CRETE: Certified Refinement Types

Niki Vazou
IMDEA Software Institute
Madrid, Spain

About me

2011-2017	PhD	UCSD	Developed Liquid Haskell **Refinement Types** for Haskell
2017-2018	Post-doc	UMD	Established **Practicality** of Refinement Types
2018-now	Research Ass. Prof.	IMDEA	Observed lack of **Soundness** and its consequences

Member of ACM SIGPLAN

Published in **all** POPL (3), PLDI (1), OOPSLA(2), and ICFP(2).
Served as PC to POPL (2), OOPSLA(1), and ICFP(1).

Mentoring Activity

Co-organized Programming Languages **Mentoring** Workshop.
Now 1 PhD student; Past: Co-supervised 1 master & 2 PhD.

Refinement Types

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

Refinement Types

types refined with logical predicates

Existing Type: $\mathrm{get} :: \mathrm{List}\ a \rightarrow \mathrm{Int} \rightarrow \mathrm{List}\ a$

Refinement Type: $\mathrm{get} :: xs{:}\mathrm{List}\ a \rightarrow i{:}\{\mathrm{Int} \mid 0 \leq i \leq \mathrm{len}\ xs\} \rightarrow \mathrm{List}\ a$

Logical predicate
here encodes safe indexing

Refinement Types are Practical

i.e., used on real programs without extra user-effort

```haskell
-- | Packet Definition
data Packet = Packet {header :: String, body :: String} de

-- | Packet Parser
parser :: String -> Packet
parser input =
    Packet (get  headerSz input)
           (drop headerSz input)
```

Refinement Types are not Sound

```
unsound :: i:{Int | 0 ≤ i} → List Char
unsound i = get i (replicate (i+1)  '😈')
```

```
>> unsound maxInt — maxInt = 2^63 – 1; maxInt+1<0
* * * Exception: Non-exhaustive patterns in function get
```

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

Refinement Types are not Sound

```
unsound :: i:{Int | 0 ≤ i} → List Char
unsound i = get i (replicate (i+1) '😈')
```

```
>> unsound maxInt — maxInt = 2^63 – 1; maxInt+1<0
* * * Exception: Non-exhaustive patterns in function get
```

✘ Error in Correspondence between Program and Logic.

Fixed-precision Arbitrary-precision

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

Refinement Types are not Sound

```
unsound :: i:{Int | 0 ≤ i} → List Char
unsound i = get i (replicate (i+1)  '😈')
```

```
>> unsound maxInt — maxInt = 2^63 – 1; maxInt+1<0
* * * Exception: Non-exhaustive patterns in function get
```

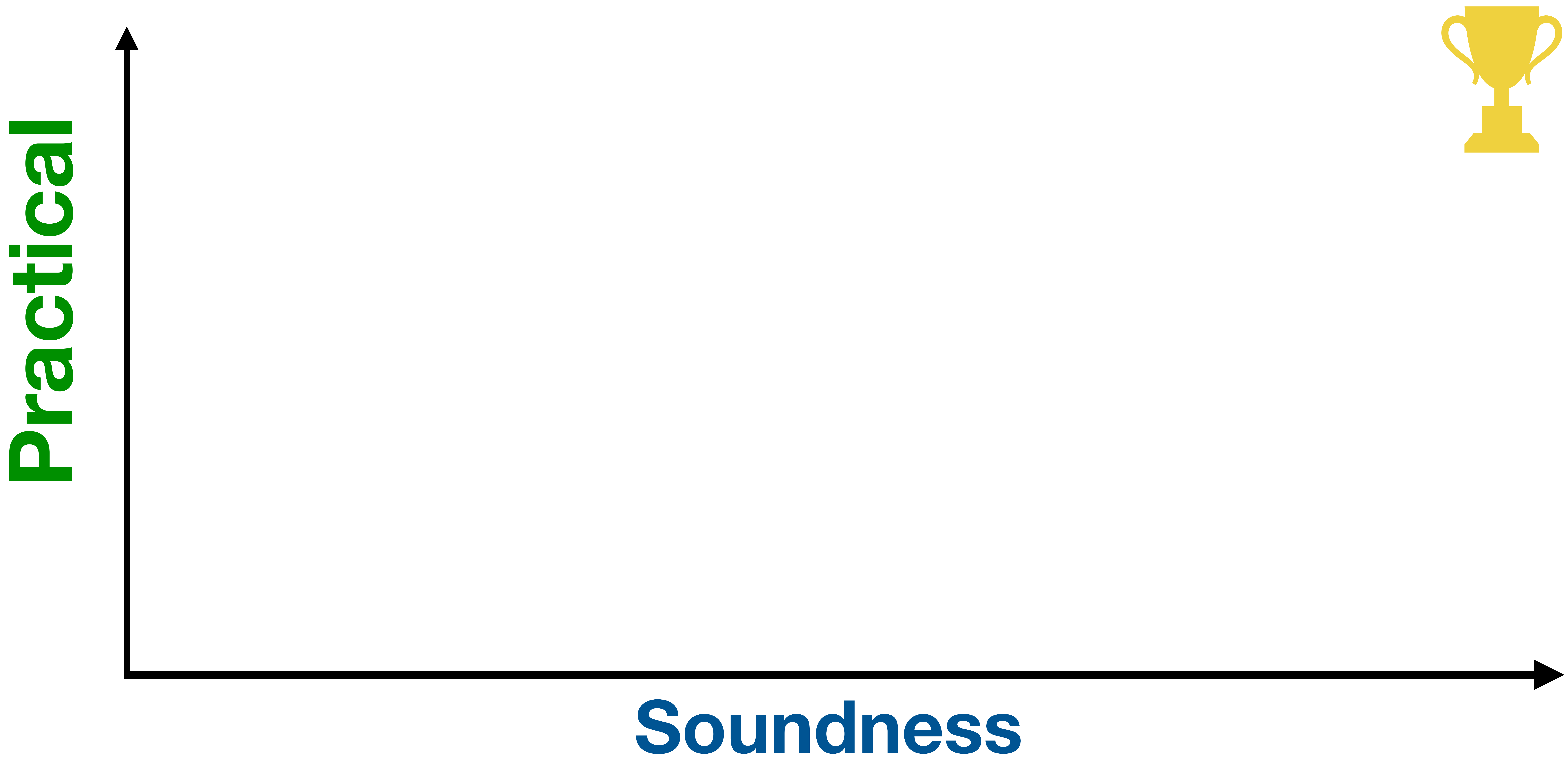
- ✗ Error in Correspondence between Program and Logic.
- ✗ Errors in the Logic of Refinement Types.
- ✗ Errors in the Implementation.

CRETE GOAL:
Establish Soundness of Refinement Types

As a result, develop a

Practical *and* Sound Verification System

Practical *and* Sound Verification



Practical *and* Sound Verification

Practical

Refinement Types
Liquid Haskell, F*

✓ modularity (type-based)
✓ automation (SMT-based)
✓ engineering (existing language)

Assertion-Based
Hoare-Logic, Dafny

✗ no type abstractions

Type-Theory
Coq, Agda

✗ no user-friendly engineering

Soundness

Practical *and* Sound Verification

Practical

Refinement Types

Liquid Haskell, F*

✗ Big, Trusted Code Bases (TCB)

✗ Partial, Paper-and-Pencil Formalism

Assertion-Based

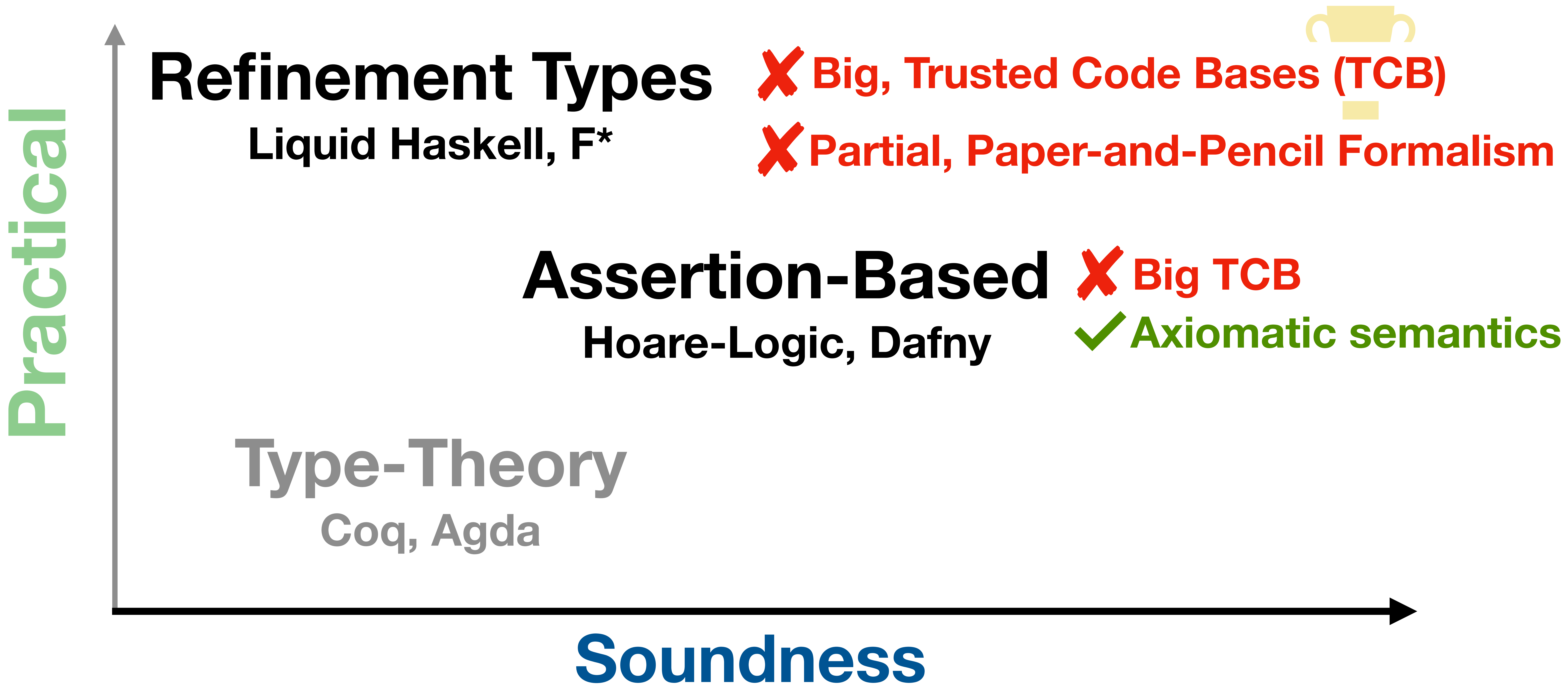
Hoare-Logic, Dafny

Type-Theory

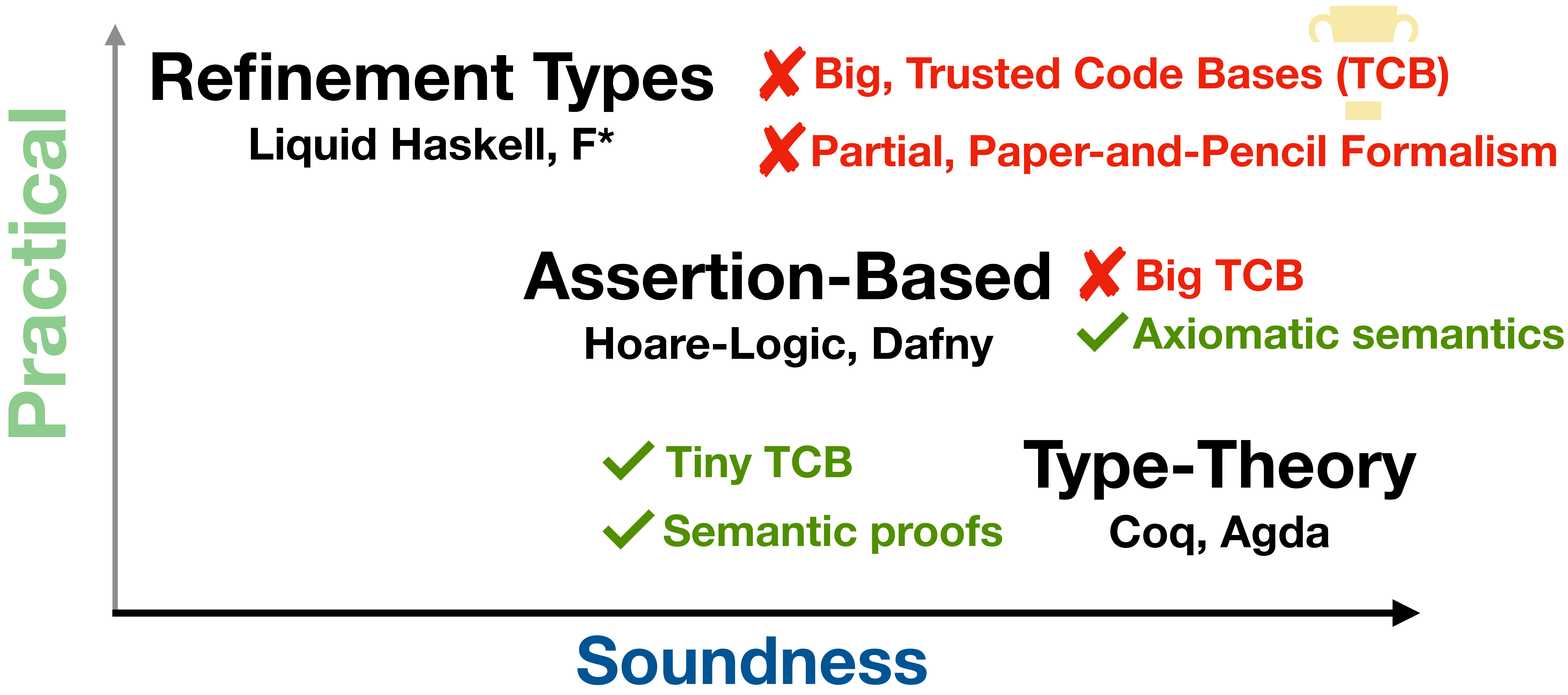
Coq, Agda

Soundness

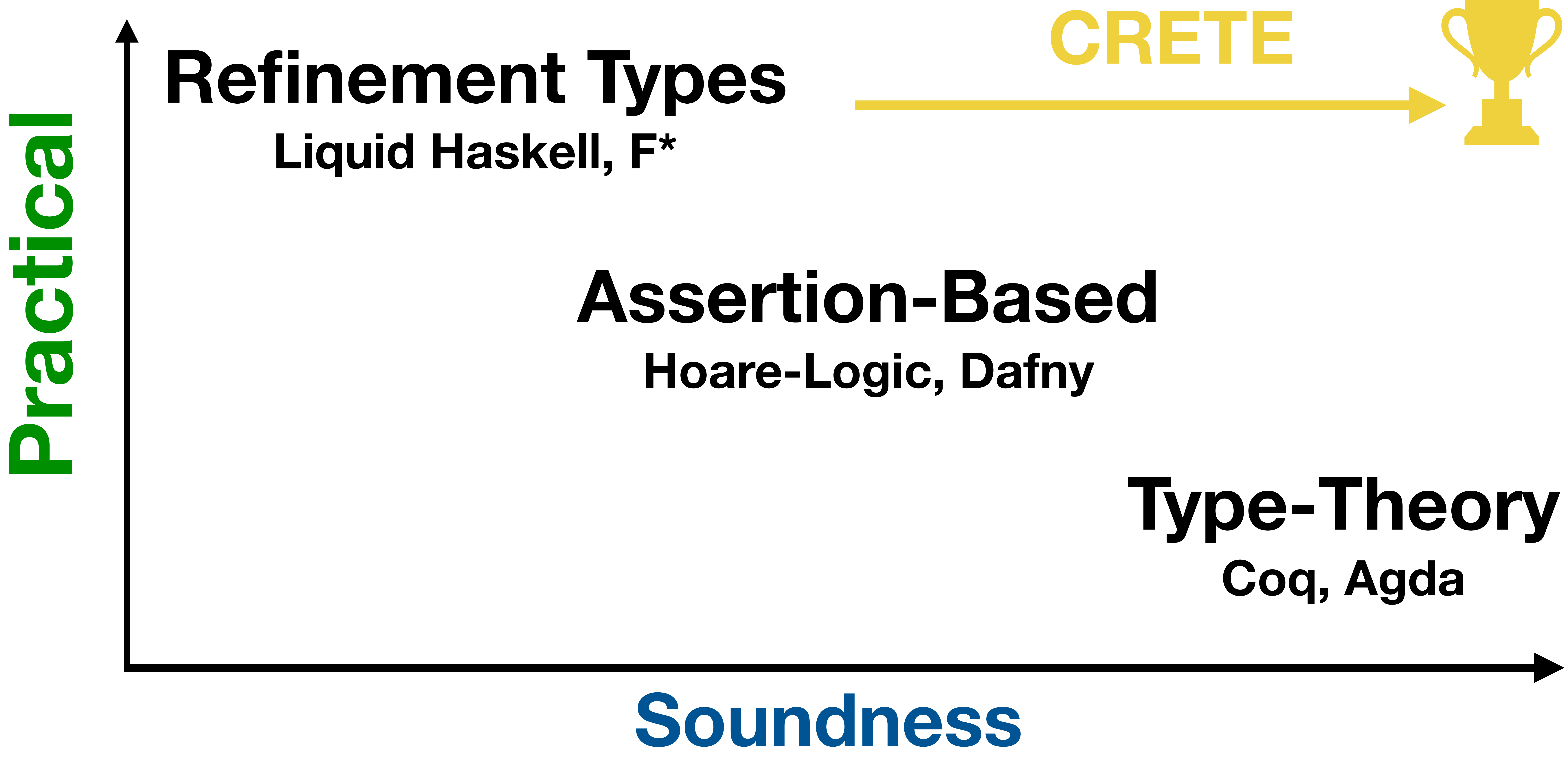
Practical *and* Sound Verification



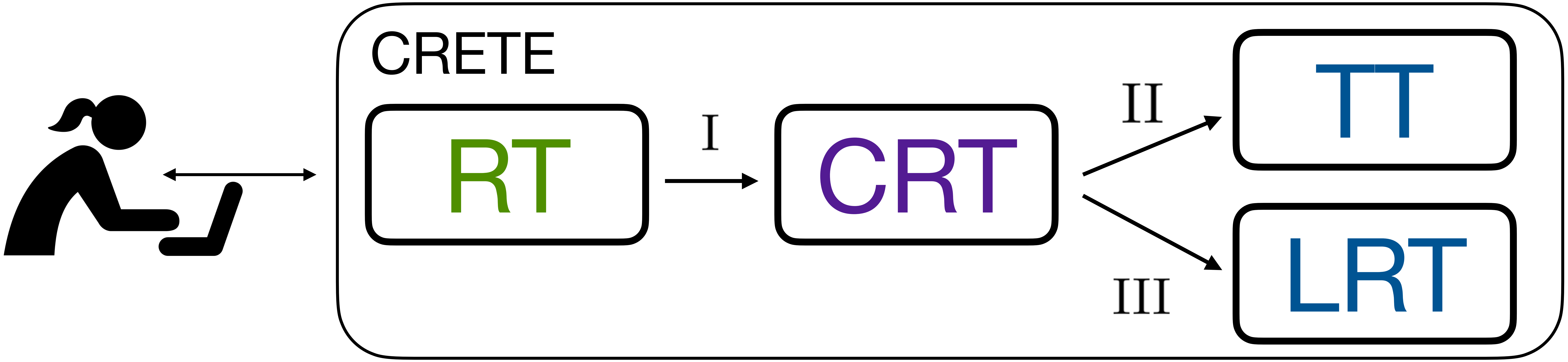
Practical *and* Sound Verification



Practical *and* Sound Verification



Scientific Objectives of CRETE

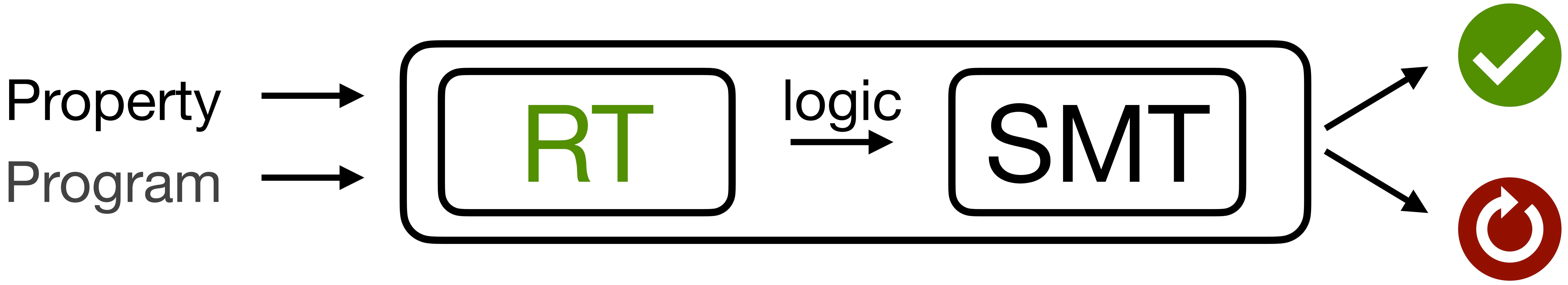


User Interacts with Refinement Types (RT)

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

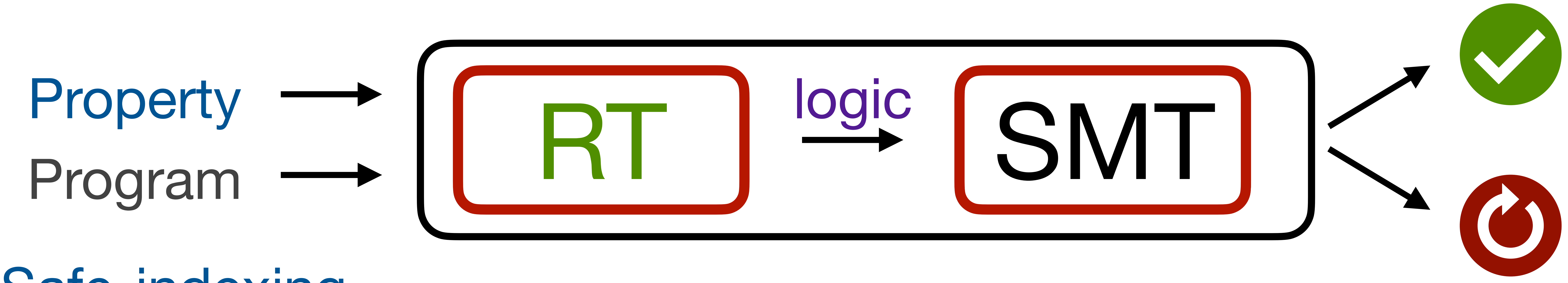
Refinement Types Now

Have Huge Trusted Code Base



Refinement Types Now

Have Huge Trusted Code Base



Safe-indexing

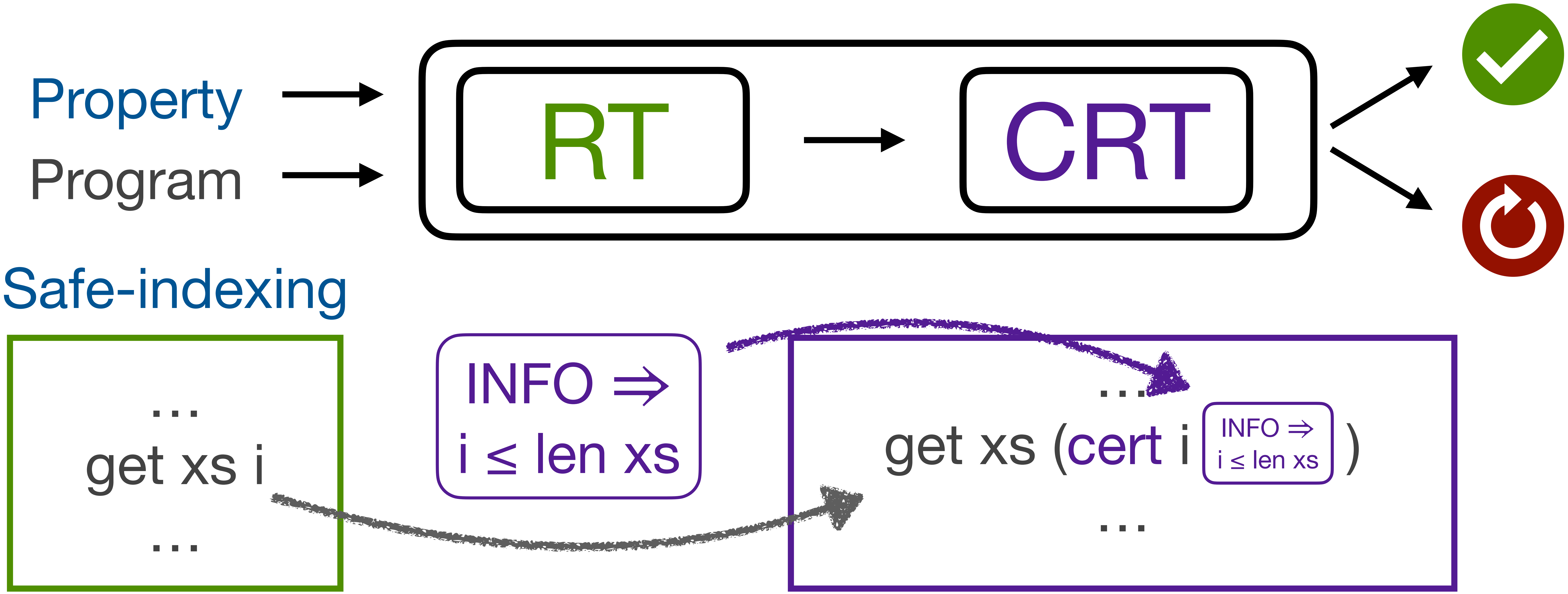
…
get xs i
…

INFO ⇒
$i \leq \text{len } xs$

✗ Errors in RT
✗ Errors in SMT

Objective I: RT→CRT

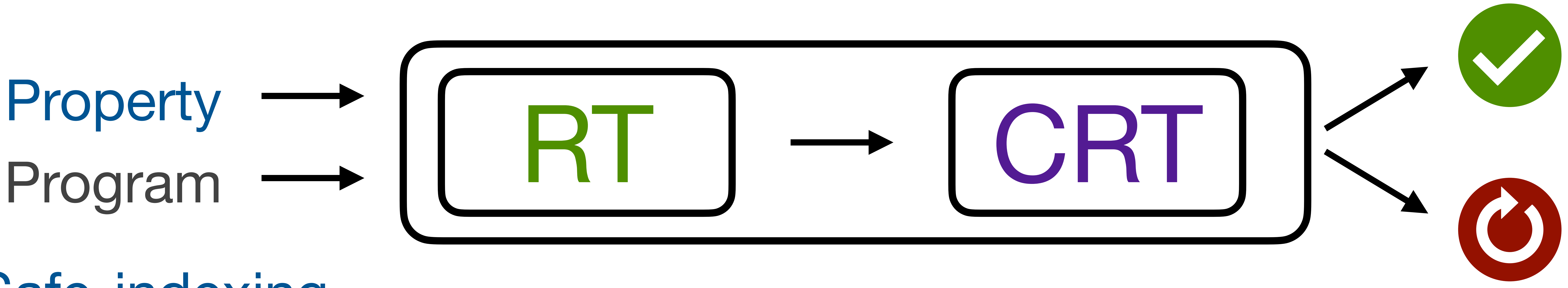
Explicit certificates that capture SMT automation.



- ✗ Errors in RT
- ✗ Errors in SMT
- ✓ Certificate Testing
- ✓ Certificate Validation

Objective I: RT→CRT

Explicit certificates that capture SMT automation.



Safe-indexing

…
get xs i
…

INFO ⇒
i ≤ len xs

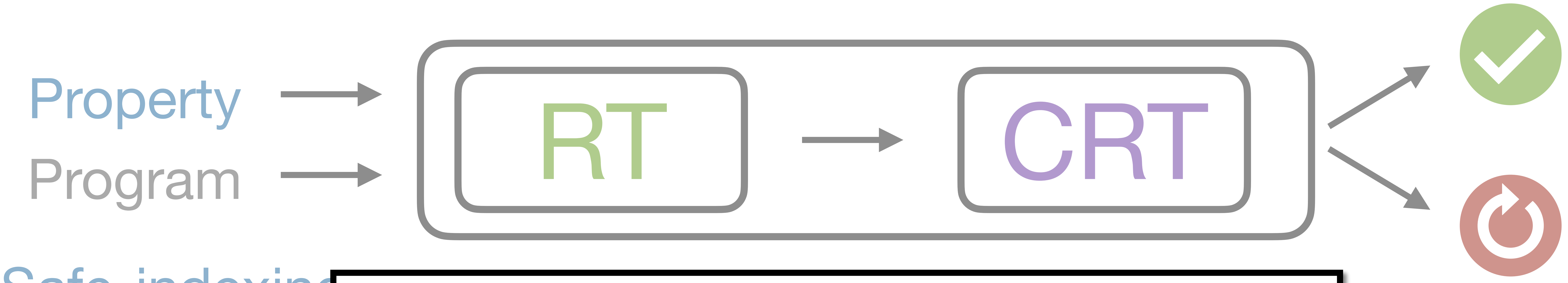
…
get xs (cert i (INFO ⇒ i ≤ len xs))
…

Certificate Testing
for the unsound example

INFO := len xs = i + 1
Certificate := i ≤ i+1
Counter-example := maxInt

Objective I: RT→CRT

Explicit certificates that capture SMT automation.



Safe-indexing

Challenge: Custom Test Generators

… get xs i …

Gain: Reduce Trusted Code Base

Gain: Stepping stone for Soundness

Certificate

for the unsound example

Counter-example := maxInt

Objective II: CRT→TT

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

RT → CRT → TT

…

get xs (cert i $\boxed{\text{INFO} \Rightarrow i \leq \text{len xs}}$)

…

→

…

get xs (exists i $\boxed{\text{PROOF}}$)

…

Syntactic Translation (cert → existential)

+

Coq automation (e.g., SMTCoq)

Objective II: CRT→TT

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

RT → CRT → TT

Practical Gain: System is Sound

Theoretical Gain: Relationship between RT and TT

get xs (cert i i ≤ len xs) → get xs (exists i PROOF)

Risk: CRT→TT is might always be possible

Syntactic Translation (cert → existential)

\+

Coq automation (e.g., SMTCoq)

Objective III: Logic of Refinement Types (LRT)

Set Sound Foundations of RT using Program Semantics

LRT $(\Gamma \vdash_{LRT} \phi)$

1) approximates RT

2) is sound

a. has consistent model (to eliminate errors in the logic)

b. has tiny kernel (to reduce errors in the implementation)

Sound Approximation:

If $\Gamma \vdash_{RT} e : \{v : \tau \mid \phi\}$,

then $\Gamma \vdash_{LRT} \phi[v/e]$.

Objective III: Logic of Refinement Types (LRT)

Set Sound Foundations of RT using Program Semantics

LRT (

Risk: LRT for real systems with divergence, state, …

1) approximates RT

Theoretical Gain: Set Principles of Refinement Types

2)

a. has consistent model. (Eliminate errors in the logic)

Practical Gain: Develop next-generation theorem provers

ation)

Objective IV: Implementation & Evaluation

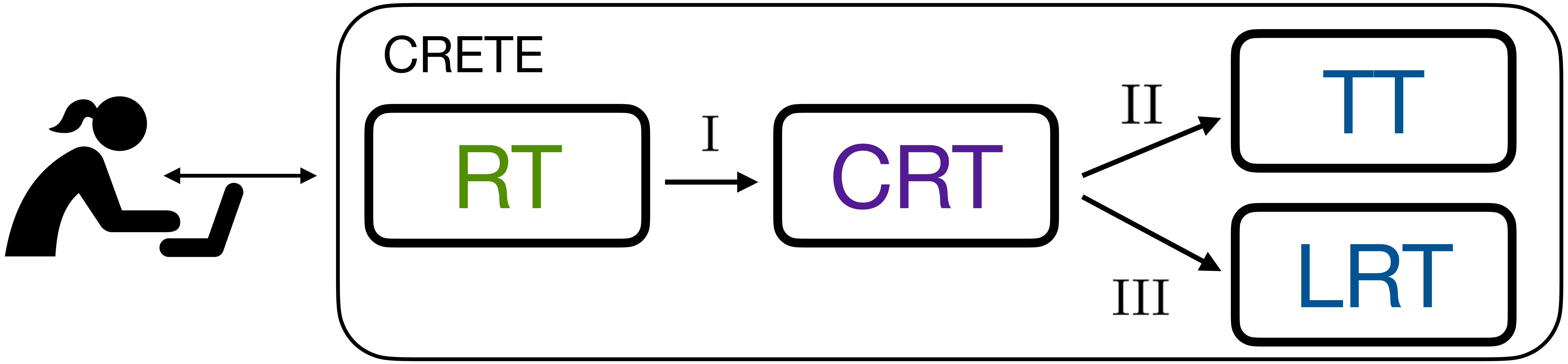
Implementation	Haskell	Rust
Current State:	Practical, but not sound	Under development
Target:	Secure Web Applications	Cryptographic Protocols

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



Gain: Set Foundations of Refinement Types.
Gain: Make sound verification accessible via practicality of refinement types.

Risk: **RT** → **TT** might not be always possible.
Risk: **LRT** for real systems is too ambitious

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

END