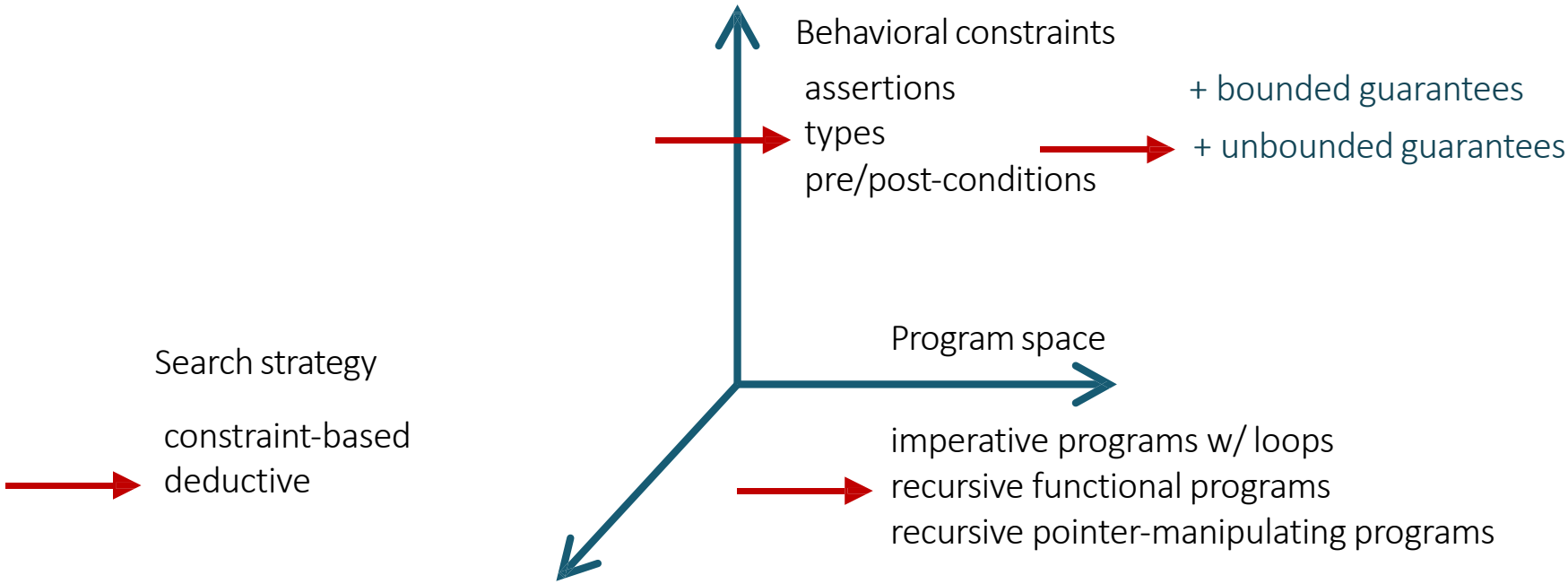
Lecture 13
Type-Driven Synthesis

This week


Behavioral constraints
assertions
types
pre/post-conditions
+ bounded guarantees
+ unbounded guarantees
Program space
imperative programs w/ loops
recursive functional programs
recursive pointer-manipulating programs
Search strategy
constraint-based
deductive

Agenda

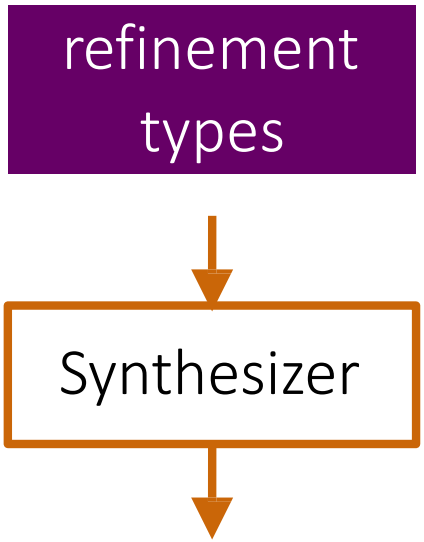
Last lecture:

- Simple types and how to check them
- Refinement types and how to check them

Today:

- Specification for insert as a refinement type
- Deductive search with refinement types

Specifications for synthesis



1. supports automatic, modular verification
2. abstract and concise
3. sufficiently expressive

```
replicate n x =
  if n ≤ 0
    then Nil
    else Cons x (replicate (dec n) x)
```

Demo: replicate

-- Specification:

replicate :: n: Nat → x: α → {v: List α | len v = n}

replicate = ??

-- Components:

zero :: {v: Int | v = 0}

inc :: x: Int → {v: Int | v = x + 1}

dec :: x: Int → {v: Int | v = x - 1}

leq :: x: Int → y: Int → {Bool | v = (x ≤ y) }

neq :: x: Int → y: Int → {Bool | v = (x ≠ y) }

Synthesis from refinement types

$$\Gamma \vdash ?? :: T$$

Synthesis from refinement types

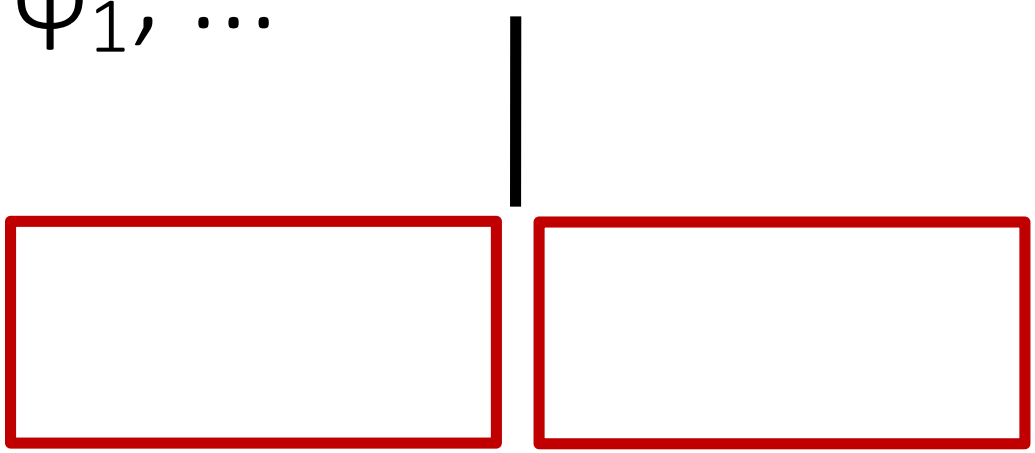
$$x_1 :: T_1; \ldots \quad \phi_1; \ldots \vdash ?? :: T$$

Synthesis from refinement types

$x_1 :: T_1; \ldots$

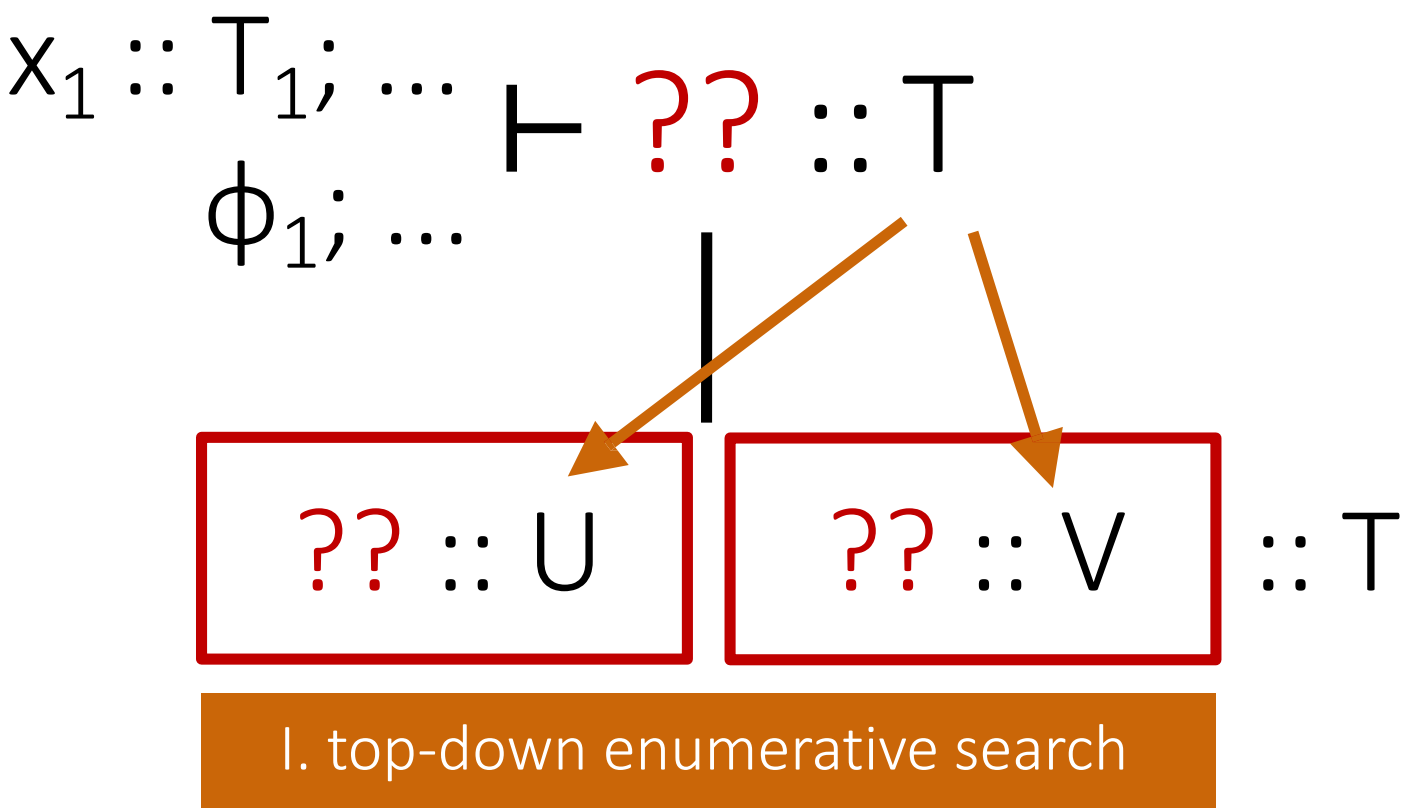
$\phi_1; \ldots$

$\vdash ?? :: T$

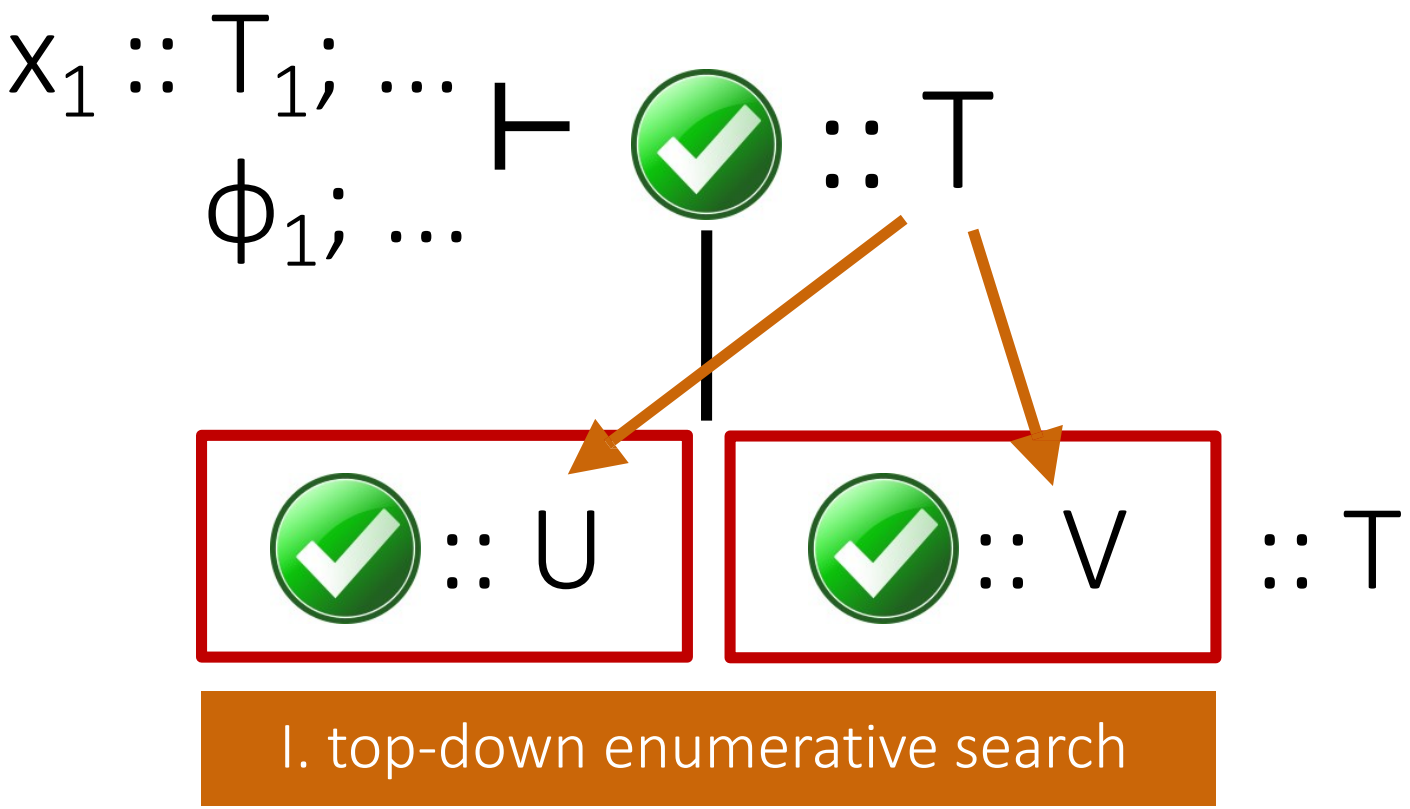


I. top-down enumerative search

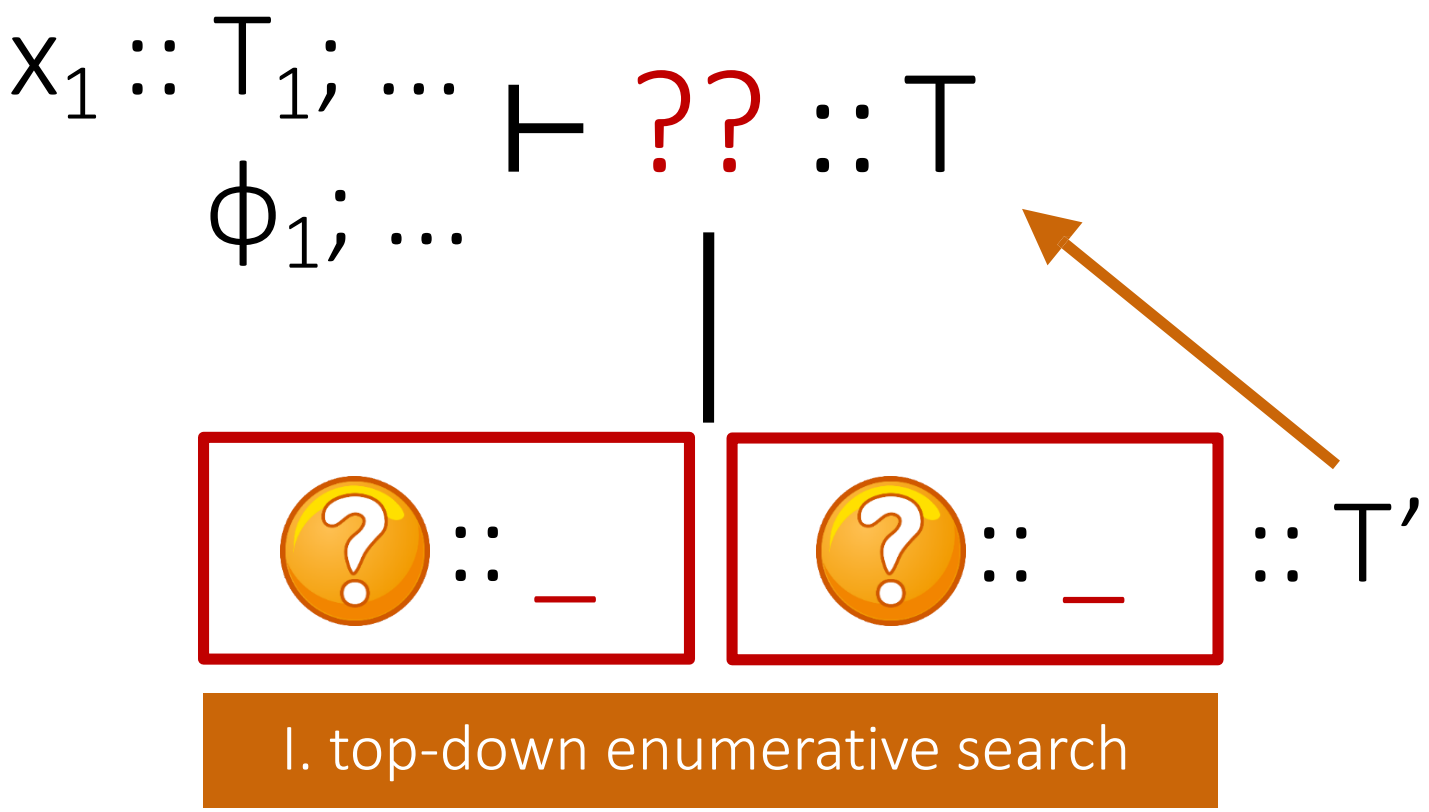
Synthesis from refinement types



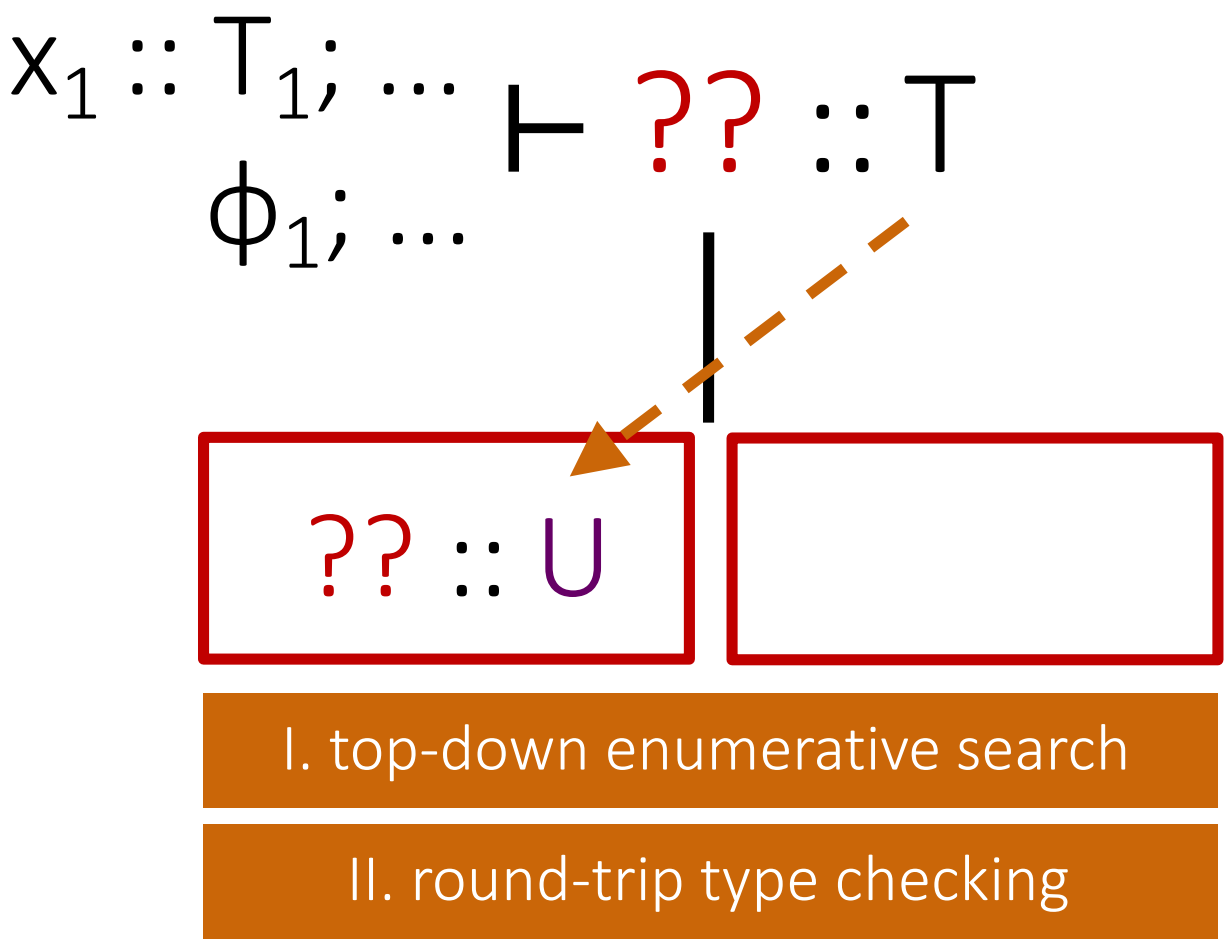
Synthesis from refinement types



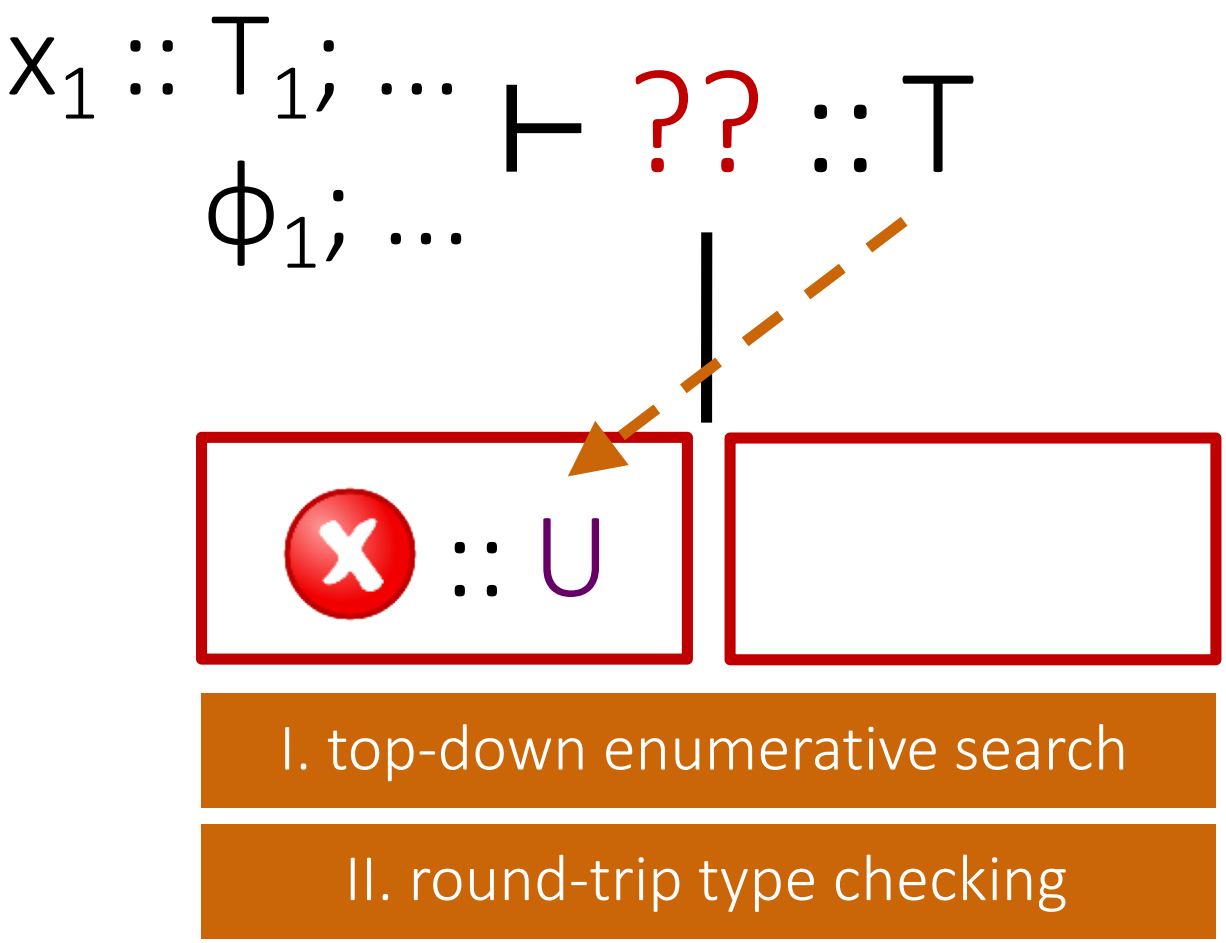
Synthesis from refinement types



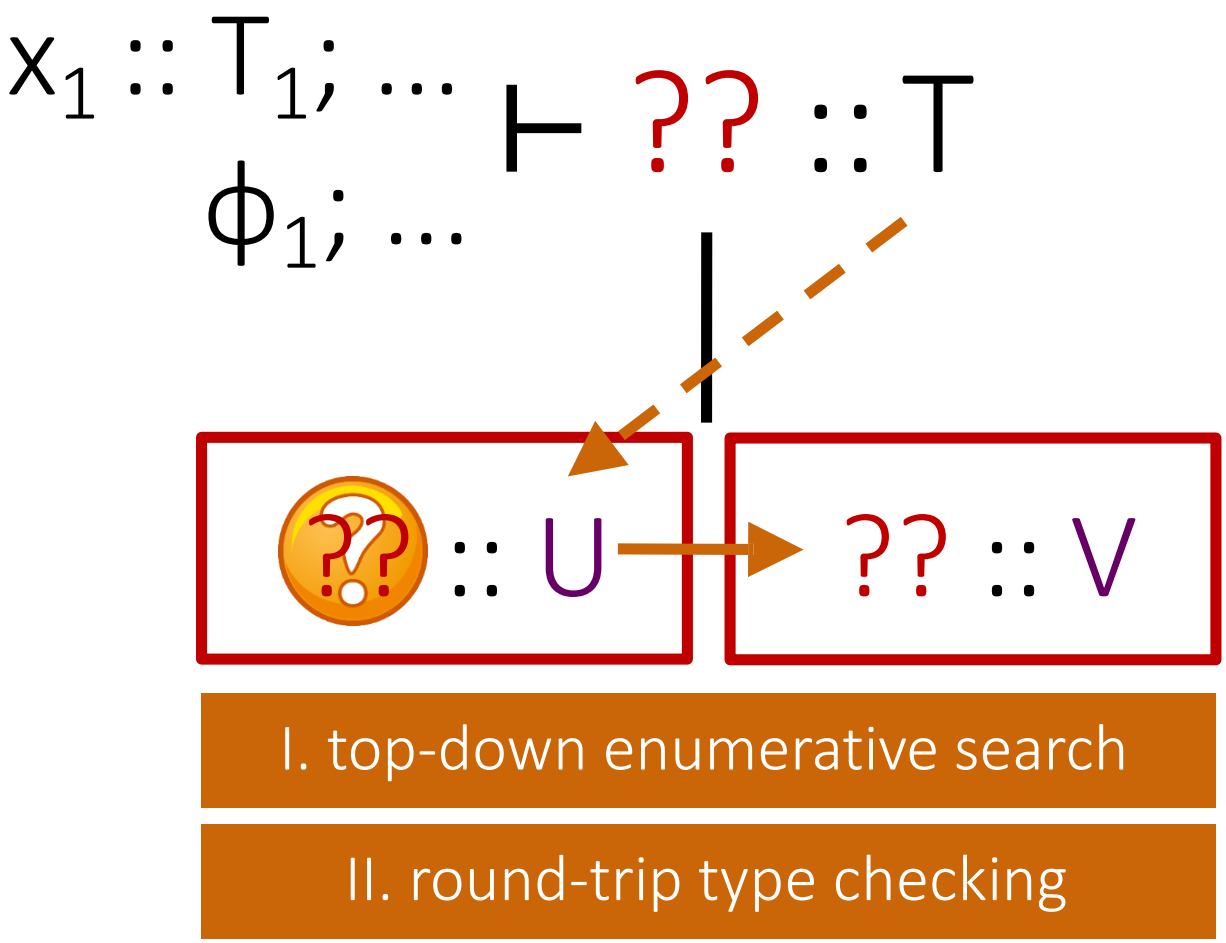
Synthesis from refinement types



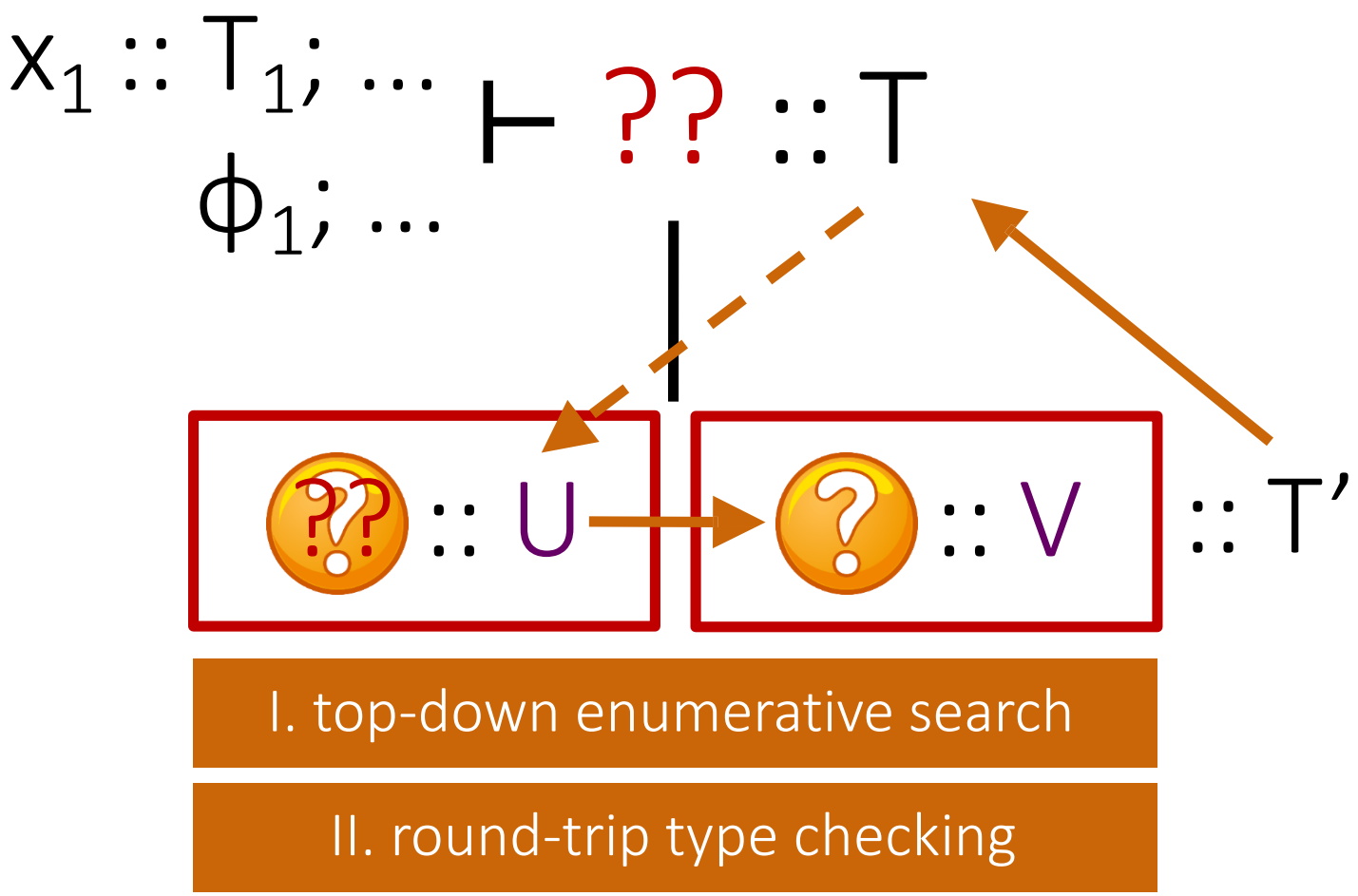
Synthesis from refinement types



Synthesis from refinement types

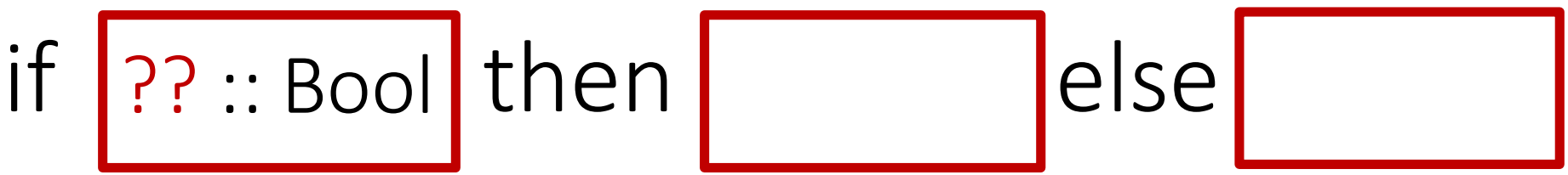


Synthesis from refinement types



Synthesis from refinement types

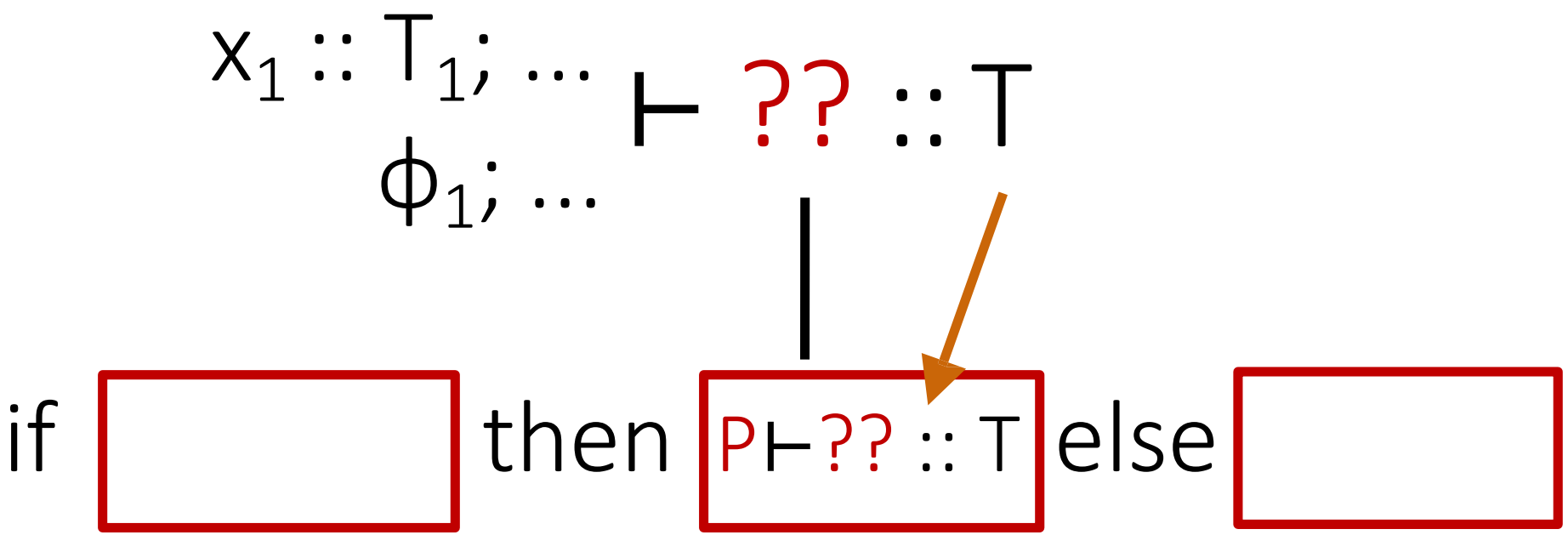
$x_1 :: T_1; \ldots$
$\phi_1; \ldots$
$\vdash$?? :: T



I. top-down enumerative search

II. round-trip type checking

Synthesis from refinement types

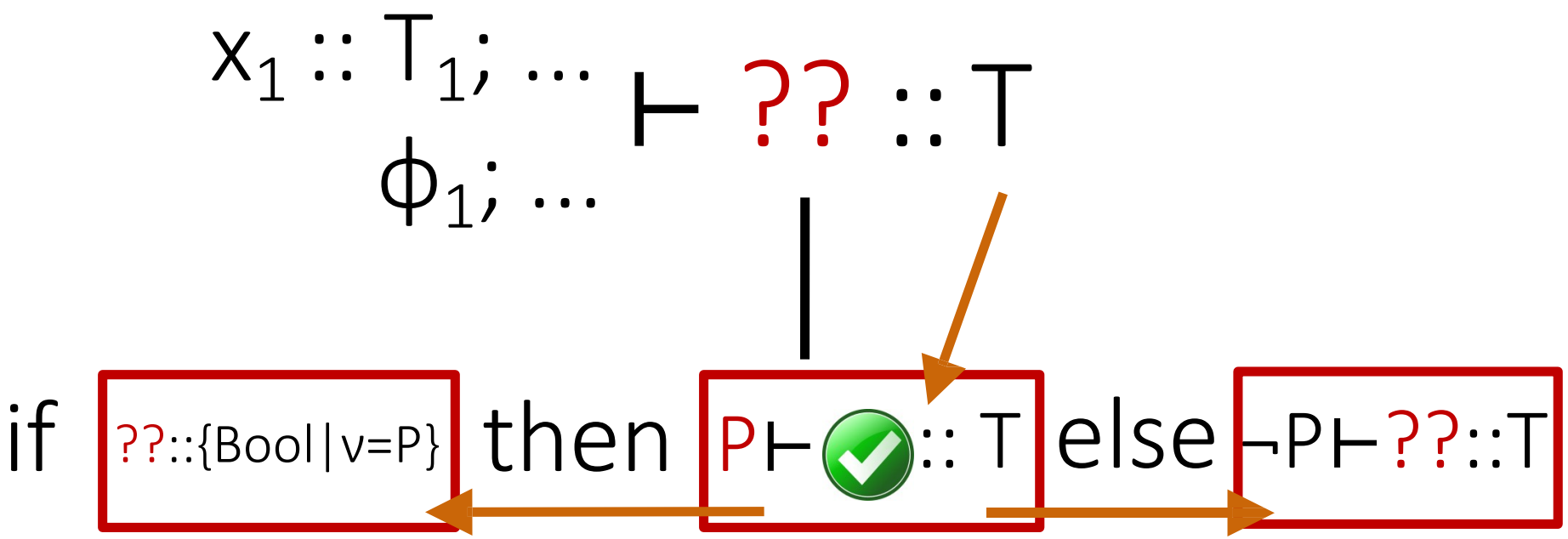


I. top-down enumerative search

II. round-trip type checking

III. condition abduction

Synthesis from refinement types



I. top-down enumerative search

II. round-trip type checking

III. condition abduction

Round-trip type checking

$$\Gamma \vdash ?? :: \{\text{List Neg} \mid \text{len } v \geq 5\}$$

Round-trip type checking

Nil ; 0 ; 5 ; -5
zeros
replicate ; Cons

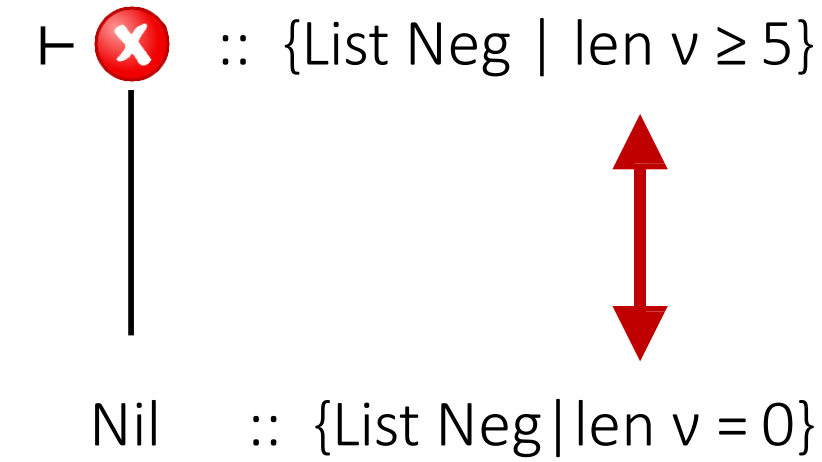
$\vdash$?? :: {List Neg | len v ≥ 5}

Round-trip type checking

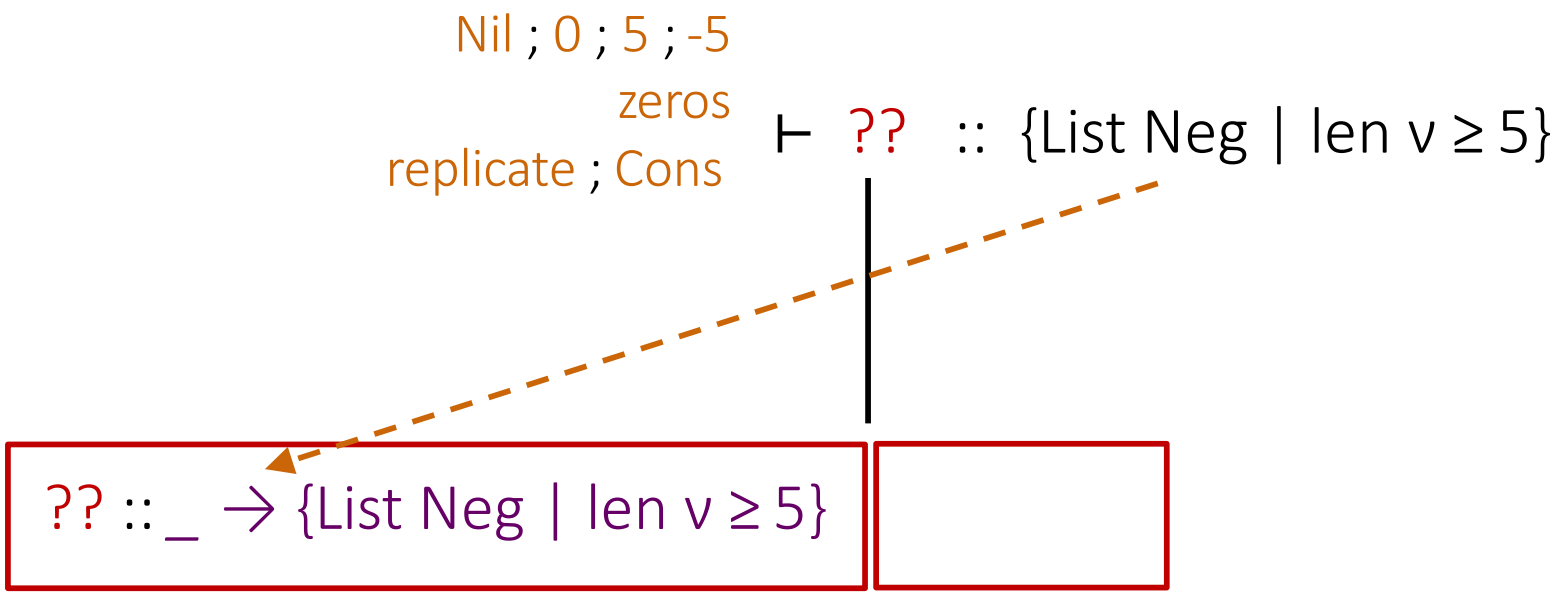
Nil :: {List a | len v = 0} ; 0 ; 5 ; -5

zeros

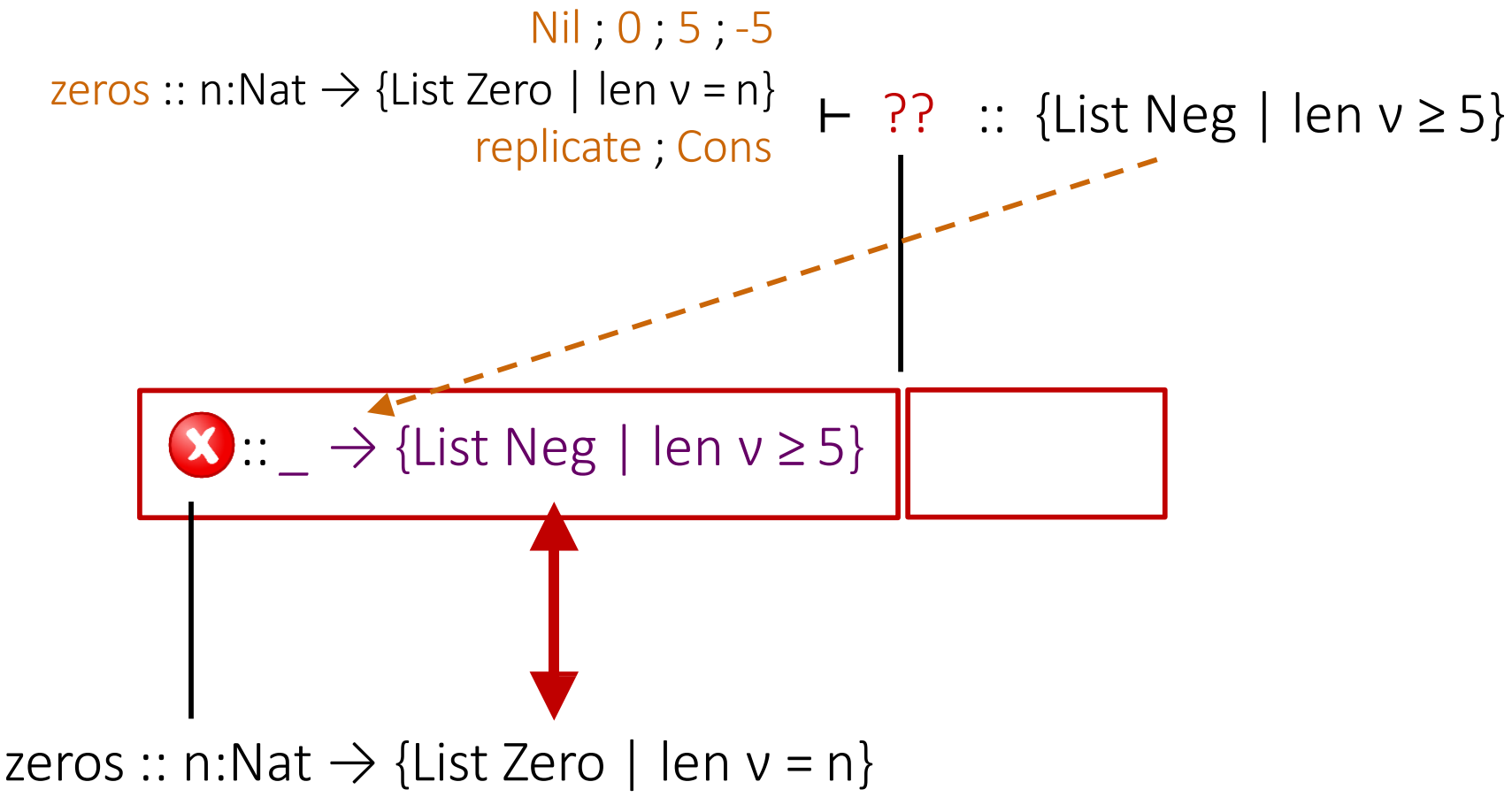
replicate ; Cons



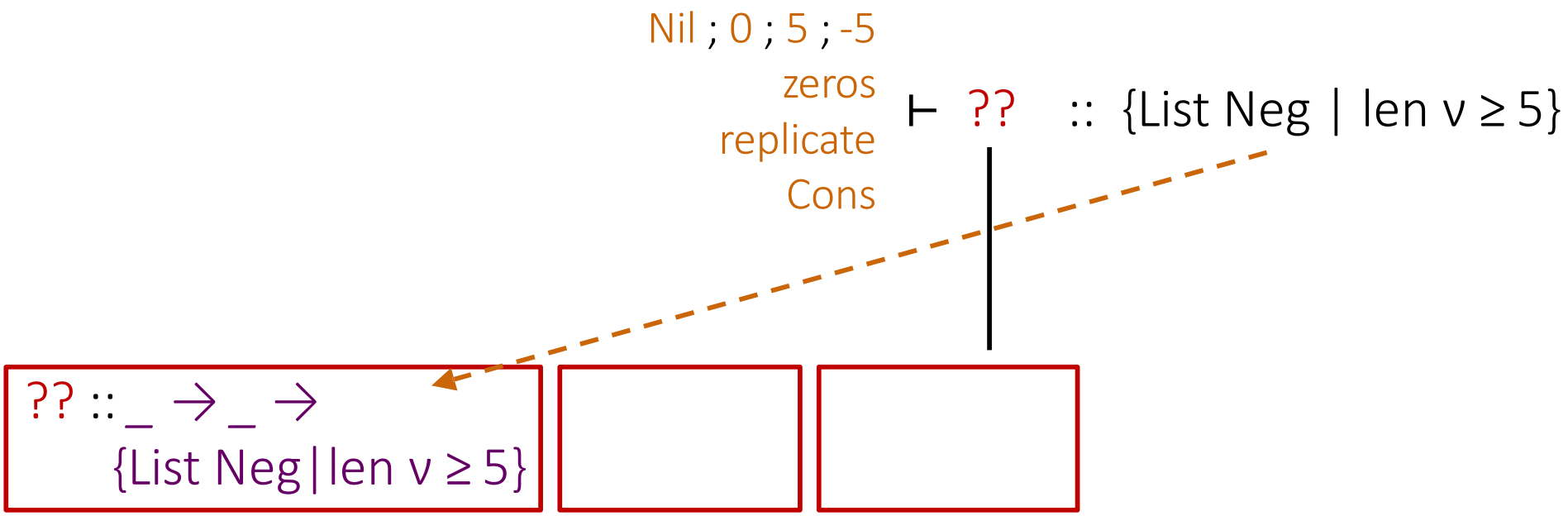
Round-trip type checking



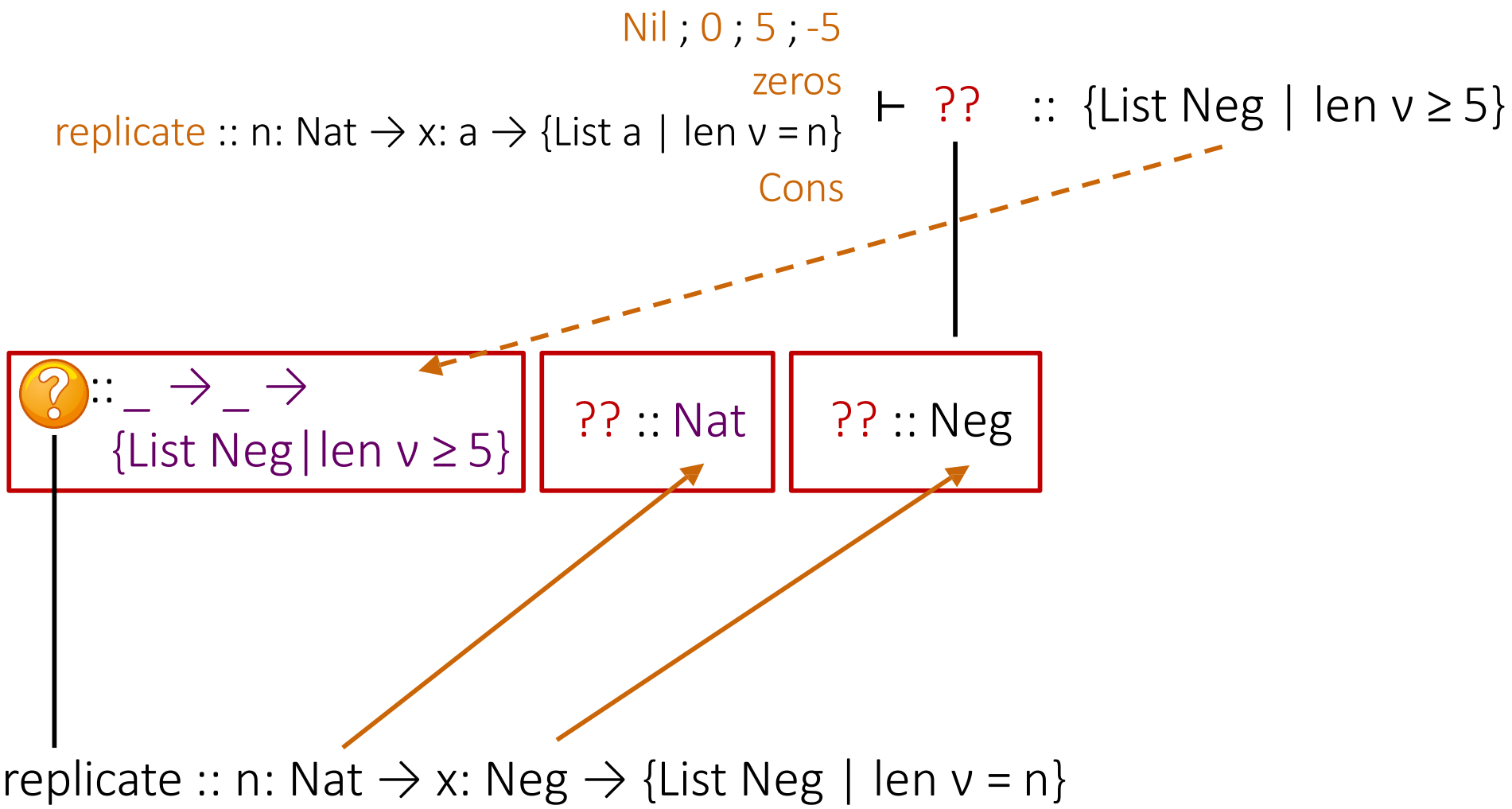
Round-trip type checking



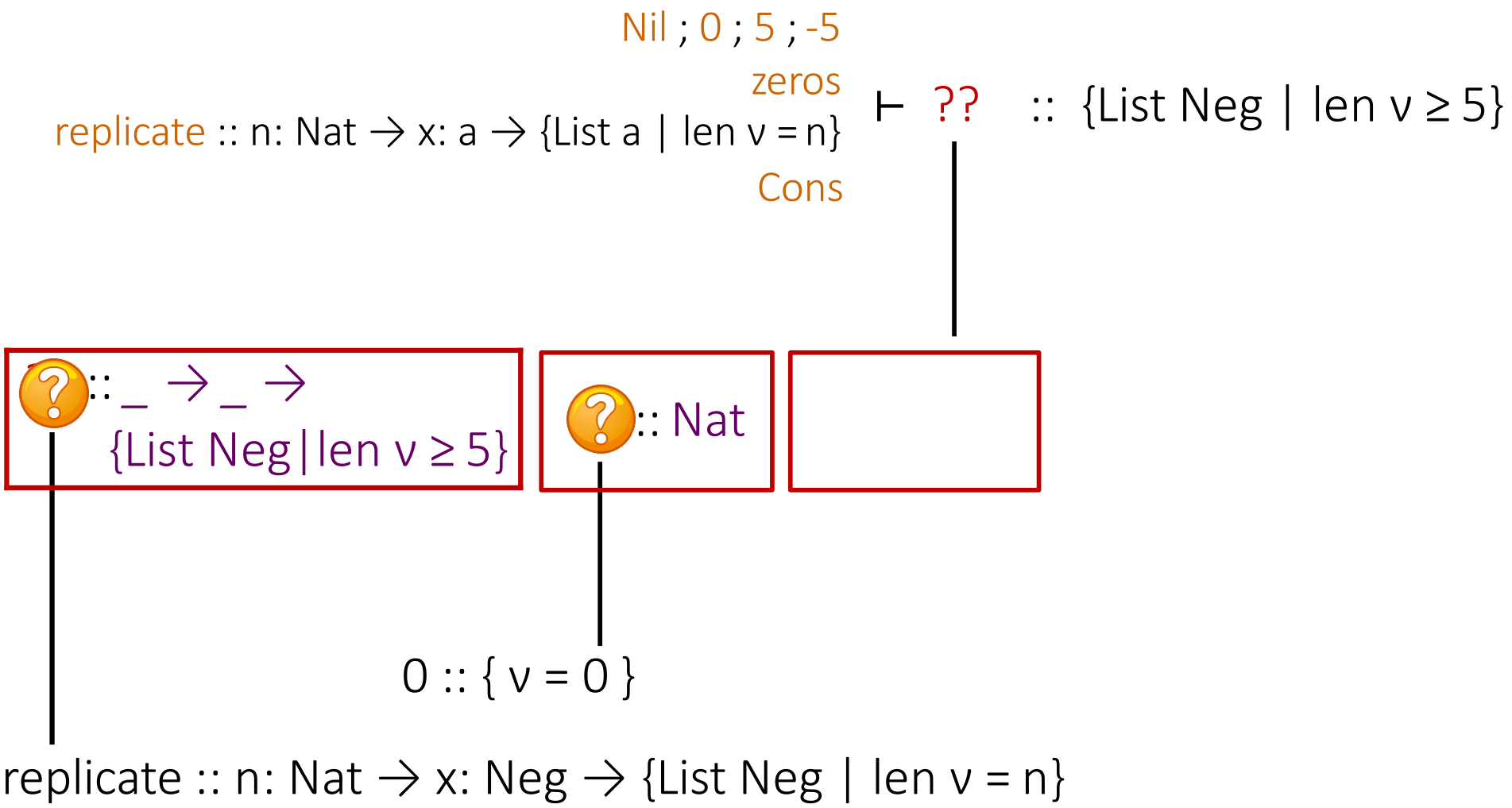
Round-trip type checking



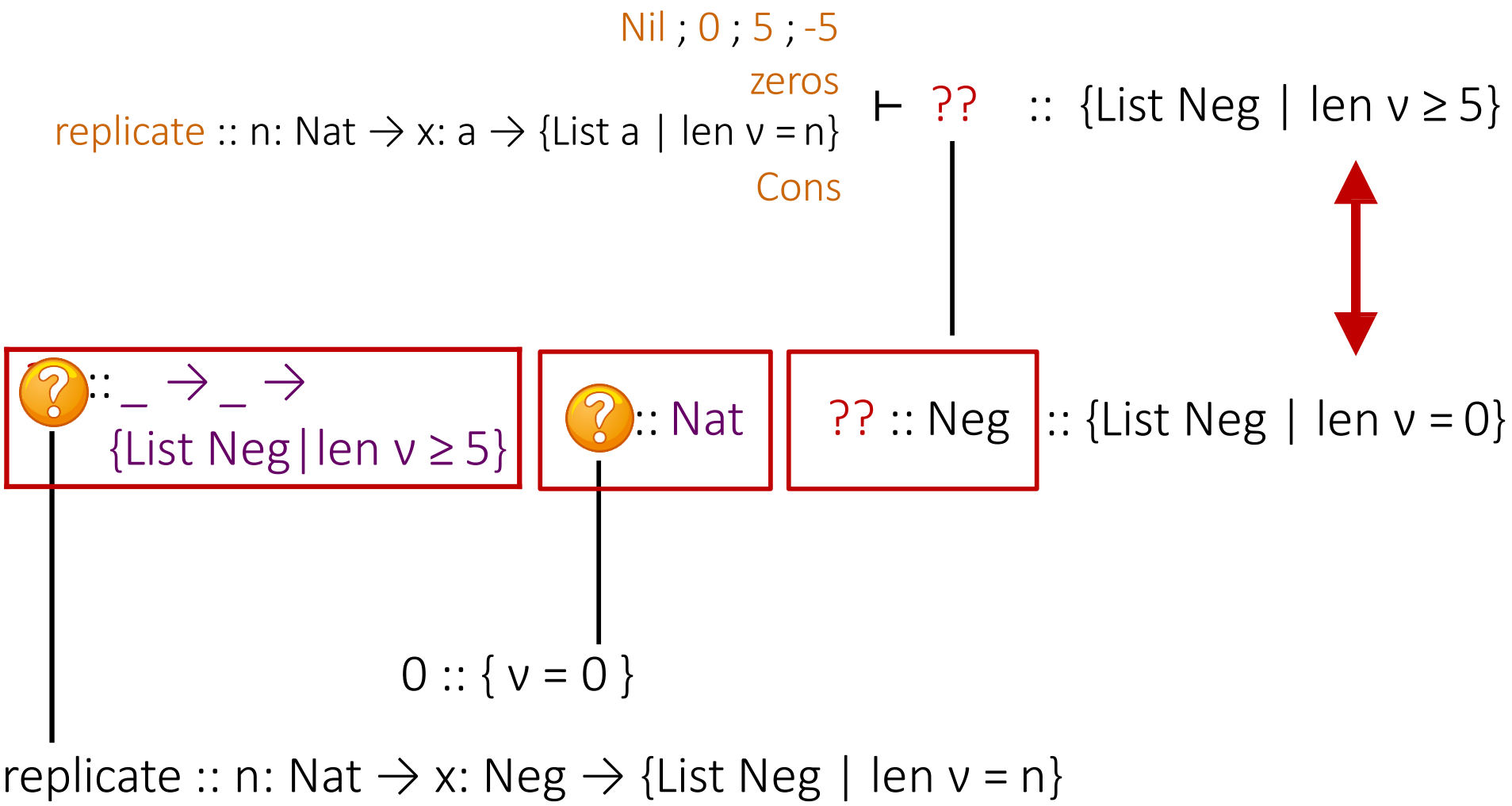
Round-trip type checking



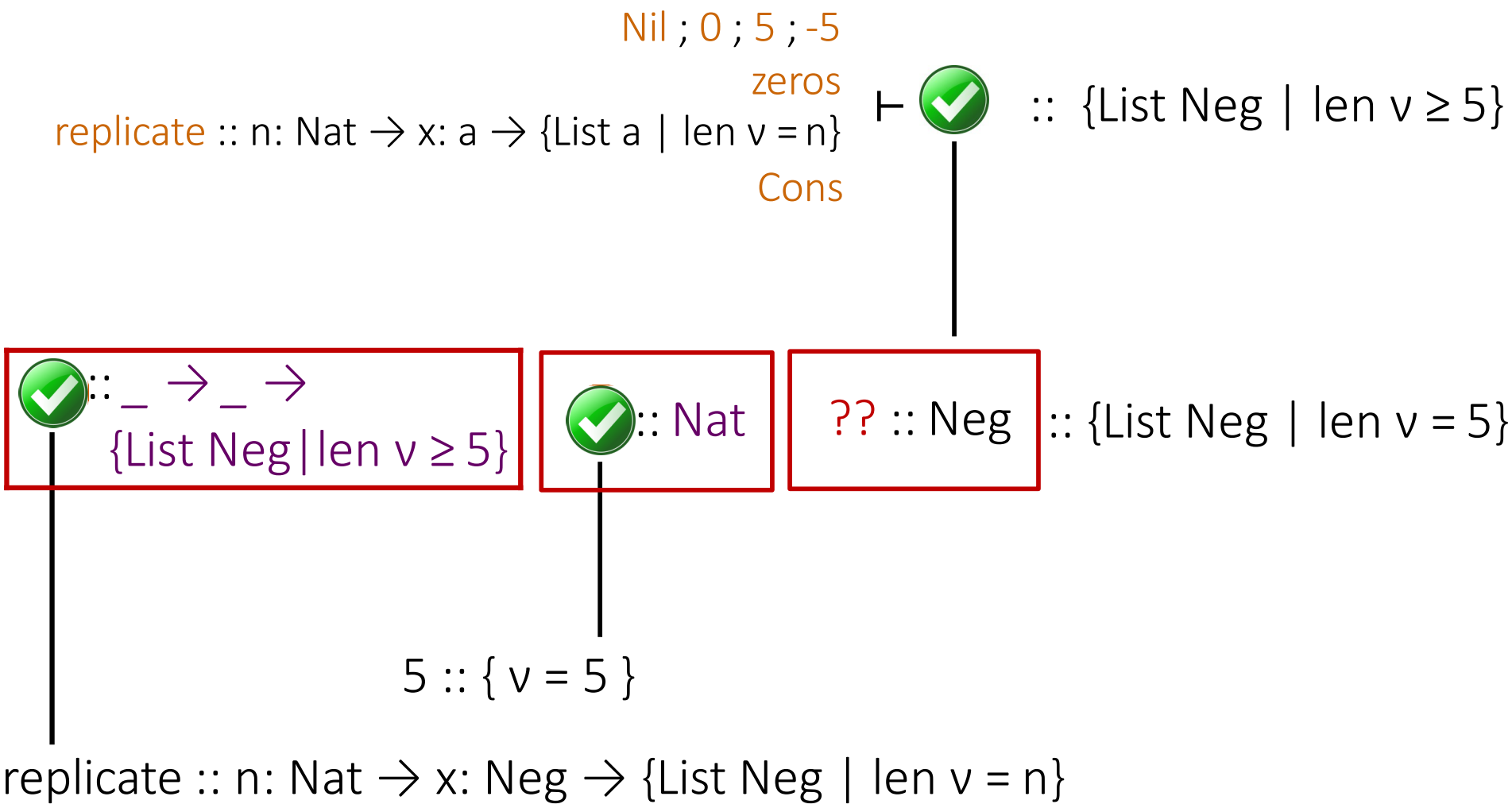
Round-trip type checking



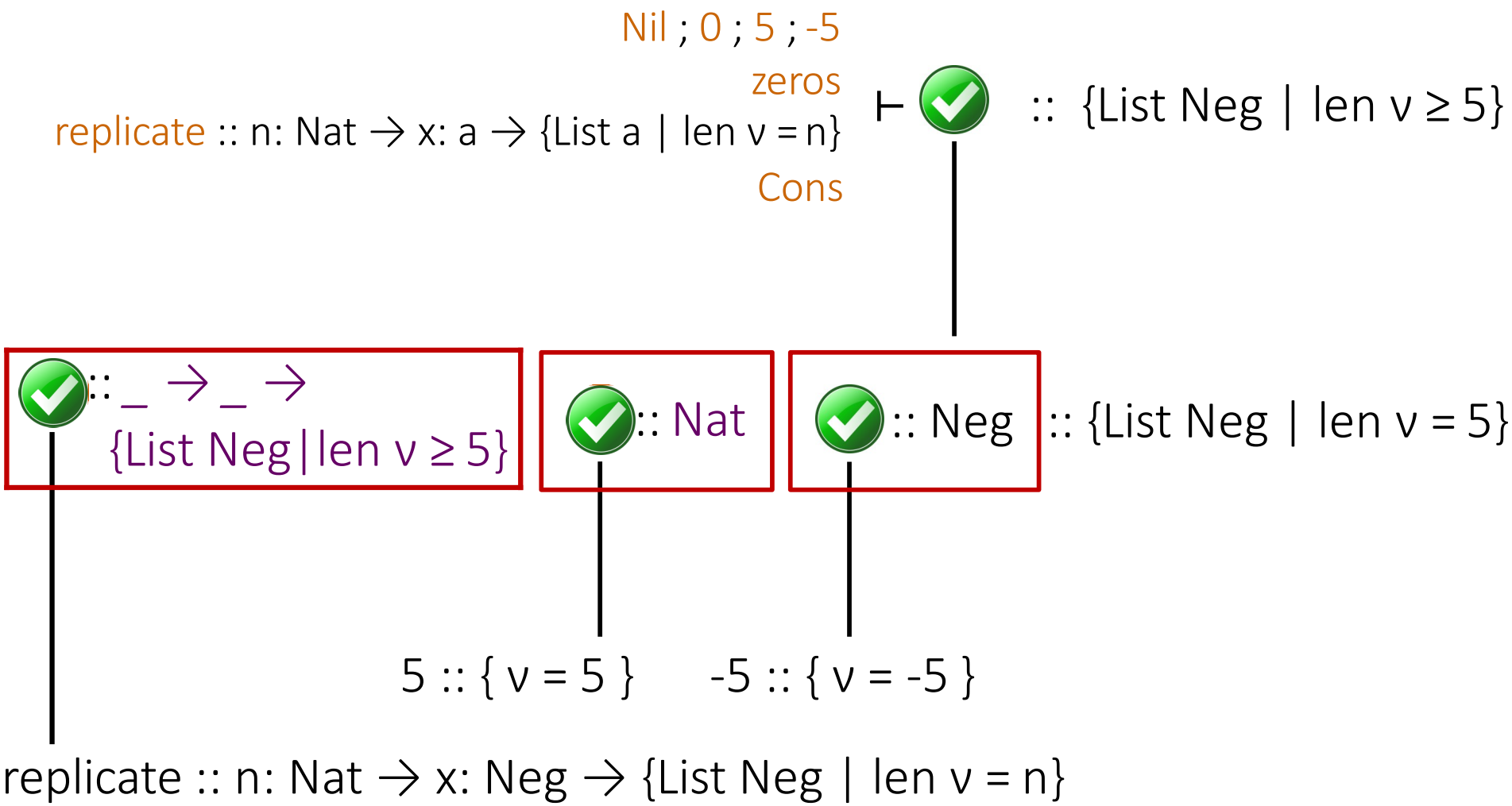
Round-trip type checking



Round-trip type checking



Round-trip type checking



Can RTTC prune away these terms?

`inc ?? :: {Int | v = 5}`

- where `inc :: x:Int → {Int | v = x + 1}`
- NO! don't know if we can find `?? :: {Int | v + 1 = 5}`

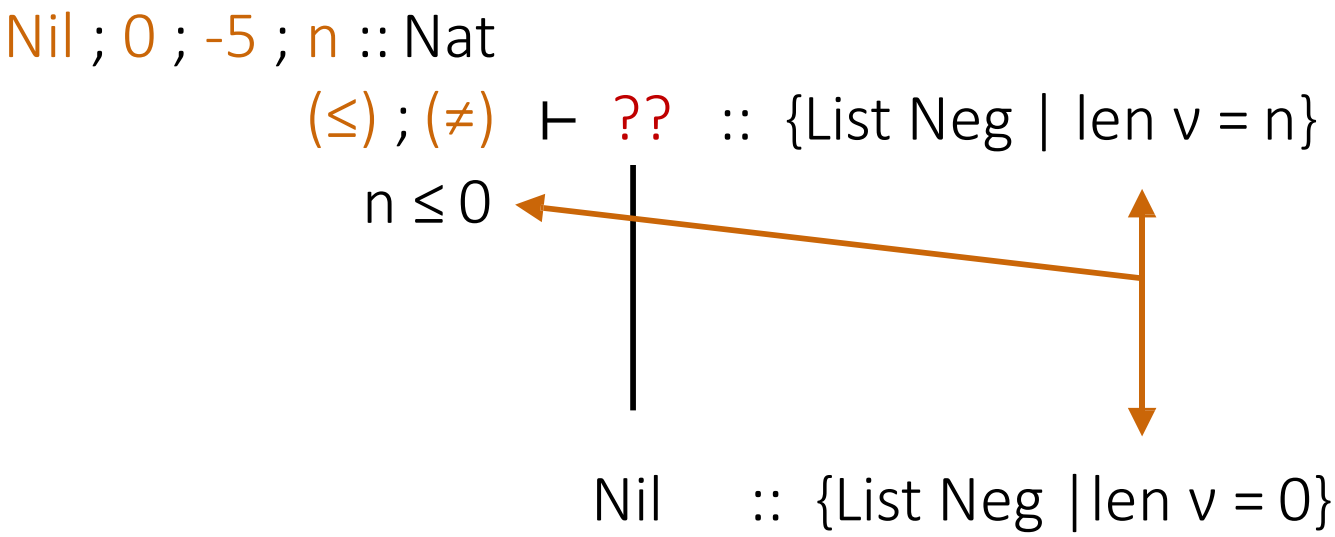
`nats ?? :: List Pos`

- where `nats :: n:Nat → {List Nat | len v = n}`
  `Nat = {Int | v >= 0}`, `Pos = {Int | v > 0}`
- YES! `n:Nat → {List Nat | len v = n}` not a subtype of
  `_ → List Pos`

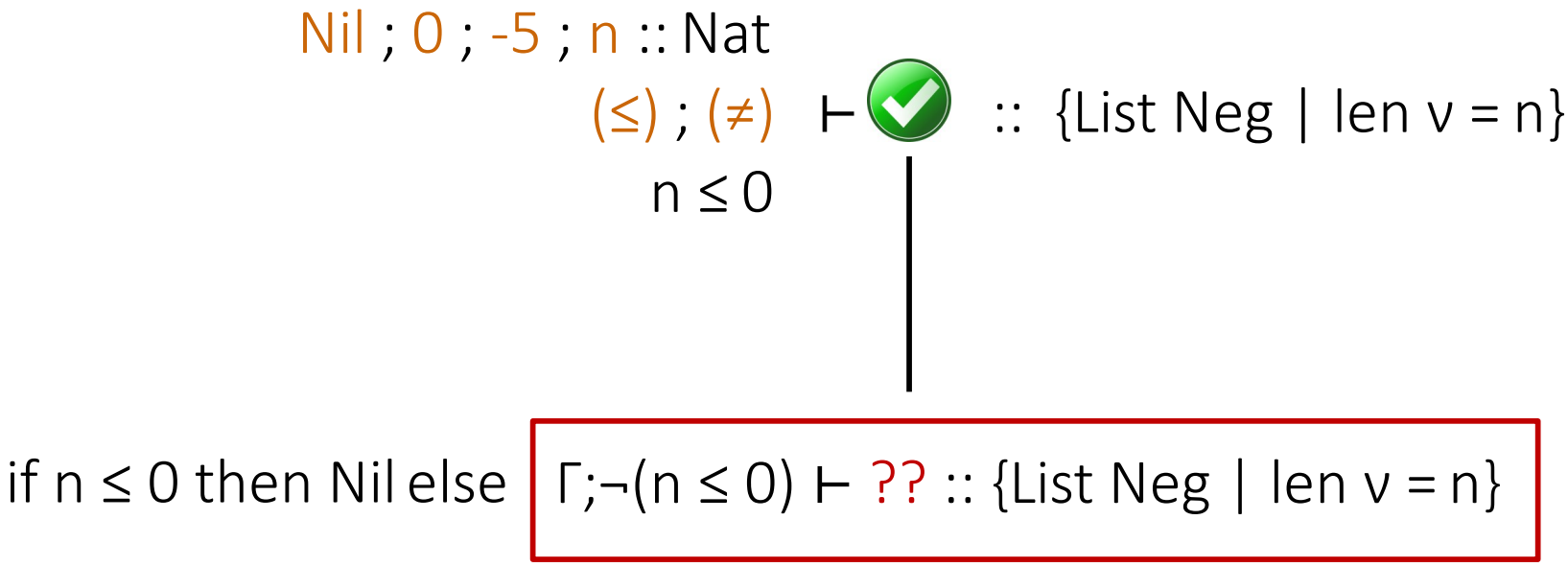
`duplicate ?? :: {List Int | len v = 5}`

- where `duplicate :: xs:List a → {List a | len v = 2*(len xs)}`
- YES! using a consistency check (len v = 2*(len xs) ∧ len v = 5 → UNSAT)

Condition abduction



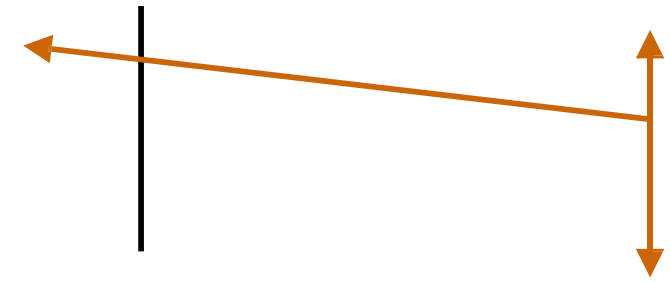
Condition abduction



Liquid abduction

Nil ; 0 ; -5 ; n :: Nat

(≤) ; (≠) ⊢ ?? :: {List Neg | len v = n}



Nil :: {List Neg | len v = 0}

---

n ≥ 0 ∧ len v = 0 ∧ P ⇒ len v = n

n :: Nat

Nil :: {List a | len v = 0}

Liquid abduction

$$n \geq 0 \ \wedge \ \text{len}\ v = 0 \ \wedge \ P \ \wedge \ \neg(\text{len}\ v = n)$$

$$\star \leq \star$$

$$\star \neq \star$$

Liquid abduction

$$n \geq 0 \ \wedge \ \text{len}\ \nu = 0 \ \wedge \ P \ \Rightarrow \ \text{len}\ \nu = n$$

$$n \leq 0$$
$$n \leq -5$$
$$-5 \leq n$$
$$n \neq 0$$
$$n \neq -5$$

Liquid abduction

UNSAT core

$$\left[\; n \geq 0 \;\wedge\; \text{len } v = 0 \;\wedge\; P \;\wedge\; \neg(\text{len } v = n) \;\right]$$

P:

$n \leq 0$

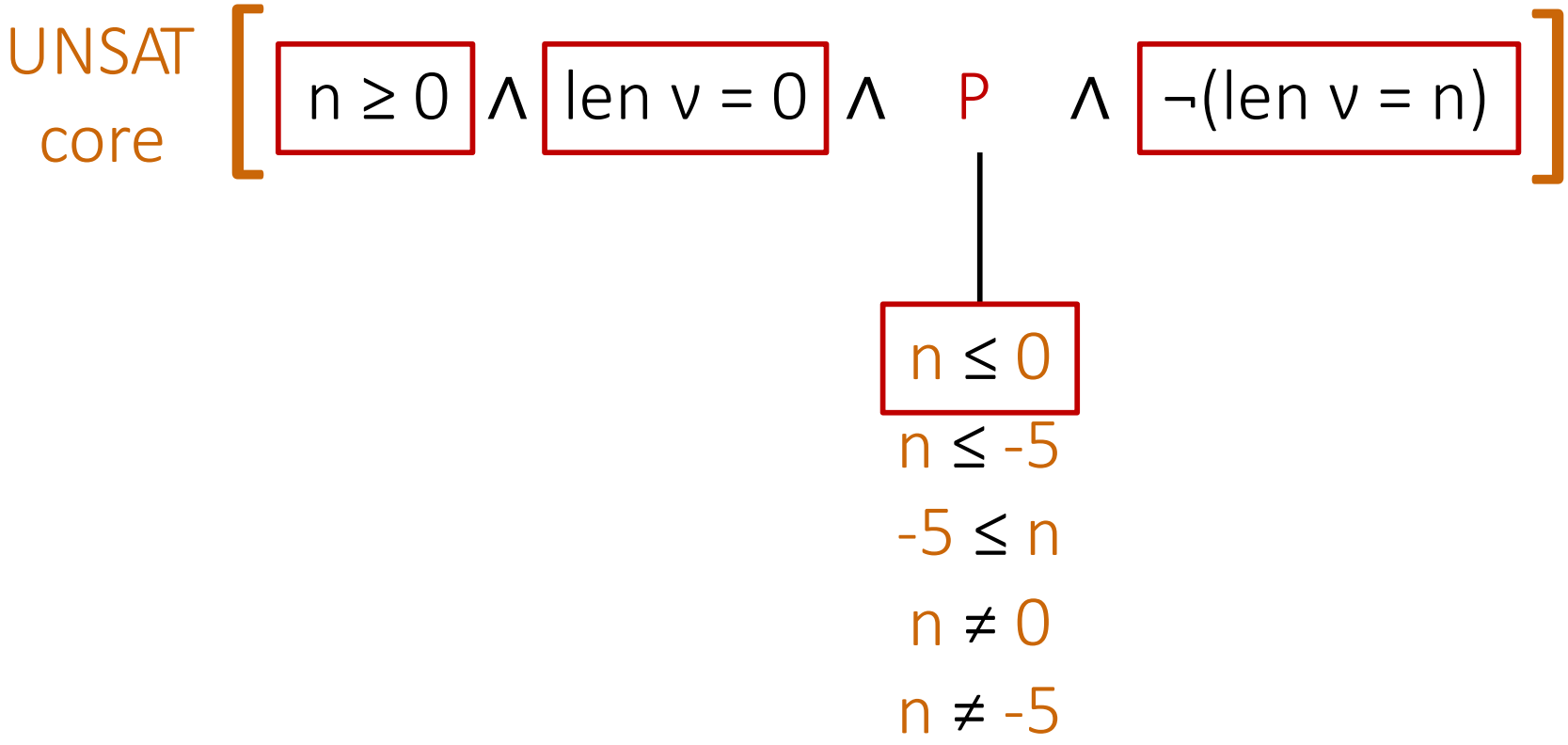
$n \leq -5$

$-5 \leq n$

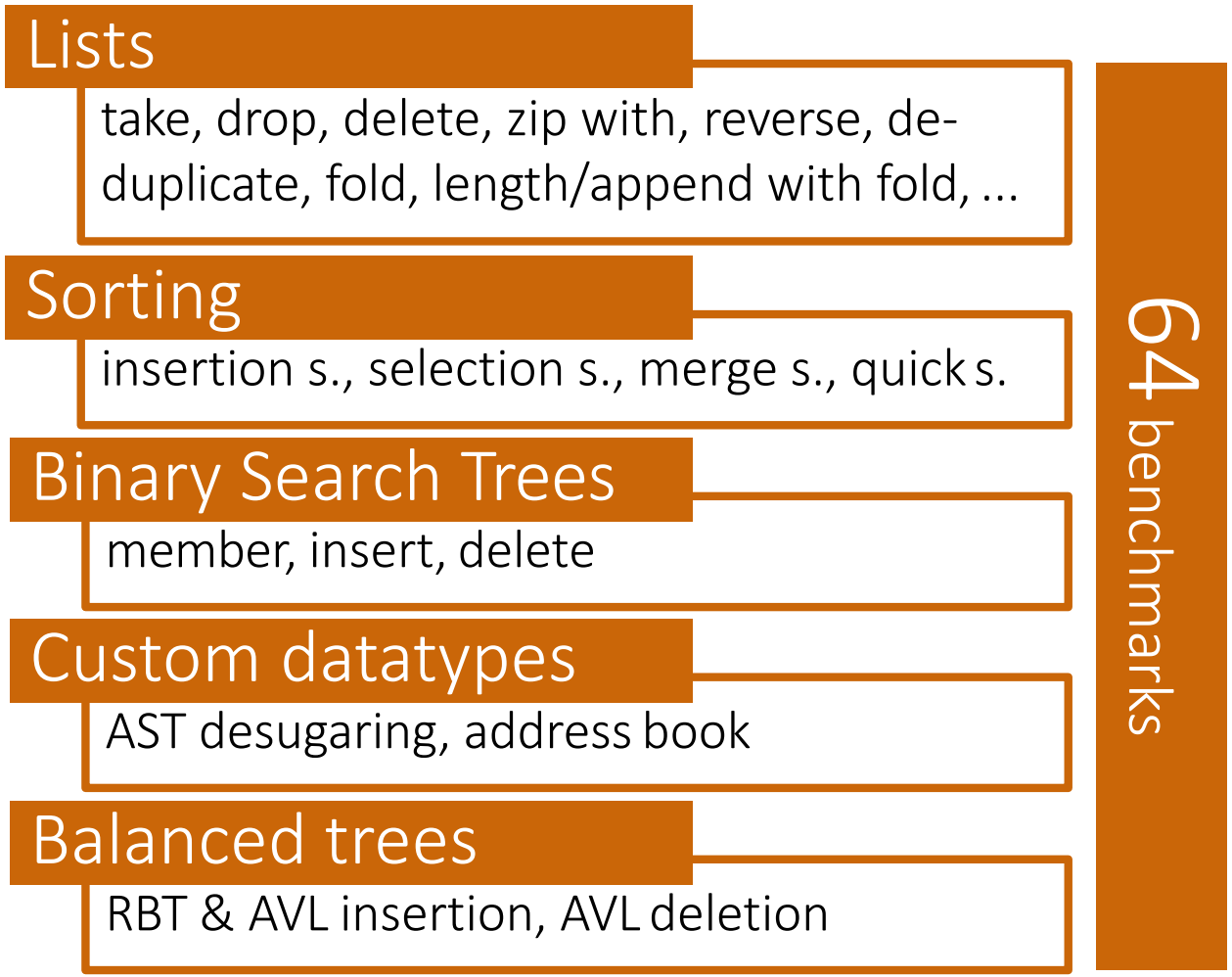
$n \neq 0$

$n \neq -5$

Liquid abduction



Evaluation


Lists
take, drop, delete, zip with, reverse, de-duplicate, fold, length/append with fold, ...
Sorting
insertion s., selection s., merge s., quick s.
Binary Search Trees
member, insert, delete
Custom datatypes
AST desugaring, address book
Balanced trees
RBT & AVL insertion, AVL deletion
64 benchmarks

Synquid: contributions

Unbounded correctness guarantees

Round-trip type system to reject incomplete programs

Refinement types can express complex properties in a simple way

- handles recursive, HO functions
- automatic verification for a large class of programs due to polymorphism (e.g. sorted list insert)

Synquid: limitations

User interaction

- refinement types can be large and hard to write
- components need to be annotated

Expressiveness limitations

- some specs are tricky or impossible to express
- cannot synthesize recursive auxiliary functions

Condition abduction is limited to liquid predicates

Cannot generate arbitrary constants

No ranking / quality metrics apart from correctness

Synquid: questions

Behavioral constraints? Structural constraints? Search strategy?

- Refinement types
- Set of components + built-in language constraints
- Top-down enumerative search with type-based pruning

Synthesis of recursive programs