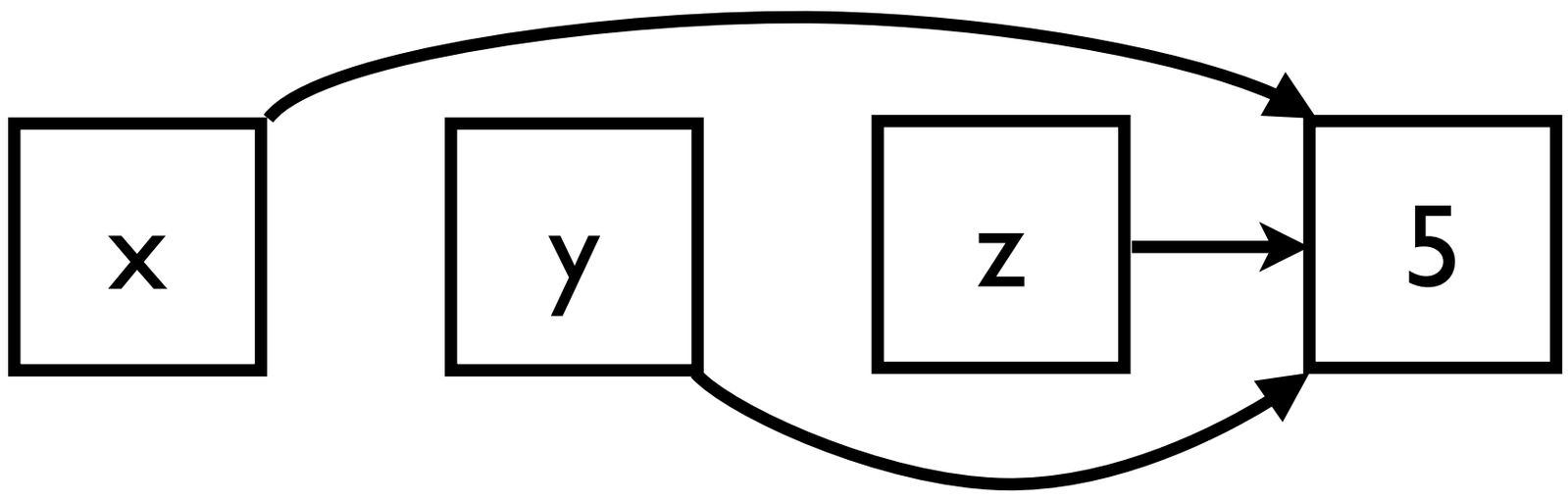
Path compression

```
(fresh (x y z)
  (== z 5)
  (== y z)
  (== x y))
```



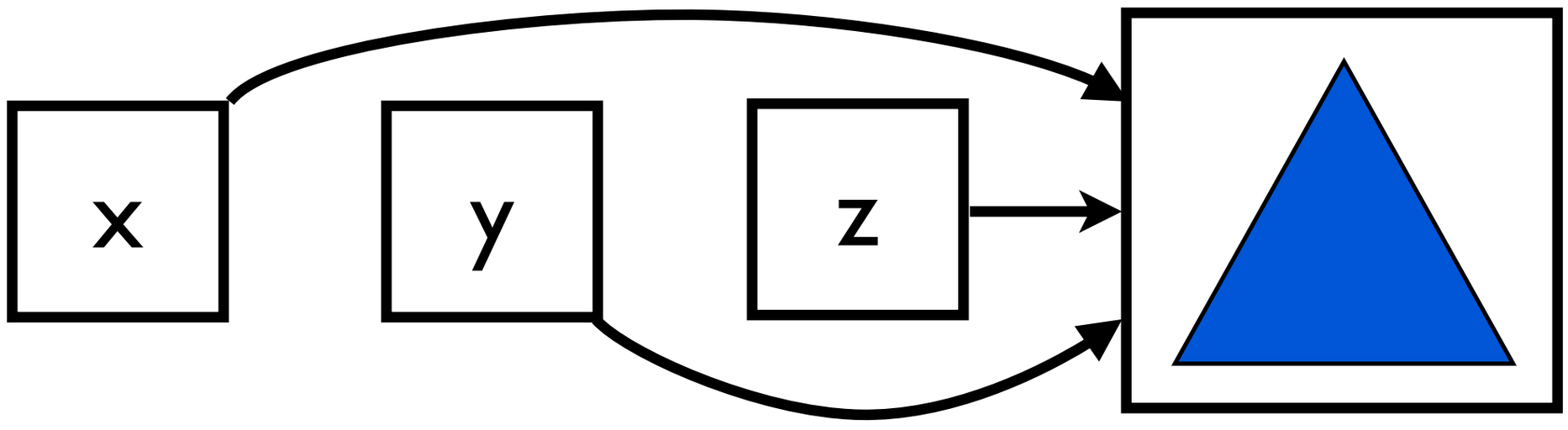
Path compression

```
(fresh (x y z)
  (== z 5)
  (== y z)
  (== x y))
```



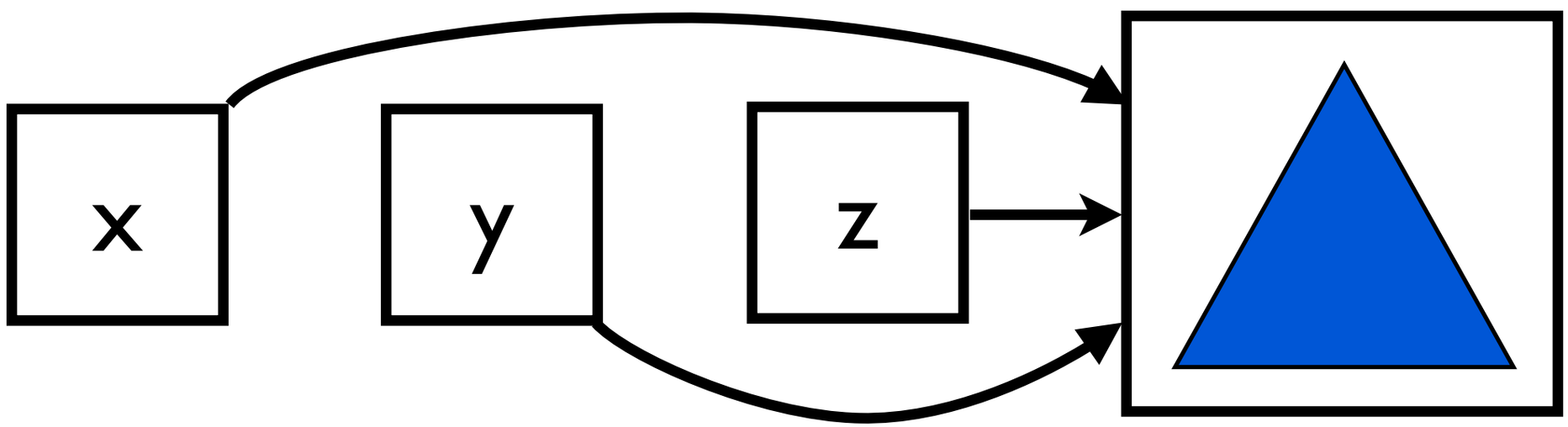
Problem?

```
(fresh (x y z)
  (== z huge-tree)
  (== y z)
  (== x y))
```



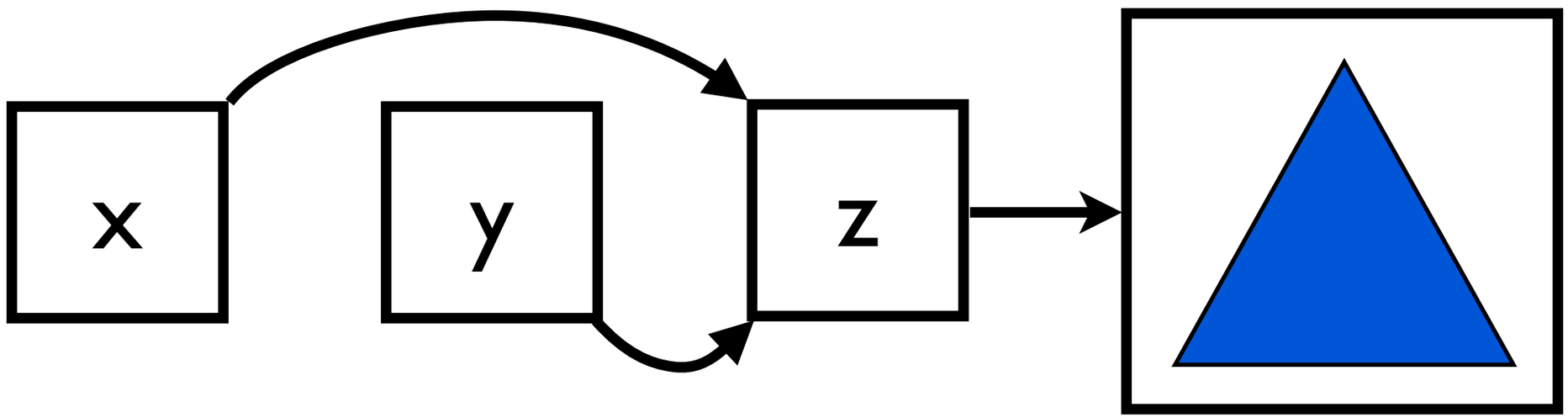
Problem?

```
(fresh (x y z)
  (== z huge-tree)
  (== y z)
  (== x y)
  (== x z))
```



Semi-pruning

```
(fresh (x y z)
  (== z huge-tree)
  (== y z)
  (== x y)
  (== x z))
```



Is this still miniKanren?

- miniKanren goals

```
Subst → [Subst]
```

- Molog computations

```
Molog a ≃ Subst → (a, [Subst])
```

- miniKanren ≃ `Molog ()`

Functional LP

- relational:

```
appendo :: Term [a]
        -> Term [a]
        -> Molog ()
```

- functional:

```
append :: Term [a]
       -> Term [a]
       -> Molog (Term [a])
```

Challenge: Arbitrary types

```
instance Unifiable Exp where
  unify x y = do
    v1 <- lookup x; v2 <- lookup y
    case (v1, v2) of
      (x', y') | x' == y' -> return ()
      (Var id, y') -> semiprune ...
      (x', Var id) -> semiprune ...
      (Val (VarE x), Val (VarE y)) -> unify x y
      (Val (IntE x), Val (IntE y)) -> unify x y
      (Val (BoolE x), Val (BoolE y)) -> unify x y
      (Val (IsZeroE x), Val (IsZeroE y)) -> unify x y
      (Val (PredE x), Val (PredE y)) -> unify x y
      (Val (MultE x1 x2), Val (MultE y1 y2)) ->
        unify x1 y1 >> unify x2 y2
      (Val (IfE xt xc xa), Val (IfE yt yc ya)) ->
        unify xt yt >> unify xc yc >> unify xa ya
      (Val (LamE x xbody), Val (LamE y ybody)) ->
        unify x y >> unify xbody ybody
      (Val (AppE xrator xrand), Val (AppE yrator yrand)) ->
        unify xrator yrator >> unify xrand yrand
      _ -> mzero
```

Challenge: Arbitrary types

- Tim Sheard 2001
- Template Haskell could help
- generic programming to the rescue?

Challenge: Arbitrary types

- Tim Sheard 2001
- Template Haskell could help
- generic programming to the rescue?
- I miss macros

Related languages

- Mercury:

```
:-func fib(int) = int.
fib(N) = (if N =< 2 then 1 else fib(N - 1) + fib(N - 2)).
```

- Curry:

```
fib :: Int -> Int
fib n = if n <= 2 then 1 else fib (n - 1) + fib (n - 2)
```

Fertile grounds

https://github.com/acfoltzer/Molog

https://github.com/acfoltzer/persistent-refs