Bounded Refinement Types

Niki Vazou, Alexander Bakst, Ranjit Jhala

UC San Diego

Given: Library

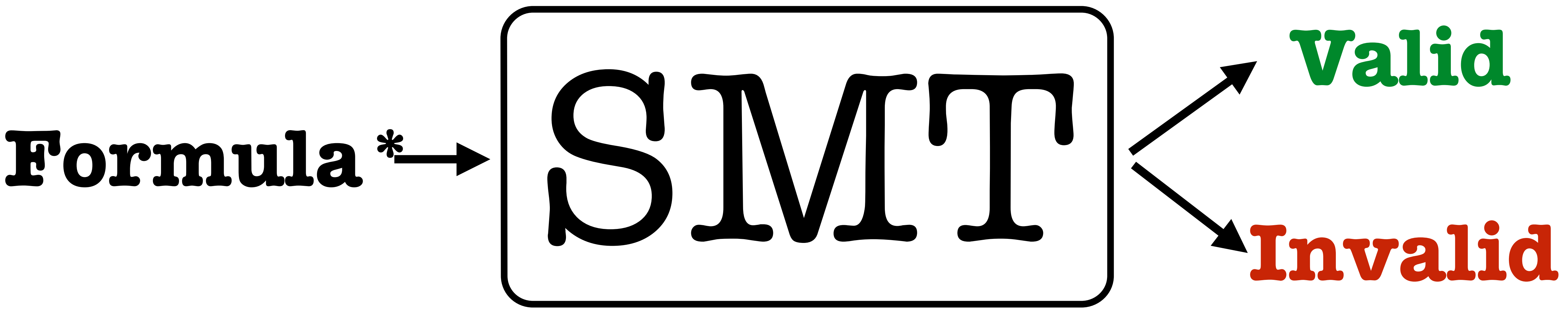
```
incr2 x = x + 2
```

Goal: Verify Client

```
p = incr2 40
assert (p == 42)

n = incr2 (-50)
assert (n < 0)
```

Automatic Verification via SMT



* From **Decidable** Logic

Specify

```
incr2 x = x + 2
```

Refinement Type

```
x:Int -> {r:Int | r = x+2}
```

… & Verify

```
p = incr2 40
assert (p == 42)
```

Via Type

```
p = 40 + 2
```

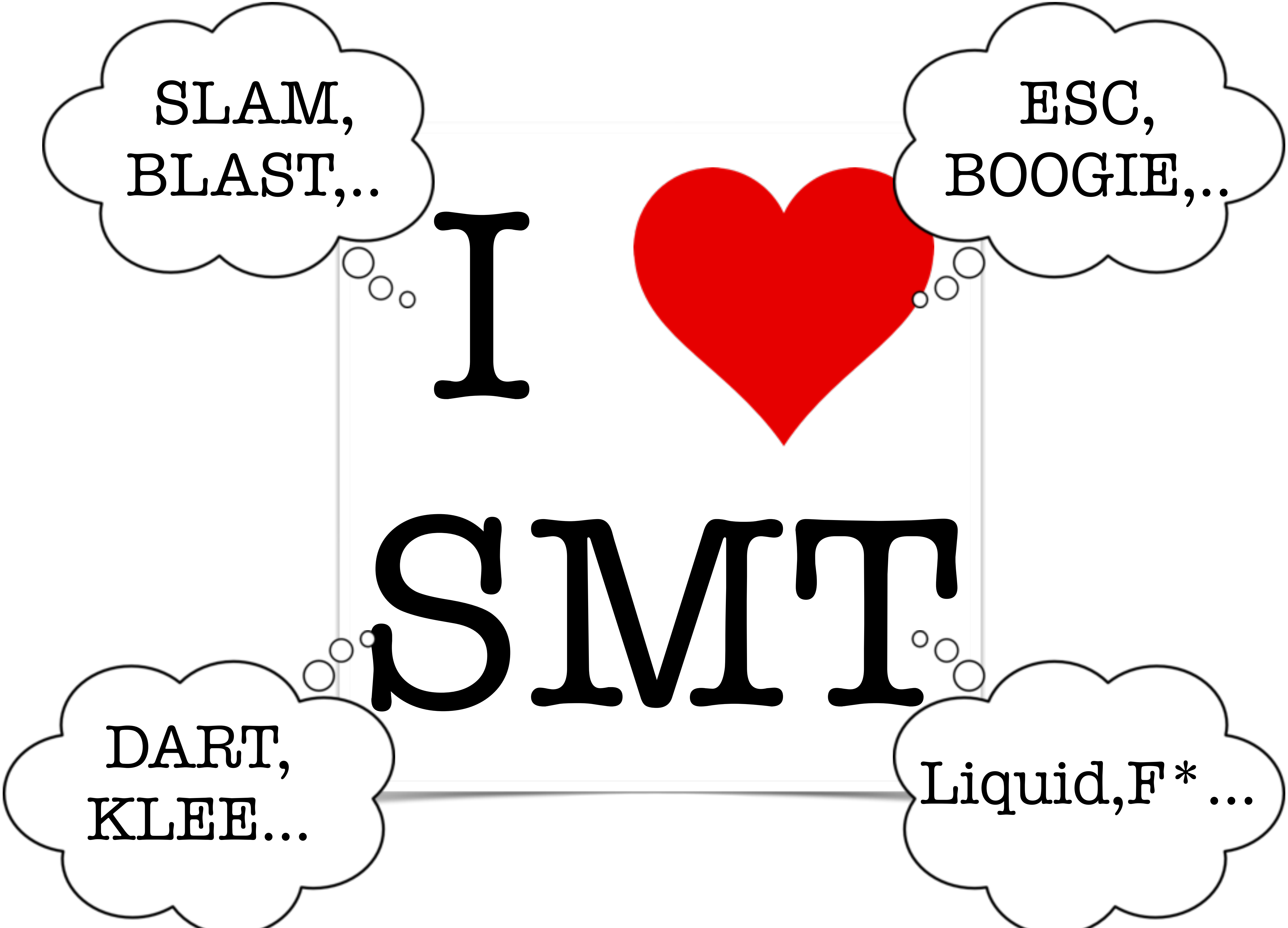
… & Verify

```
p = incr2 40
assert (p == 42)
```

SMT: Is Valid? Yes!

```
p = 40 + 2 => p = 42
```

Automatic Verification via SMT


SLAM, BLAST,..
ESC, BOOGIE,..
I ♥ SMT
DART, KLEE...
Liquid,F* ...

I have a problem with SMT

Decidable Logic

↓

Quantifier-Free, First-Order Logic

↓

Specifications Not “Modular”

Goal: Verify Client

```
f = compose (\a -> a + 1)
            (\b -> b + 1)
p = f 40
assert (p == 42)
```

Given: Library

```
compose f g x = f (g x)
```

Specification?

Given: Library

```
compose f g x = f (g x)
```

First-Order Specification

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

Not Modular

Given: Library

```
compose f g x = f (g x)
```

Higher-Order Specification

```
ensures \forall f g x. r = f (g x)
```

Not Decidable

Problem

Automatic Verification vs. **Modular** Specifications

```
y = x + 1
```

```
r = f (g x)
```

Goal

Automatic Verification of **Modular** Specifications

Key Idea

Observe: Refinements are Relations

```
x:Int-> {y:Int| y = x+2 }
```

Relation between input x and output y

Key Idea

Observe: Refinements are Relations

Step 1: Abstract Relations

```
x:Int-> {y:Int| y = x+2 }
```

Key Idea

Observe: Refinements are Relations

Step 1: Abstract Relations

```
x:Int-> {y:Int| p x y }
```

In SMT p is “Uninterpreted Function”

Key Idea

Step 1: Abstract Relations

```
x:Int-> {y:Int| p x y }
```

In SMT p is “Uninterpreted Function”

$$\forall \overline{x},\ \overline{y}.\ \overline{x} = \overline{y} \Rightarrow (p\ \overline{x}) = (p\ \overline{y})$$

Key Idea

Observe: Refinements are Relations

Step 1: Abstract Relations

Step 2: Constrain Relations

To specify properties of abstract p

Step 1: Abstract Relations

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

Step 1: Abstract Relations

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

Step 1: Abstract Relations

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

Wait! Is this specification correct?

Wait! Is this specification correct?

```
compose :: forall p, q, r.
             f:(y:b-> {z:c|p y z})
          -> g:(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
```

Wait! Is this specification correct?

```
compose :: forall p. q. r.
             f:(y:b-> {z:c|p y z})
          -> g:(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
```

Assume

Prove

Wait! Is this specification correct?

```
compose :: forall p, q, r.
              f:(y:b-> {z:c|p y z})
           -> g:(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
```

Assume

Prove

Wait! Is this specification correct?

```
compose :: forall p, q, r.
              f:(y:b-> {z:c|p y z})
           -> g:(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
```

Assume

Prove

Wait! Is this specification correct?

```
compose :: forall p, q, r.
            f:(y:b-> {z:c|p y z})
         -> g:(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
```

Assume	Prove
`y:{y:b	q x y}`

Wait! Is this specification correct?

```
compose :: forall p, q, r.
             f:(y:b-> {z:c|p y z})
          -> g:(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
```

Assume

Prove

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

Wait! Is this specification correct?

```
compose :: forall p, q, r.
              f:(y:b-> {z:c|p y z})
           -> g:(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
```

Assume

y:{y:b|q x y}

z:{z:c|p y z}

Prove

z:{z:c|r x z}

Wait! Is this specification correct?

```
compose :: forall p, q, r.
              f:(y:b-> {z:c|p y z})
           -> g:(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
```

Assume

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

Prove

```
z:{z:c|r x z}
```

Wait! Is this specification correct?

```
compose :: forall p, q, r.
              f:(y:b-> {z:c|p y z})
           -> g:(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
```

Is SMT Valid ? No!

$$q\ x\ y \ \wedge\ p\ y\ z \Rightarrow r\ x\ z$$

Specification is too general!

Key Idea

Step 1: Abstract Relations

Specification is too general!

Need: $q\ x\ y\ \wedge\ p\ y\ z \Rightarrow r\ x\ z$

Key Idea

Step 1: Abstract Relations

Specification is too general!

Step 2: Constrain Relations

Bound: $q\ x\ y \ \wedge\ p\ y\ z \Rightarrow r\ x\ z$

Key Idea

Step 1: Abstract Relations

Specification is too general!

Step 2: Constrain Relations

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

Step 2: Constrain Relations

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

Type (Bad)

```
compose :: forall p, q, r.
              (y:b-> {z:c|p y z})
           -> (x:a-> {y:b|q x y})
           -> (x:a-> {z:c|r x z})
```

Step 2: Constrain Relations

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

Type (Fixed)

```
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})
```

Step 2: Constrain Relations

```
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
```

Assume

q x y

p y z

Prove

r x z

Step 2: Constrain Relations

```
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
```

Assume	Prove
`q x y`	`r x z`
`p y z`	
`Chain p q r`	

Step 2: Constrain Relations

```
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
```

Is SMT Valid ? Yes(!)

$$
\begin{aligned}
& q\ x\ y \\
\wedge\ & p\ y\ z \\
\wedge\ & (q\ x\ y \wedge p\ y\ z \Rightarrow r\ x\ z) \Rightarrow\ r\ x\ z
\end{aligned}
$$

Step 2: Constrain Relations

```
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
```

Verification is **Decidable** ...

Step 2: Constrain Relations

```
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
```

... but is Specification **Modular**?

Next: Lets's Verify a Client

Client Verification

```
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})
```

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

```
incr2 :: x:Int -> {v:Int |v = x+2}
incr2 = compose (+1) (+1)
```

Client Verification

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

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

```
incr2 :: x:Int -> {v:Int |v = x+2}
incr2 = compose (+1) (+1)
```

Client Verification

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

Valid

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

```
incr2 :: x:Int -> {v:Int |v = x+2}
incr2 = compose (+1) (+1)
```

OK

Key Idea

Observe: Refinements are Relations

Step 1: Abstract Relations @ Lib

Step 2: Constrain Relations @ Lib

Step 3: Instantiate Relations @ Clt

Goal

Automatic Verification of Modular Specifications

Goal

Automatic Verification of Modular Specifications

Key Idea

Constrained Abstract Refinements

Goal

Automatic Verification of Modular Specifications

Key Idea

Constrained Abstract Refinements

Applications

Applications

Higher-Order Functions

Database Schema

Floyd-Hoare Logic

Capability Safe IO Monad

Higher-Order Functions

Examples

compose, foldr, filter, ...

Higher-Order Functions

Examples

compose, **foldr**, filter, ...

```
foldr f b (x:xs) = f x (foldr op b xs)
foldr f b []     = b
```

Specification

```
foldr f b (x:xs) = f x (foldr op b xs)
foldr f b []     = b
```

`inv xs b` := list `xs` and value `b`

`stp x b b'` := inputs and output of `f`

```
bound Ind inv stp = \x xs b b'->
  inv xs b ⇒ stp x b b'
           ⇒ inv (x:xs) b'
```

Specification

```
bound Ind inv stp = \x xs b b'->
   inv xs b ⇒ stp x b b'
             ⇒ inv (x:xs) b'
```

Refinement Type

```
foldr :: (Ind inv stp)
      => (x:a -> b:b -> {b':b|stp x b b'})
      -> {b:b|inv [] b} -> xs:[a]
      -> {v:b|inv xs v}
```

Applications

Higher-Order Functions

Database Schema

Floyd-Hoare Logic

Capability Safe IO Monad

Applications

Higher-Order Functions

Database Schema and Queries

Floyd-Hoare Logic

Capability Safe IO Monad

Database Schema and Queries

Modular Specifications

select, project, join, …

Abstract Refinements

Describe generic key-value relationships

Bounds

Describe schema disjointness, union,…

Floyd-Hoare Logic (in ST)

Modular Specifications

sequence, if, while, for, ...

Abstract Refinements

Describe generic state assertions

Bounds

Describe Floyd-Hoare Constraints

Applications

Higher-Order Functions

Database Schema

Floyd-Hoare Logic (in ST)

Capability Safe IO Monad

Goal

Automatic Verification of Modular Specifications

Key Idea

Constrained Abstract Refinements

Applications

HOFs, Databases, Floyd-Hoare, Capabilities.

Goal

Automatic Verification of Modular Specifications

Key Idea

Constrained Abstract Refinements

Applications

HOFs, Databases, Floyd-Hoare, Capabilities.

Implementation

Implementation

LiquidHaskell

Liquid Types for Haskell

CUFP Tutorial on Thu!

Implementation

Abstract Refinements desugar to Ghost variables

Bounds desugar to Ghost functions

(à la TypeClass Dictionaries)

Goal

Automatic Verification of Modular Specifications

Key Idea

Constrained Abstract Refinements

Applications

HOFs, Databases, Floyd-Hoare, Capabilities.

Implementation

Ghost variables (à la TypeClass Dictionaries)

Goal

Automatic Verification of Modular Specifications

Key Idea

Constrained Abstract Refinements

Applications

HOFs, Databases, Floyd-Hoare, Capabilities.

Implementation

Ghost variables (à la TypeClass Dictionaries)

Details/FAQ

Details/FAQ

1. How do we prove soundness?
2. How fast is (bounded) type checking?
3. How easy is it so come up with bounds?
4. What are the limitations?
5. What's so great about decidability?

Questions?/Thank you!

END

$O(n^k)$ calls where k is the number of parameters in the bound and n the number of variables in scope