Write You Some Proofs for Great Good

https://github.com/rpeszek/present-proofs-lc19

https://github.com/rpeszek/present-proofs-lc19/blob/master/doc/slides.pdf

Robert Peszek
r_peszek@hollandhart.com
Holland & Hart LLC
Innovation Lab

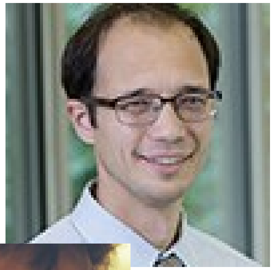
FP @ Innovation Lab, Holland & Hart

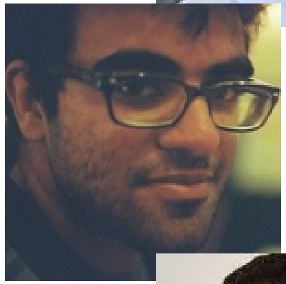
https://www.meetup.com/Boulder-Haskell-Programmers/

Jason Adaska - JWAdaska@hollandhart.com

Nihil Shah - NAShah@hollandhart.com

Robert Peszek - RPeszek@hollandhart.com





Topics

- Precise Types (Vect (n :: Nat) a, MaybeB (b :: Bool) a, Dec)
- Mathematics and Software Engineering
  - Type Precision   - Gentlemen's agreements
  - Progs == Proofs   - Software == has Bugs
  - Dependent Types   - Implementation Hiding
  - Proofs   - Maintenance Cost
- Termination / Totality (can prove nonsense by throwing exception)
- Performance, Maintenance, Peano Nat vs GHC.TypeLits

Proofs by who writes them

	Programmers	Automated Logic Solvers
Amount of Work	😰	Free Lunch!
What can be done	A Lot	Limitted
Implementations	Dependently Typed Languages	Refinement Types LiquidHaskell

```
(!!) :: Vect n a -> Fin n ->
```

(focus of this talk)

```
{-@ (!!) :: x: [a] -> { i:Nat | i < len x}  -> a @-}
(!!) :: [a] -> Int -> a
```

Motivation *(implementation safety)*

(code)

```
data List a =
            Empty
            | Cons a  (List a)
```

vs

```
data Vect (n :: Nat) a  where
   Empty ::  Vect 0 a
   Cons   ::  a -> Vect n a -> Vect (1 + n) a
```

https://github.com/rpeszek/present-proofs-lc19/tree/master/src/Motivation/Exfer.idr

Motivation *(call-side safety)*

```
(!!) :: [a] -> Int -> a
```

```
safeGet :: [a] -> Int -> Maybe a



(!!) :: Vect n a -> Fin n -> a
(!!) :: Vect n a -> SNat m -> MaybeB (m < n) a
(!!) : (xs:List a) -> (n:Nat) -> {auto ok: InBounds n xs} -> a
```

questionable improvement

… still imprecise

Why proofs?

- Curry - Howard (Why proofs == Why programs)
- Formal verification
- Enable `Type Precision` (*focus of this talk*)
  - (proofs replace unsafe coercion)

```
data Vect (n :: Nat) a  where
      Empty ::  Vect 0 a
      Cons :: a -> Vect n a -> Vect (1 + n) a
```

```
id_sanity  ::  Vect (n + 0) a -> Vect n a
id_sanity = unsafeCoerce                         -- ghrrr ☹
```

Curry-Howard to MaybeB

(code)

https://github.com/rpeszek/present-proofs-lc19/blob/master/src/Present/AnIntro.hs

https://github.com/rpeszek/present-proofs-lc19/blob/master/src/Present/MaybeB.hs

Problem in Paradise - Questions

Should type checker know basic algebra?

```
a || True <==> True  (Bool algebra)

a + b == b + a    (Nat, Int, Float ... algebra) ...
```

- Dependently Types Langs: **No** (😰 *Programmers supply proofs*)
- Refinement Types / LiquidHaskell: **Yes** (🤷‍♂️ *SMT solver does the work*)

Type Equality

```
data a :~: b where
    Refl  ::  a :~: a
```

typecheck Refl only if same types

```
test1    = Refl :: 5 :~: 5                -- GOOD
test10  = Refl :: 4 :~: 5                -- ERR

test2    = Refl :: 2 + 3 :~: 3 + 2   -- GOOD
test20 :: SNat n1 -> SNat n2 ->  n1 + n2 :~: n2 + n1
test20 _ _ = Refl                       -- ERR
```

(base) Data.Type.Equality

Type Equality

same for Bool

```
testb1    = Refl :: False :~: False    -- GOOD
testb10  = Refl :: True :~: False     -- ERR

testb2    = Refl ::  (True :|| False) :~: (False :|| True)    -- GOOD

testb20 :: SBool b1 -> SBool b2 ->      (b1 :|| b2) :~: (b2 :|| b1)
testb20 _ _ = Refl                              -- ERR
```

Just reduce to what it already knows!

Example Combinators

- library *over* op semantics
- "pattern-matching on a variable
  of type (a :~: b) produces a proof that a ~ b" - *haddoc*

```
sym :: (a :~: b) -> (b :~: a)          😭
trans :: (a :~: b) -> (b :~: c) -> (a :~: c)
apply :: (f :~: g) -> (a :~: b) -> (f a :~: g b)
inner :: (f a :~: g b) -> (a :~: b)
castWith :: (a :~: b) -> a -> b
gcastWith :: (a :~: b) -> ((a ~ b) => r) -> r
gcastWith Refl x = x
```

math

a bit unexpected ?

fullfil ~ constraint using :~:

[(base) Data.Type.Equality]()

Proofs - Bool Algebra

(code)

https://github.com/rpeszek/present-proofs-lc19/blob/master/src/Present/ProofsBoolAlg.hs

Proofs - Bool Algebra ***(Lessons Learned)***

- (||) implementation is lifted to type level
- case splits define implicit type equality *(definitional equalities)*
- implementation details impact type system behavior

(great for formal verification, sucks for maintenance)

OPINIONS:

→ *Propositions/Proofs live close to implementations*

→ *Avoid implicit use of definitional (b || True = b) equalities - treat all equalities as propositional (lint / type checker plugins?)*

Proofs - Peano Nats

(code)

- Recursive Proofs

- Performance: unsafeCoerce replacements
  - rewrite rules
  - proveFast combinators
  - CPP

- Totality
  - QuickCheck to the rescue

https://github.com/rpeszek/present-proofs-lc19/blob/master/src/Present/ProofsNatAlg.hs

Proofs - working with TypeLits

(code)

- Peano Nat is slow
- TypeLits Nat has hidden implementation - few definitional equalities (good and bad)
- TypeLits can inform us how to make (previous) Peano proofs maintainable

https://github.com/rpeszek/present-proofs-lc19/blob/master/src/Present/WorkingWithTypeLits.hs

Better Bool … Decidability

(code)

```
import Data.Void
data Dec prop = Yes prop |
                No (prop -> Void)
```

- Programming contradictions
  - -XEmptyCase
- Rewards Type Precision
- Programs are proofs!

https://github.com/rpeszek/present-proofs-lc19/blob/master/src/Present/ProofsDecidable.hs

Some Learning/References

- intro books
  - Type Driven Development in Idris great book
    - https://github.com/rpeszek/IdrisTddNotes/wiki
  - TAPL great book *(not really dep types but still)*
  - Programming foundation books (penn/Pierce et al, Wadler)
- Haskell projects (with reading references)
  - singletons
  - equational-reasoning-in-haskell
  - liquidhaskell
- blogs
  - blog.jle.im (Justin Le)
  - typesandkinds (Richard Eisenberg)
- youtube
  - Introduction to Agda series by Daniel Peebles published by Edward Kmett