# Towards a Translation

from

# Liquid Haskell

to

# Coq

Lykourgos Mastorou, **Niki Vazou**, and Michael Greenberg
IMDEA Software Institute, Spain
Stevens Institute of Technology, USA

# Liquid Haskell
## Refinement Types for Haskell

Haskell Type: $\mathtt{incr :: Int \rightarrow Int}$

Code: $\mathtt{incr\ i = i + 1}$

# Liquid Haskell
## Refinement Types for Haskell

Refined Type: $\mathtt{incr :: i{:}Int \rightarrow \{v{:}Int \mid v = 1 + i\}}$

Haskell Type: $\mathtt{incr :: Int \rightarrow Int}$

Code: $\mathtt{incr\ i = i + 1}$

# Liquid Haskell
## Refinement Types for Haskell

Refined Type: $\mathtt{incr} :: \mathtt{i{:}Int} \rightarrow \{\mathtt{v{:}Int} \mid \mathtt{v = 1 + i}\}$

Haskell Type: $\mathtt{incr} :: \mathtt{Int} \rightarrow \mathtt{Int}$

Code: `incr i = i + 1`

SMT says:
$\mathtt{i + 1 = 1 + i}$

# Liquid Haskell
## Refinement Types for Haskell

Refined Type: $\mathtt{incr} :: \mathtt{i{:}Int} \rightarrow \{\mathtt{v{:}Int} \mid \mathtt{v} > \mathtt{i}\}$

Haskell Type: $\mathtt{incr} :: \mathtt{Int} \rightarrow \mathtt{Int}$

Code: $\mathtt{incr\ i} = \mathtt{i} + 1$

SMT says:
$\mathtt{i} + 1 > \mathtt{i}$

# Liquid Haskell
# For Safe Indexing

get :: xs:[a] → {i:Int | 0 ≤ i < len xs} → [a]

toPair :: String → (String, String)
toPair input = (get input 42, drop input 42)

**Expected:**
{i:Int | 0 ≤ i < len input}
**Got:**
{i:Int | i = 42}

# Liquid Haskell
# For Safe Indexing

```
get :: xs:[a] → {i:Int | 0 ≤ i < len xs} → [a]

toPair :: String → (String, String)
toPair input = (get input’ 42, drop input’ 42)
    where input’ = input ++
                        replicate (42 - length input) ‘😊’
```

# Liquid Haskell
# For Theorem Proving

```
thm :: xs:[a] → {i:Int | 0 ≤ i < len xs} →
         {xs = get xs i ++ drop xs i}
thm xs 0      = ()
thm [] i      = ()
thm (_:xs) i = thm xs (i-1)
```

# Liquid Haskell
# For Theorem Proving

```
thm :: xs:[a] → {i:Int | 0 ≤ i < len xs} →
         {xs = get xs i ++ drop xs i}
thm xs 0      = ()
thm [] i      = ()
thm (_:xs) i = thm xs (i-1)
```

Is that a proof?

# Liquid Haskell
# For Theorem Proving

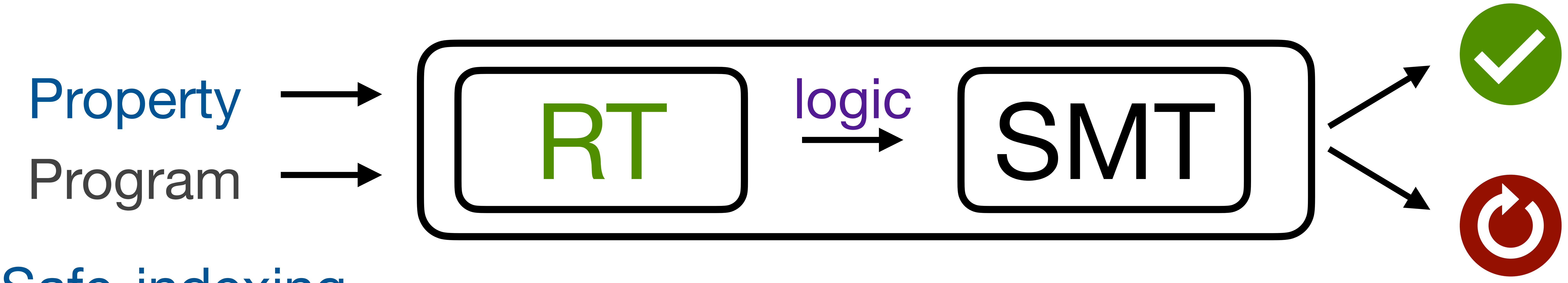
```
thm :: xs:[a] → {i:Int | 0 ≤ i < len xs} →
        {xs = get xs i ++ drop xs i}
thm xs 0      = ()
thm [] i      = ()
thm (_:xs) i = thm xs i
```

HUGE
UNREADABLE
ERROR MSG

No interaction

# Liquid Haskell

Is Practical



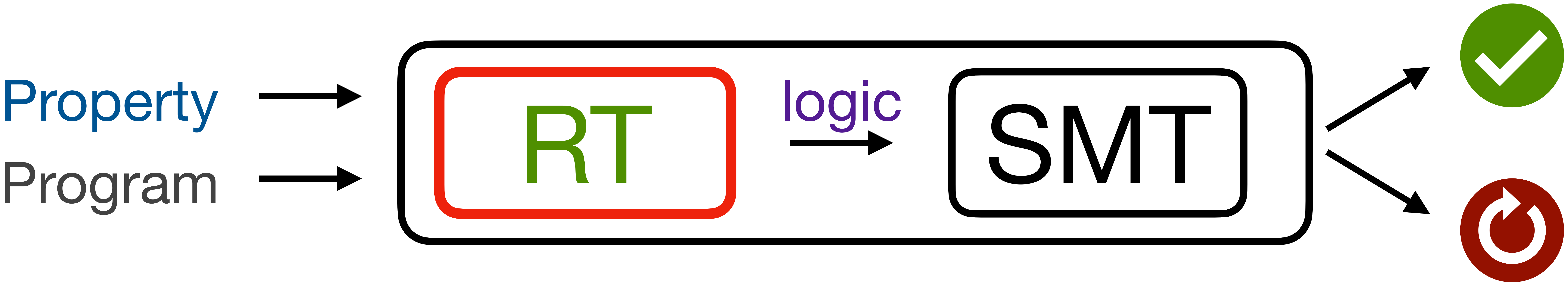

Safe-indexing

…
get xs i
…

INFO ⇒
$i \leq \text{len } xs$

# Liquid Haskell

## Is Practical, But Not Very Trustworthy




✗ Errors in RT

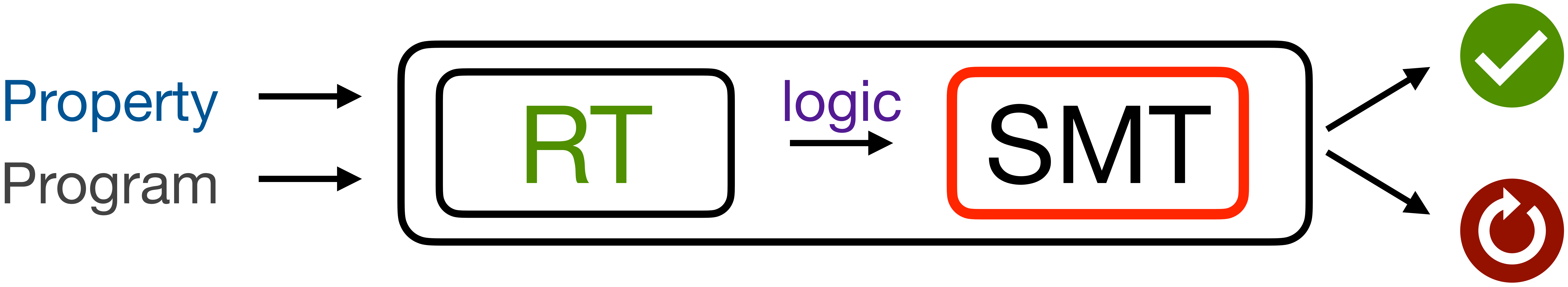
Function Extensionality is Tricky

Negative Occurrences are Unsound

How to Safely Use Extensionality in Liquid Haskell
by Niki Vazou and Michael Greenberg. Haskell 2022.

# Liquid Haskell

Is Practical, But Not Very Trustworthy




✘ Errors in RT

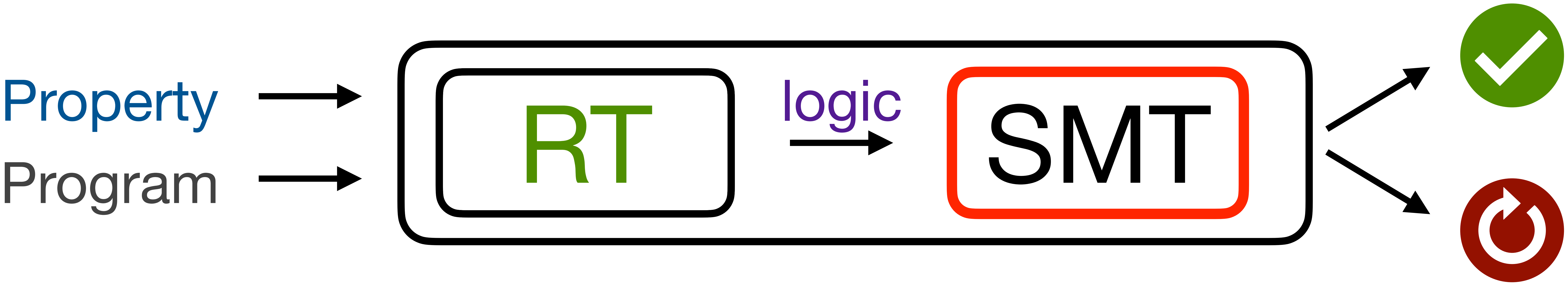
Function Extensionality is Tricky

Negative Occurrences are Unsound

✘ Errors in SMT

# Liquid Haskell

Is Practical, But Not Very Trustworthy




✘ Errors in RT

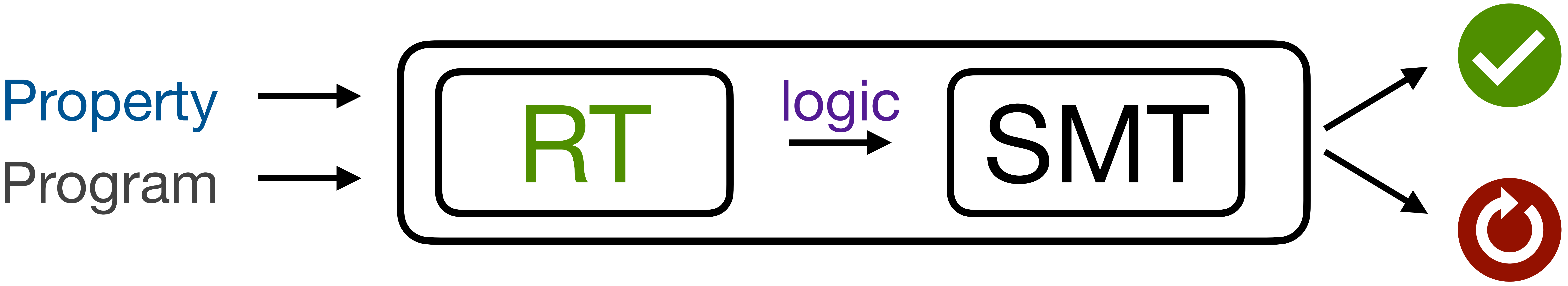
Function Extensionality is Tricky

Negative Occurrences are Unsound

✘ Errors in SMT

# Liquid Haskell

Is Practical, But Not Very Trustworthy




✘ Errors in RT

Function Extensionality is Tricky

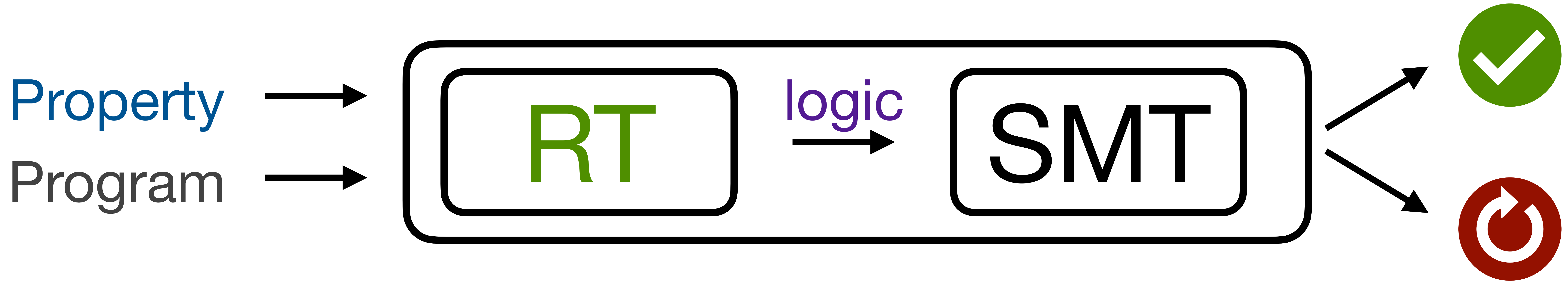
Negative Occurrences are Unsound

✘ Errors in SMT

✘ No Constructive Proofs

# Liquid Haskell

Is Practical, But Not Very Trustworthy




✗ Errors in RT

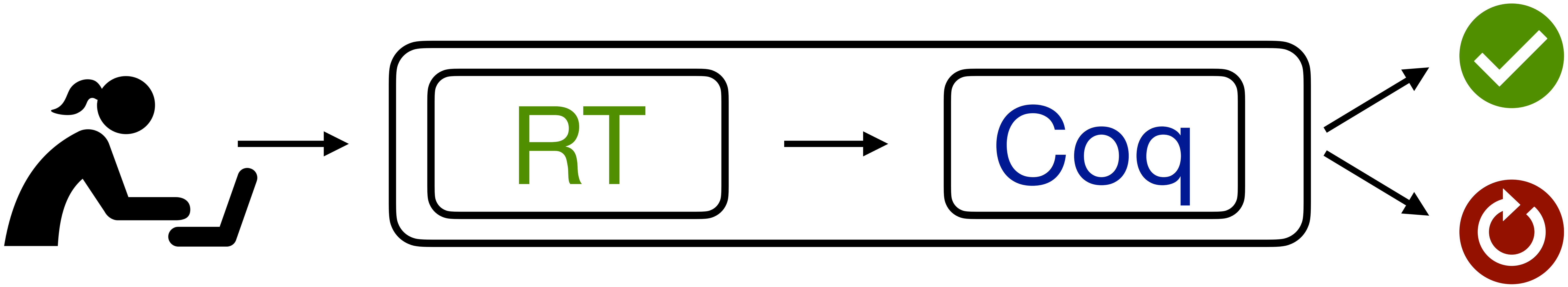
Function Extensionality is Tricky

Negative Occurrences are Unsound

✗ Errors in SMT

✗ No Constructive Proofs

# Liquid Haskell To Coq




- Keep Practical User Interface of LH
- Coq Soundness & Constructive Proofs
- Coq's Interactive Proof Development

Liquid Haskell $\Rightarrow$ Coq

Refinement Types
Functions
Semantic Subtyping
SMT Automation

Liquid Haskell $\Rightarrow$ Coq

Refinement Types

Functions

Semantic Subtyping

SMT Automation

# Refinement Types ⇒ Inductive Types

✓ Easy/Natural to Use

$\{i{:}\mathrm{Int} \mid 0 \leq i\} \;\Rightarrow\; \mathrm{Nat}$

✗ Do not break invariants

# Refinement Types ⇒ Inductive Types

✓ Easy/Natural to Use

$\{i{:}\mathrm{Int} \mid 0 \leq i\} \;\Rightarrow\; \mathrm{Nat}$

✗ Do not break invariants

$$\frac{m > 1 \qquad n :: \mathrm{Nat} \qquad m :: \mathrm{Nat}}{(n - 1) + m :: \mathrm{Nat}}$$

# Refinement Types $\Rightarrow$ Inductive Types

✓ Easy/Natural to Use

$\{i{:}\mathrm{Int} \mid 0 \leq i\} \quad \Rightarrow \quad \mathrm{Nat}$

✗ Do not break invariants

$$\frac{n :: \mathrm{Nat} \qquad m :: \mathrm{Nat} \qquad m > 1}{(n - 1) + m :: \mathrm{Nat}}$$

$\not{:}\ \mathrm{Nat}$

## Refinement Types $\Rightarrow$ Subset Types

$\{i{:}\mathrm{Int} \mid 0 \leq i\} \;\Rightarrow\; \{i{:}\mathrm{Int} \mid 0 \leq i\}$

$42 :: \{i{:}\mathrm{Int} \mid 0 \leq i\} \;\Rightarrow\; (42, \mathrm{evidence}) : \{i{:}\mathrm{Int} \mid 0 \leq i\}$

# Refinement Types $\Rightarrow$ Subset Types

$$\{i{:}\mathrm{Int} \mid 0 \leq i\} \quad \Rightarrow \quad \{i{:}\mathrm{Int} \mid 0 \leq i\}$$

$$42 :: \{i{:}\mathrm{Int} \mid 0 \leq i\} \quad \Rightarrow \quad (42, \mathrm{evidence}) : \{i{:}\mathrm{Int} \mid 0 \leq i\}$$

**Challenge: Construct evidence!**

Liquid Haskell $\Rightarrow$ Coq

Refinement Types $\Rightarrow$ Subset Types

Functions

Semantic Subtyping

SMT Automation

## Functions $\Rightarrow$ Fixpoint

Only structural induction

# Functions ⇒ Fixpoint

Only structural induction

```
ack :: m:Nat → n:Nat → Nat / [m,n]
ack 0 n = n + 1
ack m n = if n == 0 then ack (m-1) 1 else ack (m-1) (ack m (n-1))
```

## Functions ⇒ Fixpoint

Only structural induction

```
ack :: m:Nat → n:Nat → Nat / [m,n]
ack 0 n = n + 1
ack m n = if n == 0 then ack (m-1) 1 else ack (m-1) (ack m (n-1))
```

## Functions ⇒ Program Fixpoint

Only opaque functions

## Functions ⇒ Fixpoint

Only structural induction

```
ack :: m:Nat → n:Nat → Nat / [m,n]
ack 0 n = n + 1
ack m n = if n == 0 then ack (m-1) 1 else ack (m-1) (ack m (n-1))
```

## Functions ⇒ Program Fixpoint

Only opaque functions

## Functions ⇒ Equations

Function Definition & Induction Principle

Liquid Haskell ⇒ Coq

Refinement Types ⇒ Subset Types

Functions ⇒ Equations

Semantic Subtyping

SMT Automation

# Semantic Subtyping

$$\dfrac{\dfrac{4 :: \{i{:}Int \mid i = 4\} \qquad \{i{:}Int \mid i = 4\} \preccurlyeq Nat}{4 :: Nat} \qquad \dfrac{2 :: \{i{:}Int \mid i = 2\} \qquad \{i{:}Int \mid i = 2\} \preccurlyeq Nat}{2 :: Nat}}{\text{if } p \text{ then } 4 \text{ else } 2 :: Nat}$$

# Semantic Subtyping

$$\dfrac{\dfrac{4 :: \{i{:}\mathrm{Int} \mid i = 4\} \qquad \{i{:}\mathrm{Int} \mid i = 4\} \preccurlyeq \mathrm{Nat}}{4 :: \mathrm{Nat}} \qquad \dfrac{2 :: \{i{:}\mathrm{Int} \mid i = 2\} \qquad \{i{:}\mathrm{Int} \mid i = 2\} \preccurlyeq \mathrm{Nat}}{2 :: \mathrm{Nat}}}{\text{if } p \text{ then } 4 \text{ else } 2 :: \mathrm{Nat}}$$

$\Downarrow$ **Ref Custom Tactic**

if p then (ref 4 ...) else (ref 2 ...)

# Semantic Subtyping ⇒ Ref Custom Tactic

Introduce explicit casts via the **`ref`** tactic
at the places of semantic subtyping.

Concretely, because of bidirectional typing,
**join points, function calls and annotations.**

# Semantic Subtyping ⇒ Ref Custom Tactic

Introduce explicit casts via the **`ref`** tactic
at the places of semantic subtyping.

Concretely, because of bidirectional typing,
**join points, function calls and annotations.**

Modulo that
““Subtyping is difficult” — Felix” — Greta

Liquid Haskell ⇒ Coq

Refinement Types ⇒ Subset Types

Functions ⇒ Equations

Semantic Subtyping ⇒ Ref Custom Tactic

SMT Automation

# SMT Automation

Or, how do we implement `ref`?

We want SMT logic + proof search.

# SMT Automation

Or, how do we implement **`ref`**?

We want SMT logic + proof search.

Proof obligation for monotonicity of Ackerman:

$$\frac{\texttt{ack m (n-1) < ack m ((n-1)+1)} \qquad \texttt{ack m p < ack m (n-1)}}{\texttt{ack m p < ack m n}}$$

# SMT Automation

Or, how do we implement **`ref`**?

We want SMT logic + proof search.

SMT Coq + Coq Tactics = Sniper

General automation in Coq through modular transformations,
by Valentin Blot, Louise Prisque, Chantal Keller, and Pierre Vial. PxTP'21

Liquid Haskell $\Rightarrow$ Coq

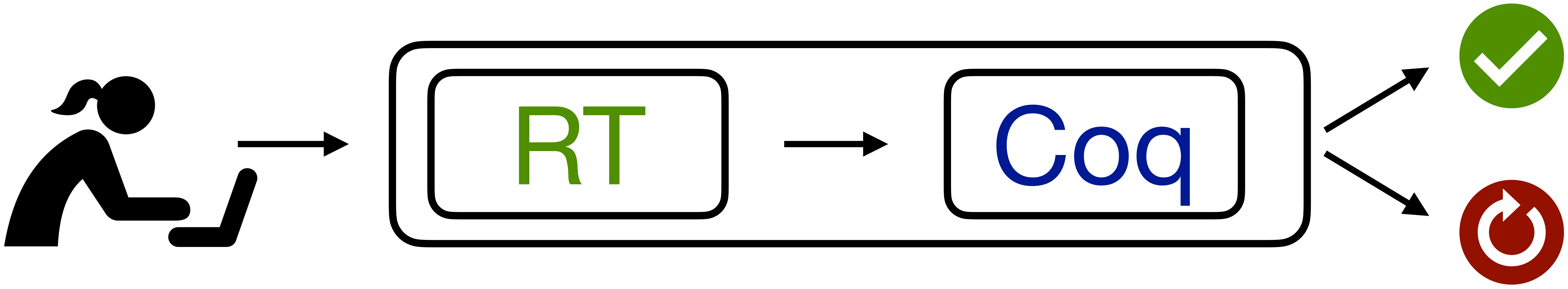
Refinement Types $\Rightarrow$ Subset Types

Functions $\Rightarrow$ Equations

Semantic Subtyping $\Rightarrow$ Ref Custom Tactic

SMT Automation $\Rightarrow$ Sniper

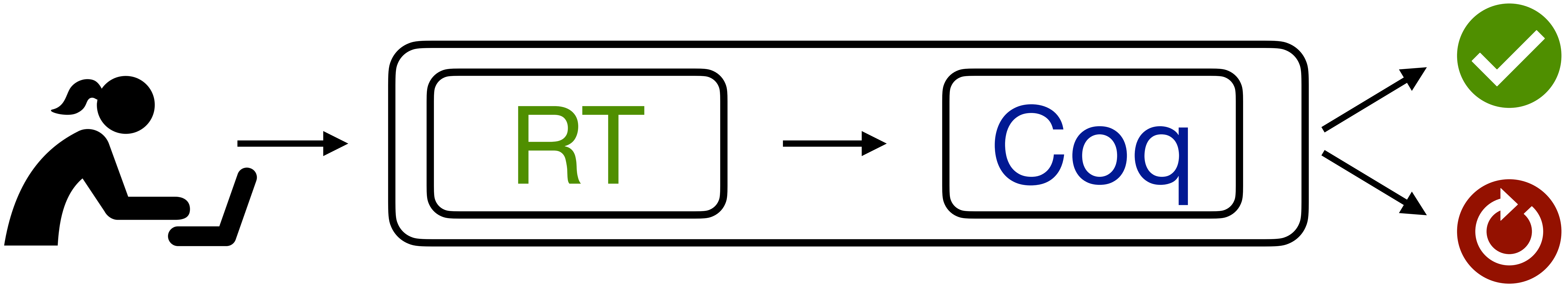
# Liquid Haskell ⇒ Coq




- ✓ **Practicality**
- ✓ **Soundness**
- ✓ **Interactivity**

| | | |
|---|---|---|
| Refinement Types | ⇒ | Subset Types |
| Functions | ⇒ | Equations |
| Semantic Subtyping | ⇒ | Ref Custom Tactic |
| SMT Automation | ⇒ | Sniper |

# Liquid Haskell ⇒ Coq




- Practicality
- Soundness
- Interactivity

Refinement Types ⇒ Subset Types
Functions ⇒ Equations
Semantic Subtyping ⇒ Ref Custom Tactic
SMT Automation ⇒ Sniper

*I am looking for students/collaborators!*

*Thanks!*