



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

Refined Type: `incr :: i:Int → {v:Int | v = 1 + i}`

Haskell Type: `incr :: Int → Int`

Code: `incr i = i + 1`



SMT says:
`i + 1 = 1 + i`



Liquid Haskell

Refinement Types for Haskell

Refined Type: `incr :: i:Int → {v:Int | v > i }`

Haskell Type: `incr :: Int → Int`

Code: `incr i = i + 1`



SMT says:
`i + 1 > 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)
```



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

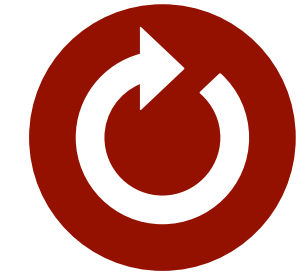
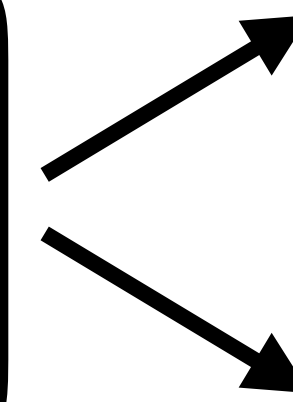
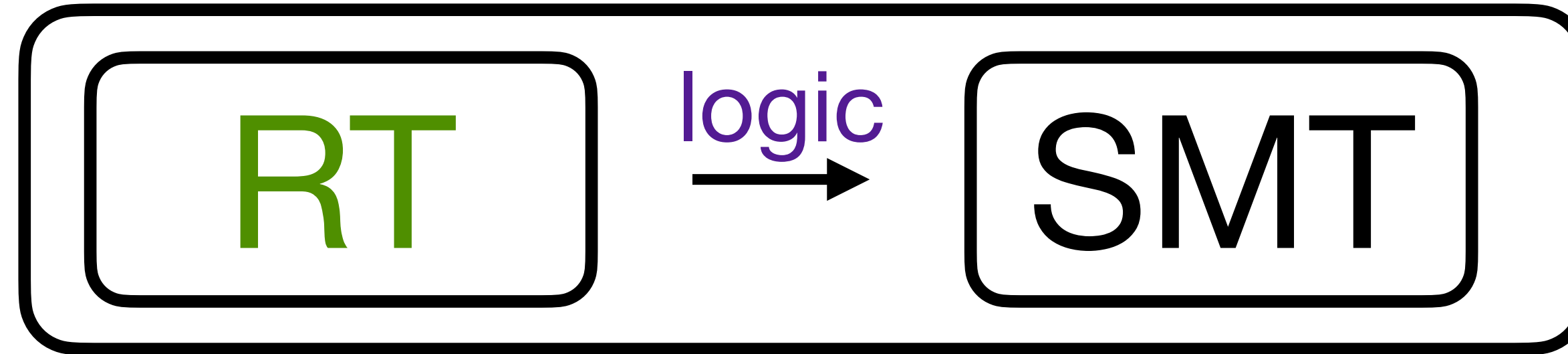
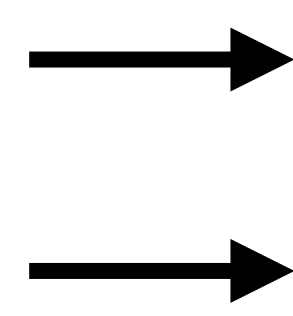


No interaction

Liquid Haskell

Is Practical

Property
Program



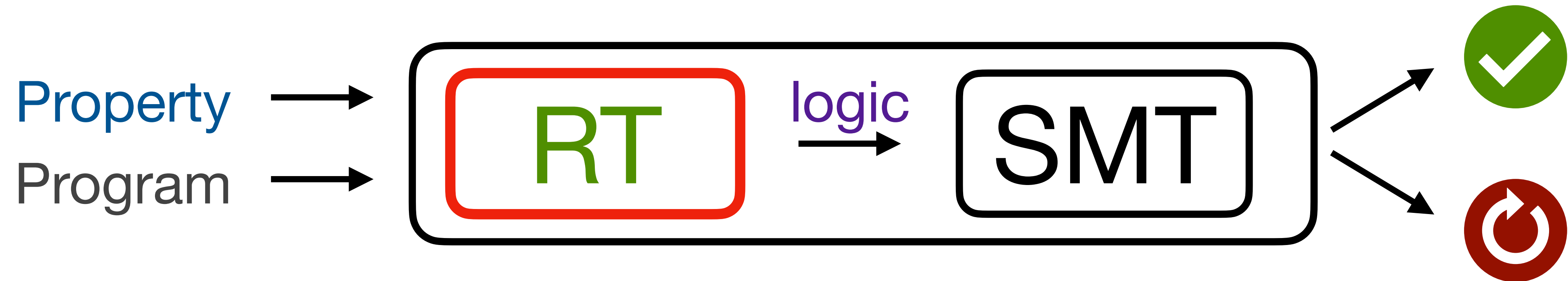
Safe-indexing

```
...  
get xs i  
...
```

```
INFO ⇒  
i ≤ len xs
```


Liquid Haskell

Is Practical, But Not Very Trustworthy



X Errors in RT

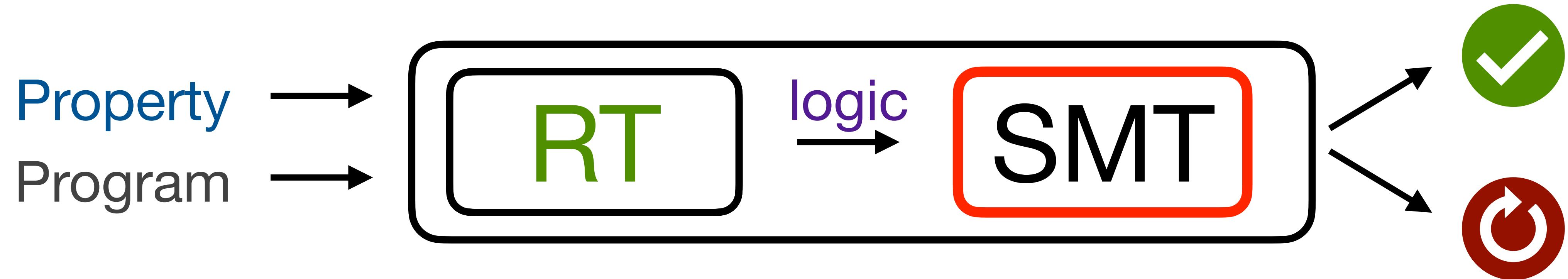
Function Extensionality is Tricky

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



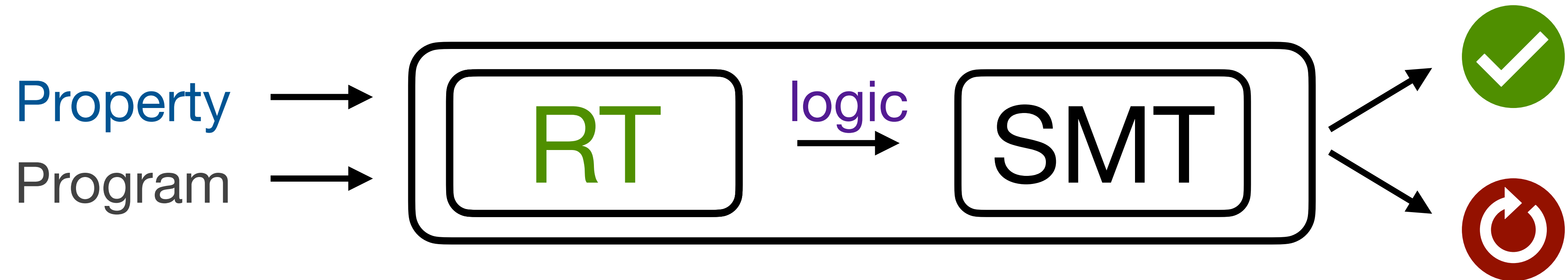
X Errors in RT

Function Extensionality is Tricky
Negative Occurrences are Unsound

X Errors in SMT

Liquid Haskell

Is Practical, But Not Very Trustworthy



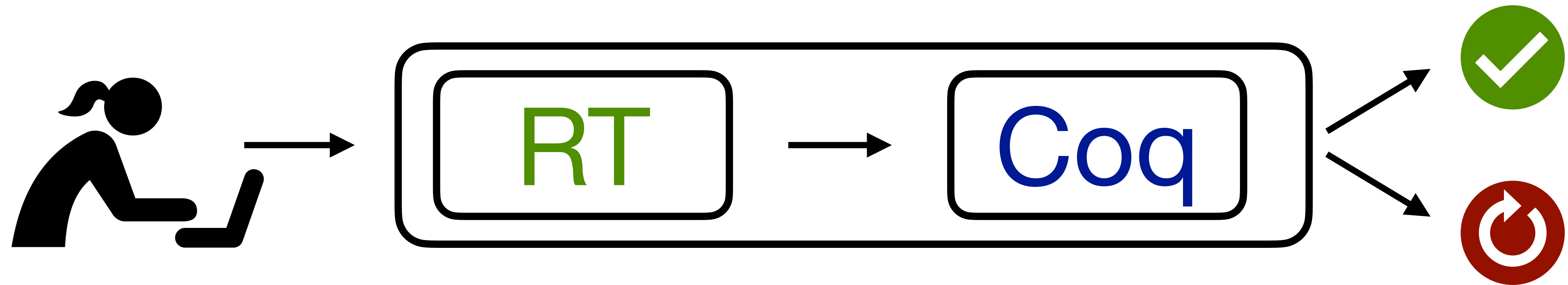
X Errors in RT

Function Extensionality is Tricky
Negative Occurrences are Unsound

X Errors in SMT

X 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 \Rightarrow Inductive Types

✓ Easy/Natural to Use

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

✗ Do not break invariants

$m > 1$

$n :: \text{Nat} \quad m :: \text{Nat}$

$(n - 1) + m :: \text{Nat}$

Refinement Types \Rightarrow Inductive Types

✓ Easy/Natural to Use

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

✗ Do not break invariants

$m > 1$

$n :: \text{Nat} \quad m :: \text{Nat}$

$(n - 1) + m :: \text{Nat}$

$\not:: \text{Nat}$

Refinement Types \Rightarrow Subset Types

$\{i:\text{Int} \mid 0 \leq i\} \Rightarrow \{i:\text{Int} \mid 0 \leq i\}$

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

Challenge: Construct evidence!

Liquid Haskell \Rightarrow **Coq**

Refinement Types \Rightarrow **Subset Types**

Functions

Semantic Subtyping

SMT Automation

Functions \Rightarrow Fixpoint

Only structural induction

`ack :: m:Nat \rightarrow n:Nat \rightarrow 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 \Rightarrow Program Fixpoint

Only opaque functions

Functions \Rightarrow Equations

Function Definition & Induction Principle

Liquid Haskell \Rightarrow **Coq**

Refinement Types \Rightarrow **Subset Types**

Functions \Rightarrow **Equations**

Semantic Subtyping

SMT Automation

Semantic Subtyping

$4 :: \{i:\text{Int} \mid i = 4\}$

$2 :: \{i:\text{Int} \mid i = 2\}$

$\{i:\text{Int} \mid i = 4\} \preceq \text{Nat}$

$\{i:\text{Int} \mid i = 2\} \preceq \text{Nat}$

$4 :: \text{Nat}$

$2 :: \text{Nat}$

$\text{if } p \text{ then } 4 \text{ else } 2 :: \text{Nat}$

↓ **Ref Custom Tactic**

$\text{if } p \text{ then } (\text{ref } 4 \dots) \text{ else } (\text{ref } 2 \dots)$

Semantic Subtyping \Rightarrow 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 \Rightarrow **Coq**

Refinement Types \Rightarrow **Subset Types**

Functions \Rightarrow **Equations**

Semantic Subtyping \Rightarrow **Ref Custom Tactic**

SMT Automation

SMT Automation

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

We want SMT logic + proof search.

Proof obligation for monotonicity of Ackerman:

$$\text{ack } m \ (n-1) < \text{ack } m \ ((n-1)+1)$$

$$\text{ack } m \ p < \text{ack } m \ (n-1)$$

$$\text{ack } m \ p < \text{ack } m \ n$$

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**

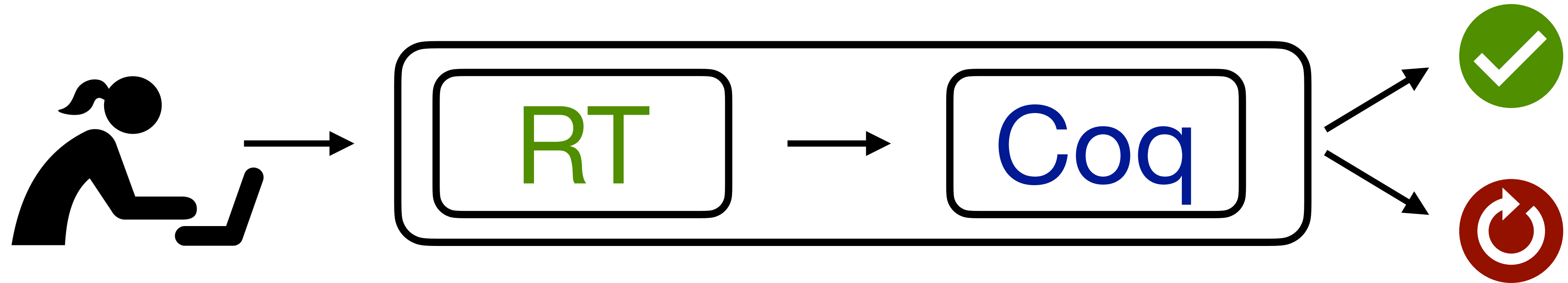
Refinement Types \Rightarrow **Subset Types**

Functions \Rightarrow **Equations**

Semantic Subtyping \Rightarrow **Ref Custom Tactic**

SMT Automation \Rightarrow **Sniper**

Liquid Haskell \Rightarrow Coq



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

Refinement Types \Rightarrow **Subset Types**
Functions \Rightarrow **Equations**
Semantic Subtyping \Rightarrow **Ref Custom Tactic**
SMT Automation \Rightarrow **Sniper**

I am looking for students/collaborators!

Thanks!