Niki Vazou
IMDEA
Madrid, Spain

CRETE:
Certified Refinement Types

Curiculue Vitae

2005-2010	Diploma	Advisor: Nikos Papaspyrou	NTUA
2011-2017	PhD	Advisor: Ranjit Jhala Title: “Refinement Types for Haskell”	UCSD
2017-2018	Post-doc	Host: David Van Horn Victor Basili Postdoc Fellow	UMD
2018-now	Research Ass. Prof.	Juan de la Cierva Fellow Atraccion de Talendo Fellow	IMDEA

Active Member of ACM SIGPLAN

Published in POPL (3), PLDI (1), OOPSLA(2), and ICFP(2).
Co-organized PLMW and 6 more venues.

Refinement Types

A type-based, SMT-automated verification technique, designed to be practical, ***but without*** strong foundations.

Refinement Types

Library:

```
get :: [a] → Int → a
zeros :: Int -> [Int]
```

User:

```
bad  i = get (zeros (i-1)) i   ✓
good i = get (zeros (i+1)) i   ✓
```

Existing Programming Language:
In-bound indexing cannot be expressed by types.

Refinement Types

Existing Type: $\mathtt{get} :: [\mathtt{a}] \rightarrow \mathtt{Int} \rightarrow \mathtt{a}$

Refinement Type: $\mathtt{get} :: \mathtt{xs}{:}[\mathtt{a}] \rightarrow \mathtt{i}{:}\{\mathtt{Int} \mid 0 \leq \mathtt{i} < \mathtt{len\ xs}\} \rightarrow \mathtt{a}$

Logical predicate
here encodes safe indexing

Refinement Types are Practical

Library:

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

User:

bad i = get (zeros (i-1)) i ✗
good i = get (zeros (i+1)) i ✓

Refinement Types are Practical

Library:

$\text{get} :: xs{:}[a] \to i{:}\{\text{Int} \mid 0 \leq i < \text{len } xs\} \to a$

$\text{zeros} :: i{:}\{\text{Int} \mid 0 \leq i\} \to \{o{:}[\text{Int}] \mid i = \text{len } o\}$

User:

```
bad  i = get (zeros (i-1)) i   ✘
good i = get (zeros (i+1)) i   ✔
```

Specs are naturally encoded.

Refinement Types are Practical

Library:

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

User:

```
bad  i = get (zeros (i-1)) i   ✗
good i = get (zeros (i+1)) i   ✓
```

Specs are naturally encoded.

User code is unmodified (thanks SMT!)

Refinement Types are Practical

Library:

$\text{get} :: xs{:}[a] \to i{:}\{\text{Int} \mid 0 \le i < \text{len } xs\} \to a$

$\text{zeros} :: i{:}\{\text{Int} \mid 0 \le i\} \to \{o{:}[\text{Int}] \mid i = \text{len } o\}$

User:

```
bad  i = get (zeros (i-1)) i   ✗
good i = get (zeros (i+1)) i   ✓
```

Specs are naturally encoded.

User code is unmodified (thanks SMT!)

Successfully used in industry and academia!

Refinement Types are not Sound

Library:

get :: xs:[a] → i:{Int | $0 \le i < \text{len } xs$} → a

zeros :: i:{Int | $0 \le i$} → {o:[Int] | $i = \text{len } o$}

User:

bad i = get (zeros (i-1)) i

good i = get (zeros (i+1)) i

>> good maxInt — maxInt = $2^{63} - 1$; maxInt+1<0

* * * Exception: Non-exhaustive patterns in function get

Runtime Spec Violation

* A *sound* system only accepts programs that never violate their specs

Refinement Types are not Sound

Library:

$$\text{get} :: xs{:}[a] \to i{:}\{\text{Int} \mid 0 \le i < \text{len } xs\} \to a$$

$$\text{zeros} :: i{:}\{\text{Int} \mid 0 \le i\} \to \{o{:}[\text{Int}] \mid i = \text{len } o\}$$

User:

```
bad  i = get (zeros (i-1)) i   ✗
good i = get (zeros (i+1)) i   ✓
```

Runtime Spec Violation

Axioms Can Make System Inconsistent

* A *sound* system only accepts programs that never violate their specs

Refinement Types are not Sound

Library:

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

User:

bad i = get (zeros (i-1)) i ✔ Unsoundly
good i = get (zeros (i+1)) i ✔

Axioms: E.g., "function extensionality"

Runtime Spec Violation

Axioms Can Make System Inconsistent

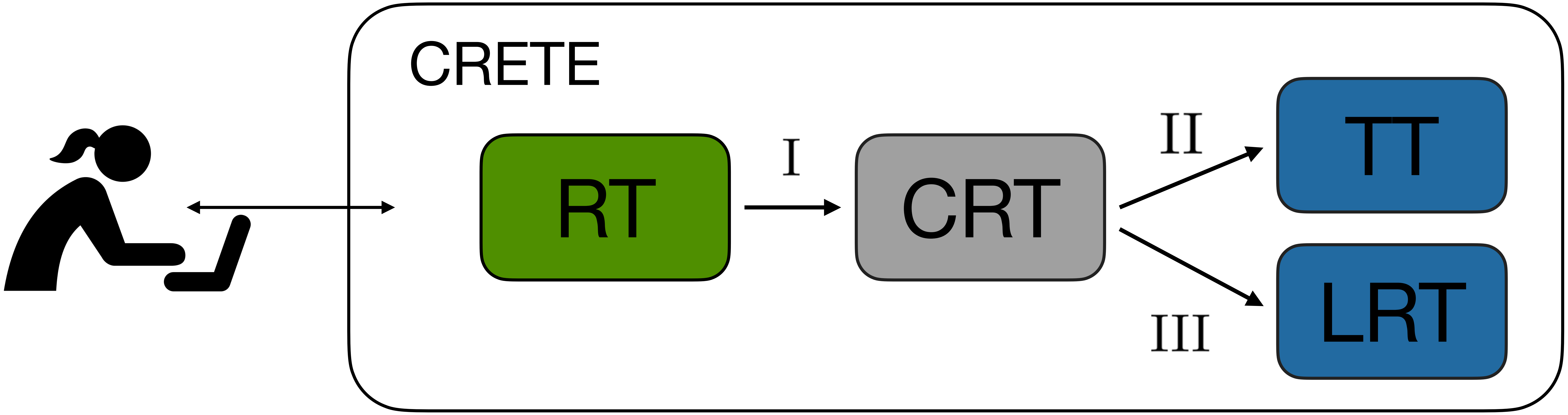
* A *sound* system only accepts programs that never violate their specs

Practical & Sound

CRETE:

A Practical *and* Sound Refinement Type System

Objectives of CRETE



User Interacts with Refinement Types (RT)

Objective I: RT → Certified Refinement Types (CRT)
Objective II: CRT → Sound Type Theory (TT)
Objective III: CRT → Logic of Refinement Types (LRT)

Objective I: RT→CRT

Explicit certificates that capture SMT automation.

```
good i = get (zeros (i+1)) i
```

```
good i = get (zeros (i+1)) (cert i {v:N | v < i + 1 } {i:N})
```

Explicit Certificate

Method: Type-based Syntactic Translation

Goal: Validate/test explicit certificates

Objective I: RT→CRT

Explicit certificates that capture SMT automation.

```
good i = get (zeros (i+1)) i
```

```
good i = get (zeros (i+1)) (cert i {v:N | v < i + 1 } {i:N})
```

Testing certificate $i < i + 1$:

```
Error counter-example found for i = maxInt
```

Challenge: Custom test generators (for corner cases)

Objective II: CRT→TT

The system is now as sound as TT (here Coq).

```
good i = get (zeros (i+1)) (cert i {v:N | v < i + 1 } {i:N})
```

```
Definition good (i:N) : N :=
   get (zeros (1+i)) (exist (fun v:N => v < 1 + i) i (lemma i)).
```

Proof of i < i+1

Method: Type-based Syntactic Translation

Objective II: CRT→TT

The system is now as sound as TT (here Coq).

```
good i = get (zeros (i+1)) (cert i {v:N | v < i + 1 } {i:N})
```

```
Definition good (i:N) : N :=
   get (zeros (1+i)) (exist (fun v:N => v < 1 + i) i (lemma i)).
```

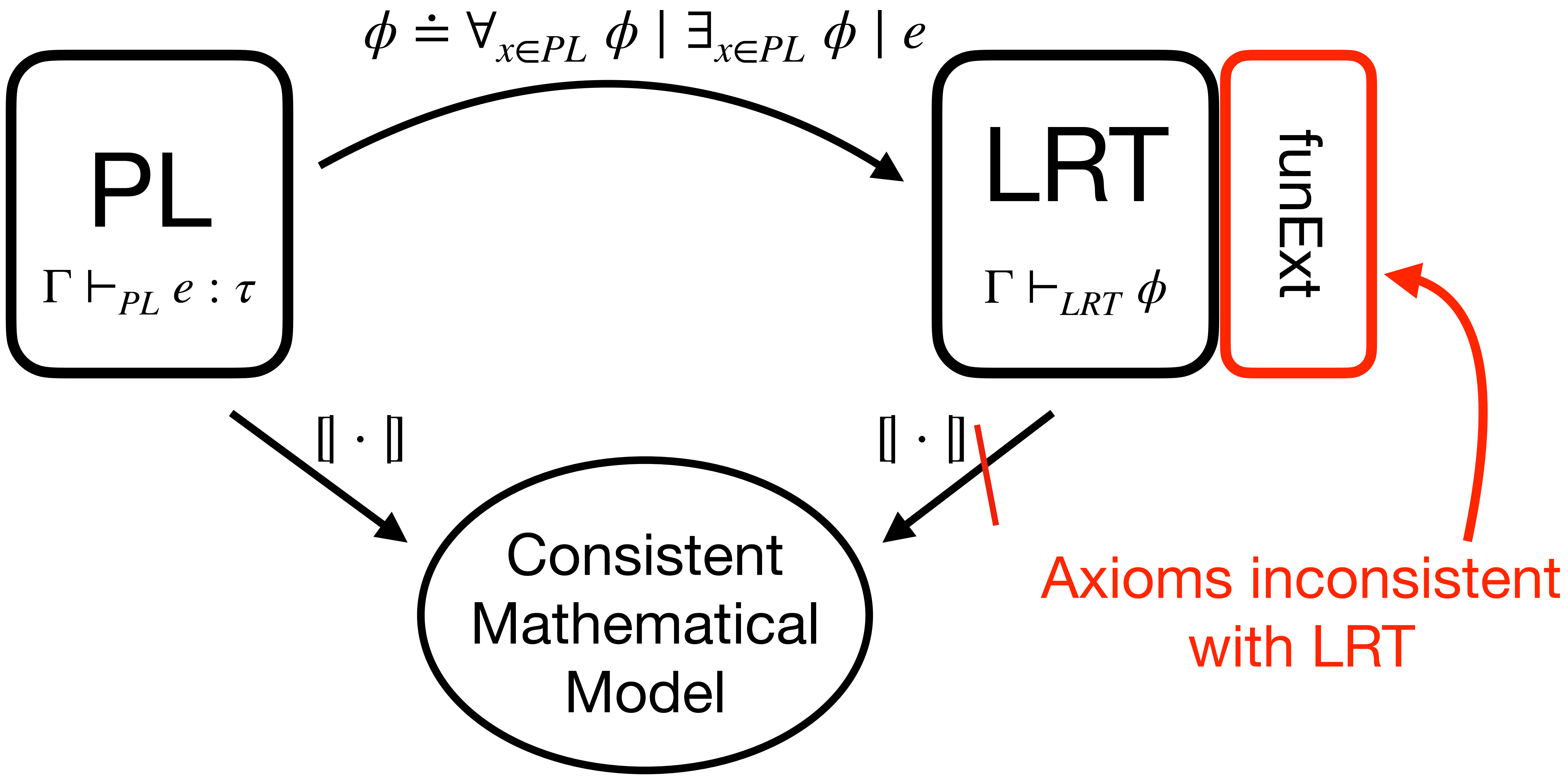
Proof of i < i+1

Risk: CRT→TT is not always possible

Theoretical Gain: Relationship between RT and TT

Objective III: Logic of Refinement Types (LRT)

Set Sound Foundations of RT using Program Semantics



Sound Approximation:

If $\Gamma \vdash_{RT} e : \{v : \tau \mid \phi\}$,
then $\Gamma \vdash_{LRT} \phi[v/e]$.

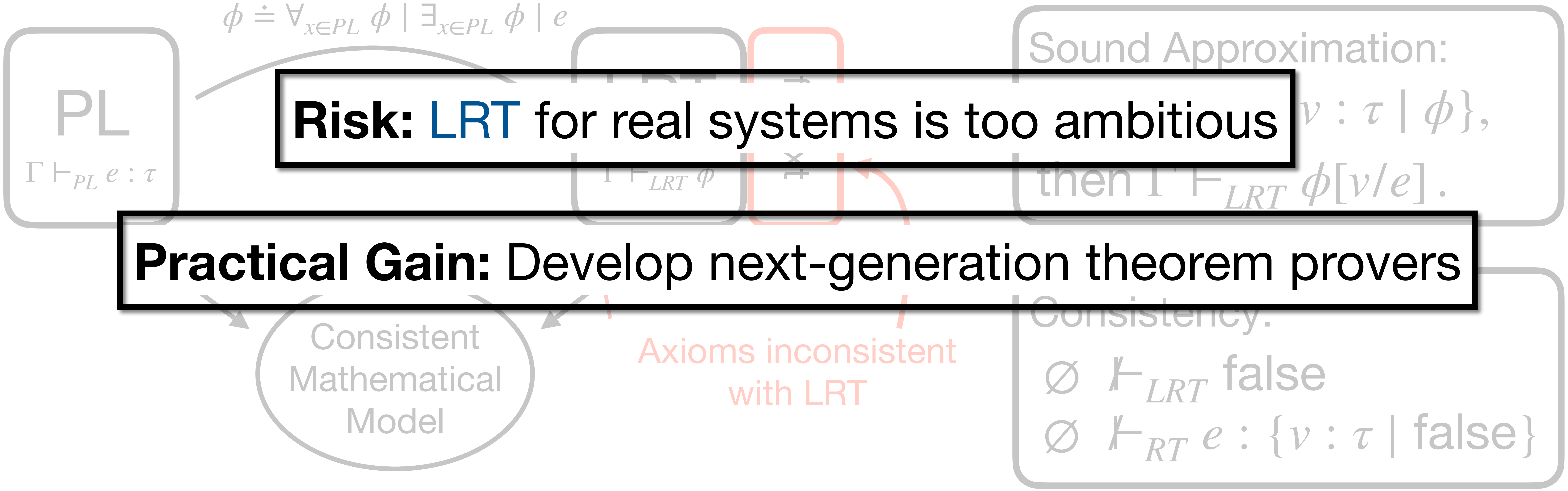
Consistency:

$\varnothing \nvdash_{LRT}$ false

$\varnothing \nvdash_{RT} e : \{v : \tau \mid \text{false}\}$

Objective III: Logic of Refinement Types (LRT)

Set Sound Foundations of RT using Program Semantics



Risk: LRT for real systems is too ambitious

Practical Gain: Develop next-generation theorem provers

Objective IV: Implementation & Evaluation

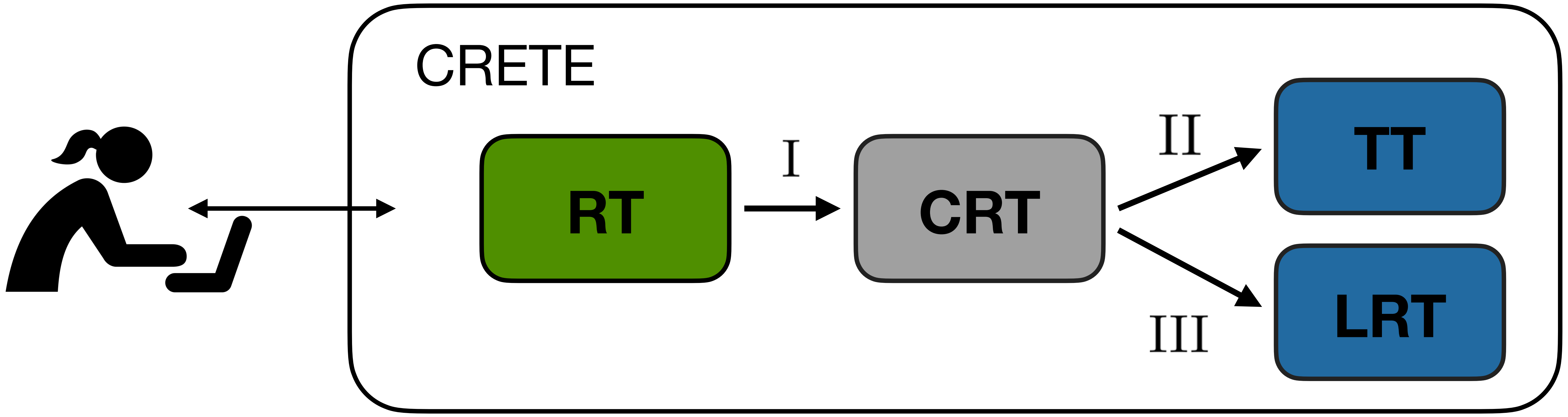
Implementation	**Haskell**	**Rust**
Current State:	Practical, but not sound	Under development
Target:	Secure Web Applications	Cryptographic Protocols

Evaluation:

Feasibility: Is CRETE both sound and practical in real programs?

Generality: Can we apply CRETE to more programming languages?

CRETE:
A Practical and Sound Refinement Type System used to verify real world application



Risk: **RT** → **TT** is not always possible

Risk: **LRT** for real systems is too ambitious

Gain: Set Foundations of Refinement Types

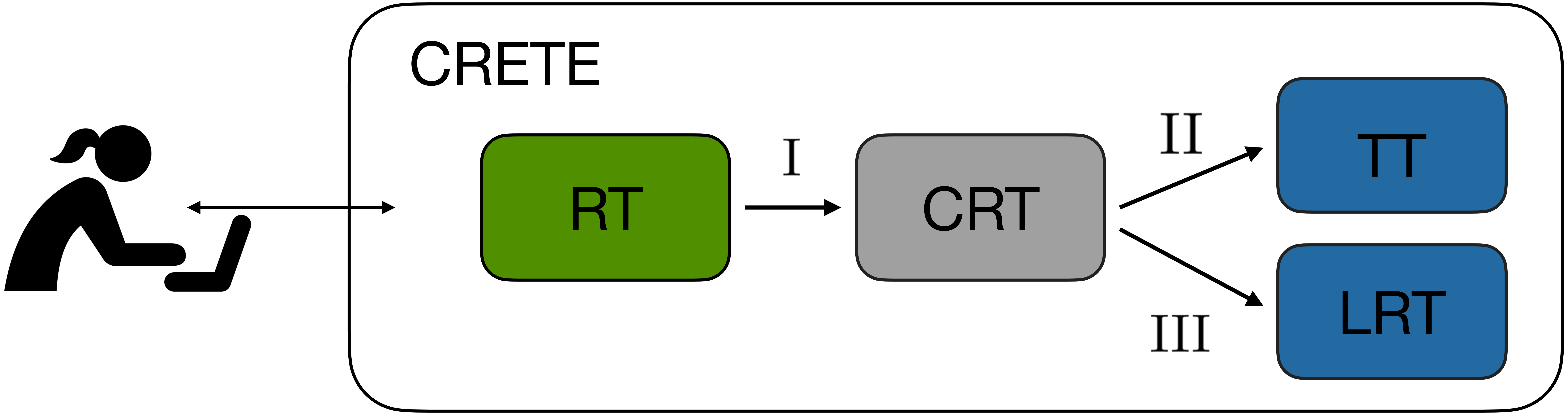
Gain: Novel low-cost high-gain verification

CRETE Group: Me (75%), 1 postdoc, 3 PhD, 1 engineer
+ Existing Collaborations (UCSD, UMD, …)

Thanks!

END

CRETE:
A Practical and Sound Refinement Type System



Risk: RT → TT might not always be possible

Parallel Approach: Development of the Logic of Refinement Types (LRT).

Objective III: Logic of Refinement Types

Goal: Define a Consistent Logic for RT

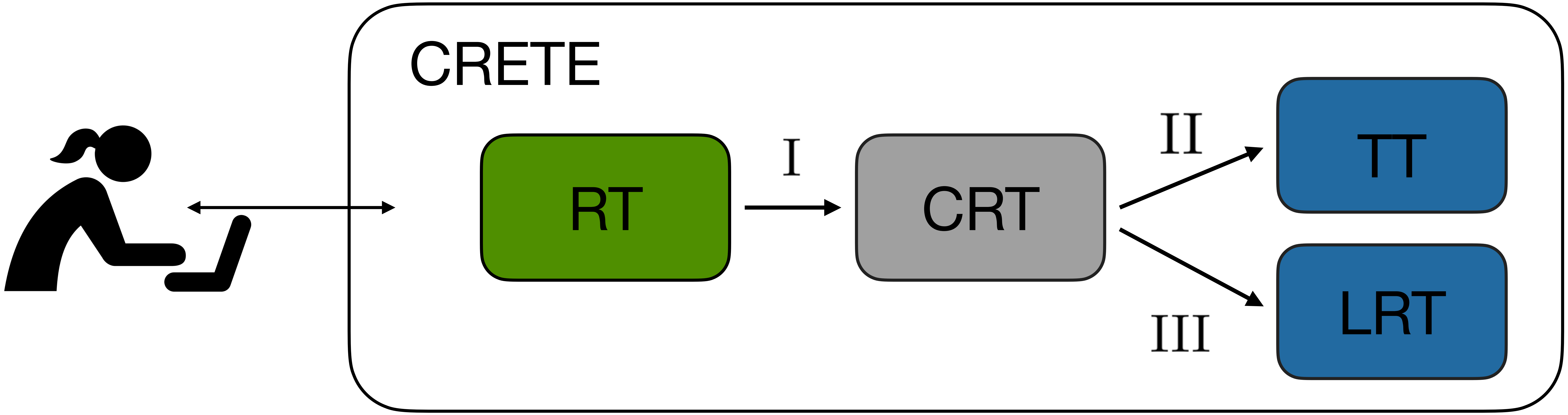
Method: Classical Program Semantics

Risk: LRT for real systems is too ambitious

Theoretical Gain: Define Foundations of RT

Practical Gain: Develop next-generation theorem provers

CRETE:
A Practical and Sound Refinement Type System used to verify real world application



Thanks!

Objective II: CRT→TT

Goal: Translation of RT to Coq

Method: Type-based Syntactic Translation

Risk: CRT→TT is not always possible

Practical Gain: Tiny TCB, I.e., that of Coq

Theoretical Gain: Relationship between RT and TT

Sound Type Systems

Soundness Recipe

Ingredients:
1. One Deductive System (DS)
2. One consistent mathematical theory
3. One tiny Trusted Code Base (TCB)

Methodology:
1. Show the DS has a model in the consistent theory
2. Implement the DS
3. Your implementation is your tiny TCB

Practicality Suggestion:
Extend your implementation with automation ultimately checked by your TCB

Coq	**Isabelle**
CIC	HOL
Set Theory	
15K	5K

~1 critical bug/year in Coq

Refinement Types are Practical

Graham Hutton
@haskellhutt

Replying to @alpha_convert @nikivazou and 2 others

What I like about refinement types is the low barrier to entry. Even a simple Haskell programmer like me can do refinement types (not a joke).

7:11 PM · Jun 11, 2021 · Twitter for iPhone

Refinement Types are not Sound

In existing refinement type checkers
(e.g., Liquid Haskell, F*, Stainless)
~5 unsoundness errors/year

In sound systems
(e.g.,Coq)
~1 critical bug/year

Lack sound foundations for Refinement Types

Practical  **Sound**

Currently:

Refinement types are Practical but not Sound

Objective I: RT→CRT

Goal: Extract explicit certificates for testing/validation

Method: Type-based Syntactic Translation

Gain: Reduce Trusted Code Base (TCB)

Objective I: RT→CRT

Goal: Extract explicit certificates for testing/validation

Method: Type-based Syntactic Translation

Gain: Reduce Trusted Code Base (TCB)