Stronger Types!

A Brief Introduction to Refinement Types and Dependent Types

Eric Bond
Haskell eXchange 2020

Types!

- Why types?
  - Used to model effects
  - Rule out errors at compile time
  - Increase confidence in program correctness
  - …
- Can Haskell types eliminate all errors?
  - Nope! `head [] -- *** Exception: Prelude.head: empty list`
- How to do better?
  - Proving vs testing

Refinement Types!

- Via Liquid Haskell compiler plugin
- Regular Haskell types annotated with logical predicates
- Logical constraints solved at compile time via an “oracle” (SMT solver)
    - Boolean Expressions
    - Linear Arithmetic
    - Uninterpreted Functions
    - Arrays/Lists
    - Bit Vectors
- Comes with a base library and alternative Prelude

```haskell
{-@ type LessThan5 = { v:Integer | v < 5 } @-}

{-@ ex :: LessThan5 @-}
ex = 8 -- fails to typecheck!
```

```
Liquid Type Mismatch
.
The inferred type
  VV : {v : GHC.Integer.Type.Integer | v == 8}
.
is not a subtype of the required type
  VV : {VV : GHC.Integer.Type.Integer | VV < 5}
```

Example: Head

```
ex = head []
```

Example: Head

```
ex = head []
```

```
Liquid Type Mismatch
.
The inferred type
  VV : {v : [a] | len v == 0
                  && len v >= 0
                  && v == ?a}
.
is not a subtype of the required type
  VV : {VV : [a] | len VV > 0}
.
in the context
  ?a : {?a : [a] | len ?a == 0
                   && len ?a >= 0}

ex = head []
^^^^^^^^^^^^
```

Example: Head - nonEmpty

```haskell
data List a = Nil | Cons a (List a)


{-@ measure nonEmpty @-}
nonEmpty :: List a -> Bool
nonEmpty Nil = False
nonEmpty _ = True


{-@ head :: {xs:List a | nonEmpty xs} -> a @-}
head (Cons x _) = x
```

Example: Head - nonEmpty

```haskell
data List a = Nil | Cons a (List a)


{-@ measure nonEmpty @-}
nonEmpty :: List a -> Bool
nonEmpty Nil = False
nonEmpty _ = True


{-@ head :: {xs:List a | nonEmpty xs} -> a @-}
head (Cons x _) = x
```

```
Liquid Type Mismatch
.
The inferred type
  VV : {v : (AltRef.List a) | (AltRef.nonEmpty v <=> false)
                               && v == ?a}
.
is not a subtype of the required type
  VV : {VV : (AltRef.List a) | AltRef.nonEmpty VV}
.
in the context
  ?a : {?a : (AltRef.List a) | AltRef.nonEmpty ?a <=> false}
|
| ex = head Nil
  ^^^^^^^^^^^^^
```

Example: Head - nonEmpty

```haskell
-- does not typecheck!
doWork :: List a -> a
doWork xs = head xs

-- typechecks!
{-@ doWork' :: {xs:List a | nonEmpty xs} -> a @-}
doWork' xs = head xs

-- typechecks!
doWork'' :: List a -> Maybe a
doWork'' xs | nonEmpty xs = Just $ head xs
doWork'' xs | otherwise   = Nothing
```

Example: Index into List

```
ex = [1,2,3] !! 7
```

Example: Index into List

```haskell
ex = [1,2,3] !! 7
```

```
Liquid Type Mismatch
.
The inferred type
  VV : {v : GHC.Types.Int | v == 7}
.
is not a subtype of the required type
  VV : {VV : GHC.Types.Int | 0 <= VV
                             && VV < len ?d}
.
```

Example: Index into List

```haskell
{-@ safeGet :: xs:[a] -> { i:Int | i >= 0 && i < len xs } -> a @-}
safeGet xs i = xs !! i

safeGet' :: [a] -> Int -> Maybe a
safeGet' xs i
    | i >= 0 && i < length xs = Just $ xs !! i
    | otherwise = Nothing
```

Example: Refine Datatypes

```haskell
data BST a = Leaf
           | Node { root  :: a
                  , left  :: BST a
                  , right :: BST a } deriving Show
```

Example: Refine Datatypes

```haskell
{-@ data BST a    = Leaf
                  | Node { root  :: a
                         , left  :: BST {v:a | v < root}
                         , right :: BST {v:a | root < v} } @-}

data BST a = Leaf
           | Node { root  :: a
                  , left  :: BST a
                  , right :: BST a } deriving Show
```

Example: Refine Datatypes

```
ex = Node 2 (Node 1 Leaf Leaf) (Node 0 Leaf Leaf)
```

Example: Refine Datatypes

```
ex = Node 2 (Node 1 Leaf Leaf) (Node 0 Leaf Leaf)
```

```
Liquid Type Mismatch
.
The inferred type
  VV : GHC.Integer.Type.Integer
.
is not a subtype of the required type
  VV : {VV : GHC.Integer.Type.Integer | 2 < VV}
.
|
|  ex = Node 2 (Node 1 Leaf Leaf) (Node 0 Leaf Leaf)
|                                 ^^^^^^^^^^^^^^^^^^
```

Example: Refine Datatypes

```haskell
insert :: (Ord a) => a -> BST a -> BST a
insert x Leaf = Node x Leaf Leaf
insert x (Node y l r) | x < y  = (Node y (insert x l) r)
insert x (Node y l r) | x > y  = (Node y l (insert x r))
insert x (Node y l r) | x == y = (Node y l r)
```

Example: Refine Datatypes

```haskell
insert :: (Ord a) => a -> BST a -> BST a
insert x Leaf = Node x Leaf Leaf
insert x (Node y l r) | x < y  = (Node y (insert x l) r)
insert x (Node y l r) | x > y  = (Node y l (insert x r))
insert x (Node y l r) | x == y = (Node y r r) -- error here
```

```
**** LIQUID: UNSAFE ************************************

Ref.hs:64:42: error:
    Liquid Type Mismatch
    .
    The inferred type
      VV : {VV : a | y < VV}
    .
    is not a subtype of the required type
      VV : {VV : a | VV < y}
    .
    in the context
      y : a
   |
64 | insert x (Node y l r) | x == y = (Node y r r)
   |                                          ^
```

Type Class Laws?

- Many type classes have laws
- Instances should obey these laws
- Some of these laws can be proven with Refinement Types
- Type Class laws are not *yet* a feature of Liquid Haskell

```
class Monoid a where
    e :: a
    (*) :: a -> a -> a
    -- laws
    associative :: x:a -> y:a -> z:a -> { x * (y * z) == (x * y) * z }
    leftUnit :: x:a -> {e * a == a}
    rightUnit :: x:a -> {a * e == a}
```

Dependent Types!

- Via Agda - a dependently typed programming language
- Stronger core type theory (Haskell core vs Agda core)
- Types can depend on terms

```
{-@ type LessThan3 = { v:Nat | v < 3 } @-}

{-@ ex :: LessThan3 @-}
ex = 4
```

Liquid Haskell

```haskell
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE GADTs #-}
{-# LANGUAGE KindSignatures #-}

data Nat where
    Z :: Nat
    S :: Nat -> Nat

data Fin (n :: Nat) where
    FZ :: Fin ('S n)
    FS :: Fin n -> Fin ('S n)

ex :: Fin ('S ('S ('S 'Z)))
ex = FS (FS (FS (FS FZ)))
```

Haskell

```agda
data ℕ : Set where
  Z : ℕ
  S : ℕ -> ℕ

data Fin : ℕ → Set where
  FZ : {n : ℕ} → Fin (S n)
  FS : {n : ℕ} (i : Fin n) → Fin (S n)

_ : Fin 3
_ = FS (FS (FS (FS FZ)))
```

Agda

Example: Monoid Laws

```haskell
class Monoid a where
    e :: a
    (*) :: a -> a -> a
    associative :: x:a -> y:a -> z:a -> { x * (y * z) == (x * y) * z }
    leftUnit :: x:a -> {e * a == a}
    rightUnit :: x:a -> {a * e == a}
```

Haskell

```agda
record Monoid (A : Set) : Set where
  field
    e : A
    _*_ : A -> A -> A
    associative : ∀ (x y z : A) -> x * (y * z) ≡ (x * y) * z
    leftUnit : ∀ (x : A) -> e * x ≡ x
    rightUnit : ∀ (x : A) -> x * e ≡ x
```

Agda

Example: Monoid Laws

```
ℕ-Monoid : Monoid ℕ
ℕ-Monoid = record {
      e = 0
    ; _*_ = _+_
    ; associative = +-associative
    ; leftUnit = λ x -> refl
    ; rightUnit = λ x -> n+0=n x
  }
```

```
record Monoid (A : Set) : Set where
  field
    e : A
    _*_ : A -> A -> A
    associative : ∀ (x y z : A) -> x * (y * z) ≡ (x * y) * z
    leftUnit : ∀ (x : A) -> e * x ≡ x
    rightUnit : ∀ (x : A) -> x * e ≡ x
```

Example: Monoid Laws

```
ℕ-Monoid : Monoid ℕ
ℕ-Monoid = record {
      e = 0
    ; _*_ = _+_
    ; associative = +-associative
    ; leftUnit = λ x -> refl
    ; rightUnit = λ x -> n+0=n x
  }

n+0=n : ∀ (n : ℕ) -> n + 0 ≡ n
n+0=n 0 = refl
n+0=n (suc m) = cong suc (n+0=n m)

+-associative : ∀ (x y z : ℕ) -> x + (y + z) ≡ (x + y) + z
+-associative 0 _ _ = refl
+-associative (suc x') y z = cong suc (+-associative x' y z)
```

```
record Monoid (A : Set) : Set where
  field
    e : A
    _*_ : A -> A -> A
    associative : ∀ (x y z : A) -> x * (y * z) ≡ (x * y) * z
    leftUnit : ∀ (x : A) -> e * x ≡ x
    rightUnit : ∀ (x : A) -> x * e ≡ x
```

Example: Monad Laws

```
record Monad (F : Set₀ -> Set₀) : Set₁ where
  field
    return : ∀ {A : Set} -> A -> F A
    _>>=_  : ∀ {A B : Set} -> F A -> (A -> F B) -> F B

    leftUnit : ∀ {A B : Set}
                 (a : A)
                 (f : A -> F B)
                   -> (return a) >>= f ≡ f a
    rightUnit : ∀ {A : Set}
                 (m : F A)
                   ->  m >>= return ≡ m
    associative : ∀ {A B C : Set}
                   (m : F A)
                   (f : A -> F B)
                   (g : B -> F C)
                     -> (m >>= f) >>= g ≡ m >>= (λ x -> (f x >>= g))
```

Example: Monad Laws

```
record Monad (F : Set₀ -> Set₀) : Set₁ where
  field
    return : ∀ {A : Set} -> A -> F A
    _>>=_  : ∀ {A B : Set} -> F A -> (A -> F B) -> F B

    leftUnit : ∀ {A B : Set}
                 (a : A)
                 (f : A -> F B)
                   -> (return a) >>= f ≡ f a
    rightUnit : ∀ {A : Set}
                  (m : F A)
                   ->  m >>= return ≡ m
    associative : ∀ {A B C : Set}
                    (m : F A)
                    (f : A -> F B)
                    (g : B -> F C)
                      -> (m >>= f) >>= g ≡ m >>= (λ x -> (f x >>= g))
```

```
data Maybe (A : Set) : Set where
  Nothing : Maybe A
  Just : A -> Maybe A
```

```
Maybe-Monad : Monad Maybe
Maybe-Monad = record {
      return = Just
    ; _>>=_ = λ { Nothing _  -> Nothing
                ; (Just a) f -> f a }
    ; leftUnit = λ a -> λ f -> refl
    ; rightUnit = λ { Nothing  -> refl
                    ; (Just a) -> refl }
    ; associative = λ { Nothing f g  -> refl
                      ; (Just a) f g -> refl }
  }
```

Overall

Refinement Types

Pros:

- More automated
- Easier to use

Cons:

- Type refinements are limited to what the SMT “oracle” knows

Dependent Types

Pros:

- Extremely expressive

Cons:

- Not native in Haskell
- Much more manual

Where To Learn More!

- Liquid Haskell Book

  https://ucsd-progsys.github.io/liquidhaskell-tutorial/

- Programming Language Foundations in Agda

  https://plfa.github.io/

- Lambda Pi - A Tutorial Implementation of a Dependently Typed Lambda Calculus

  https://www.andres-loeh.de/LambdaPi/LambdaPi.pdf

- Programming Z3 - An SMT Solver Tutorial

  https://theory.stanford.edu/~nikolaj/programmingz3.html

- Why Types - A 47 Degrees Blog on Dependent Types and Refinement Types

  https://www.47deg.com/blog/why-types/

- Scala Stainless - Refinement Types for Scala

  https://stainless.epfl.ch/