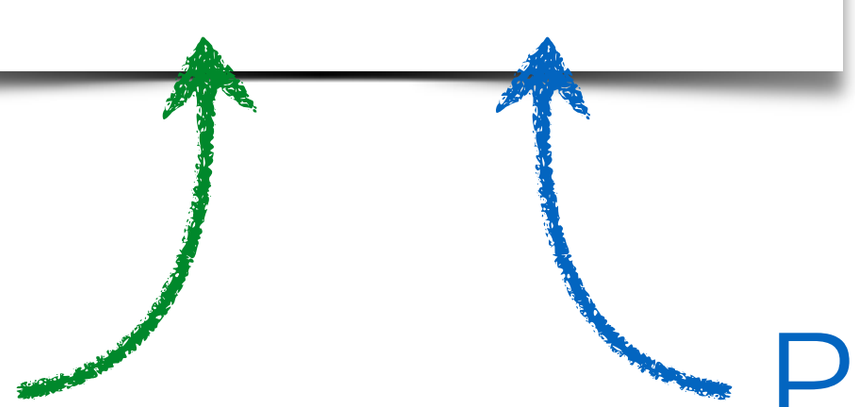
Bounded Refinement Types

Niki Vazou,
Alexander Bakst,
Ranjit Jhala
(*UC San Diego*)

Refinement Types

```
{v:Int | v>0}
```



Basic Type

Predicate

LiquidHaskell:

Refinement types to express specifications

Choice:

Refinements drawn from decidable logic

Question:

How to enhance expressiveness?

Question:

How to enhance expressiveness?

Can we specify function composition?

Can we specify function composition?

```
incr    :: x:Int -> {v| v = x+1}
incr x = x+1
```

```
incr2 :: x:Int -> {v| v = x+2}
incr2 = compose incr incr
```

Can we specify function composition?

Type is too specific

```
compose ::
          (y:b-> {z:c|z = y+1})
       -> (x:a-> {z:b|y = x+1})
       -> (x:a-> {z:c|z = x+2})
compose f g x = f (g x)
```

```
incr2 :: x:Int -> {v| v = x+2}
incr2 = compose incr incr
```

OK

Can we specify function composition?

Type is too specific

```
compose ::
          (y:b-> {z:c|z = y+1})
       -> (x:a-> {z:b|y = x+1})
       -> (x:a-> {z:c|z = x+2})
compose f g x = f (g x)
```

```
incr2 :: x:Int -> {v| v = x+2}
incr2 = compose decr incr
```

Fail

Can we specify function composition?

Type is too specific

```
compose ::
          (y:b-> {z:c|z = y+1})
       -> (x:a-> {z:b|y = x+1})
       -> (x:a-> {z:c|z = x+2})
compose f g x = f (g x)
```

```
incr2 :: x:Int -> {v| v = x+2}
incr2 = compose incr incr
```

Can we specify function composition?

```
p, q |-> \x z->z=x+1
r    |-> \x z->z=x+2
```

```
compose ::
          (y:b-> {z:c|p y z})
       -> (x:a-> {y:b|q x y})
       -> (x:a-> {z:c|r x z})
compose f g x = f (g x)
```

```
incr2 :: x:Int -> {v| v = x+2}
incr2 = compose incr incr
```

Can we specify function composition?

Type is wrong

```
compose ::
          (y:b-> {z:c|p y z})
       -> (x:a-> {y:b|q x y})
       -> (x:a-> {z:c|r x z})
compose f g x = f (g x)
```

Can we specify function composition?

Type is wrong

```
compose ::
          (y:b-> {z:c|p y z})
       -> (x:a-> {y:b|q x y})
       -> (x:a-> {z:c|r x z})
compose f g x = let y = g x in f y
```

```
y:{y:b|q x y} ⊢ {z:c|p y z} <: {z:c|r x z}
```

Can we specify function composition?

Type is wrong

```
compose ::
          (y:b-> {z:c|p y z})
       -> (x:a-> {y:b|q x y})
       -> (x:a-> {z:c|r x z})
compose f g x = let y = g x in f y
```

```
y:{y:b|q x y} ⊢ {z:c|p y z} <: {z:c|r x z}
```

Can we specify function composition?

Type is wrong

```
compose ::
          (y:b-> {z:c|p y z})
       -> (x:a-> {y:b|q x y})
       -> (x:a-> {z:c|r x z})
compose f g x = let y = g x in f y
```

```
y:{y:b|q x y} ⊢ {z:c|p y z} <: {z:c|r x z}
```

Can we specify function composition?

Type is wrong

```
compose ::
          => (y:b-> {z:c|p y z})
          -> (x:a-> {y:b|q x y})
          -> (x:a-> {z:c|r x z})
compose f g x = let y = g x in f y
```

```
bound Chain p q r = \x y z ->
  q x y => p y z => r x z
```

Can we specify function composition?

```
compose :: (Chain p q r)
        => (y:b-> {z:c|p y z})
        -> (x:a-> {y:b|q x y})
        -> (x:a-> {z:c|r x z})
compose f g x = let y = g x in f y
```

OK

```
bound Chain p q r = \x y z ->
  q x y => p y z => r x z
```

Can we specify function composition?

```
p, q |-> \x z->z=x+1
r    |-> \x z->z=x+2
```

```
incr2 :: x:Int -> {v| v = x+2}
incr2 = compose incr incr
```

```
bound Chain p q r = \x y z ->
   q x y => p y z => r x z
```

Can we specify function composition?

```
p, q |-> \x z->z=x+1
r    |-> \x z->z=x+2
```

```
incr2 :: x:Int -> {v| v = x+2}
incr2 = compose incr incr
```

OK

```
bound Chain = \x y z ->
  y=x+1 => z=y+1 => z=x+2
```

Valid

Can we specify function composition?

Bounds let us specify function composition

Do bounds add complexity?

No. Bounds are desugared to unbounded types

Bounds are desugared to unbounded types

```
compose :: (Chain p q r)
        => (y:b-> {z:c|p y z})
        -> (x:a-> {y:b|q x y})
        -> (x:a-> {z:c|r x z})
compose f g x =
  let y = g x in
  let z = f y in z
```

```
bound Chain p q r = \x y z ->
   q x y => p y z => r x z
```

Bounds are desugared to unbounded types

```
compose :: $chain:(tchain p q r)
        -> (y:b-> {z:c|p y z})
        -> (x:a-> {y:b|q x y})
        -> (x:a-> {z:c|r x z})
compose $chain f g x =
  let y = g x in
  let z = f y in z
```

```
type tchain p q r = x:a -> y:b -> z:c ->
   {v|q x y => p y z => r x z}
```

Bounds are desugared to unbounded types

```
compose :: $chain:(tchain p q r)
        -> (y:b-> {z:c|p y z})
        -> (x:a-> {y:b|q x y})
        -> (x:a-> {z:c|r x z})
compose $chain f g x =
  let y = g x in
  let z = f y in
  let _ = $chain x y z in z
```

OK

```
type tchain p q r = x:a -> y:b -> z:c ->
   {v|q x y => p y z => r x z}
```

Bounds are desugared to unbounded types

```
p, q |-> \x z->z=x+1
r    |-> \x z->z=x+2
```

```
incr2 :: x:Int -> {v| v = x+2}
incr2 = compose $chain incr incr
 where $chain :: (tchain p q r)
       $chain =   ???
```

```
type tchain p q r = x:a -> y:b -> z:c ->
  {v|q x y => p y z => r x z}
```

Bounds are desugared to unbounded types

```
p, q |-> \x z->z=x+1
r    |-> \x z->z=x+2
```

```
incr2 :: x:Int -> {v| v = x+2}
incr2 = compose $chain incr incr
 where $chain :: tchain
       $chain =   ???
```

```
type tchain     = x:a -> y:b -> z:c ->
  {v|y=x+1 => z=y+1 => z=x+2}
```

Bounds are desugared to unbounded types

```
p, q |-> \x z->z=x+1
r    |-> \x z->z=x+2
```

```
incr2 :: x:Int -> {v| v = x+2}
incr2 = compose $chain incr incr
 where $chain :: tchain
       $chain =   ???
```

```
type tchain     = x:a -> y:b -> z:c ->
  {v|true}
```

Bounds are desugared to unbounded types

```
p, q |-> \x z->z=x+1
r    |-> \x z->z=x+2
```

```
incr2 :: x:Int -> {v| v = x+2}
incr2 = compose $chain incr incr
 where $chain :: tchain
       $chain =   ???
```

```
type tchain     = x:a -> y:b -> z:c ->
  {v|true}
```

Bounds enhance expressiveness

Do bounds add complexity?

No. Bounds are desugared to unbounded types

Are bounds useful?

Function Composition

List Filtering and List Folding

List Filtering

```
filter :: (Witness p w)
       => (x:a-> {v:Bool|w x v})
       -> [a] -> [{v:a|p v}]
filter q (x:xs)
  | q x          = x : filter q xs
  | otherwise = filter q xs
filter _ []   = []
```

```
bound Witness p w = \x b ->
   b => w x b => p x
```

List Filtering

```
isPos :: x:Int -> {v|v <=> 0<x}

ex  :: x:[Int] -> [{v|0<v}]
ex = filter isPos
```

```
p -> \x -> 0 < x
w -> \x b -> b <=> 0 < x
```

```
bound Witness p w = \x b ->
  b => w x b => p x
```

List Filtering

```
isPos :: x:Int -> {v|v <=> 0<x}

ex  :: x:[Int] -> [{v|0<v}]
ex = filter isPos
```

```
  p -> \x -> 0 < x
  w -> \x b -> b <=> 0 < x
```

```
bound Witness = \x b ->
  b => (b <=> 0 < x ) => 0 < x
```

Valid

Bounds enhance expressiveness

Do bounds add complexity?

No. Bounds are desugared to unbounded types

Are bounds useful?

Function Composition

List Filtering and List Folding

List Folding

```
foldr :: (Inductive inv step)
 => (x:a -> acc:b -> {v:b|step x acc v})
 -> {v:b|inv [] v} -> xs:[a]
 -> {v:b|inv xs v}
foldr f b (x:xs) = f x (foldr op b xs)
foldr f b []     = b
```

```
bound Inductive inv step = \x xs b b'->
   inv xs b => step x b b'
            => inv (x:xs) b'
```

List Folding

```
incr :: x:Int -> {v|v = x+1 }

ex   :: xs:[a] -> {v:Int |v = len xs}
ex = foldr (\x -> incr) 0
```

```
  inv  -> \xs b    -> b = len xs
  step -> \x b b' -> b' = b + 1
```

```
bound Inductive inv step = \x xs b b'->
  inv xs b => step x b b'
           => inv (x:xs) b'
```

List Folding

```
incr :: x:Int -> {v|v = x+1 }

ex   :: xs:[a] -> {v:Int |v = len xs}
ex = foldr (\x -> incr) 0
```

```
  inv  -> \xs b    -> b = len xs
  step -> \x b b' -> b' = b + 1
```

```
bound Inductive = \x xs b b'->
  b = len xs => b' = b + 1
             => b' = len (x:xs)
```

Valid

Bounds enhance expressiveness

Do bounds add complexity?

No. Bounds are desugared to unbounded types

Are bounds useful?

Function Composition

List Filtering and List Folding

Floyd-Hoare Logic in the State monad

Relational DataBases

Thank you!

END