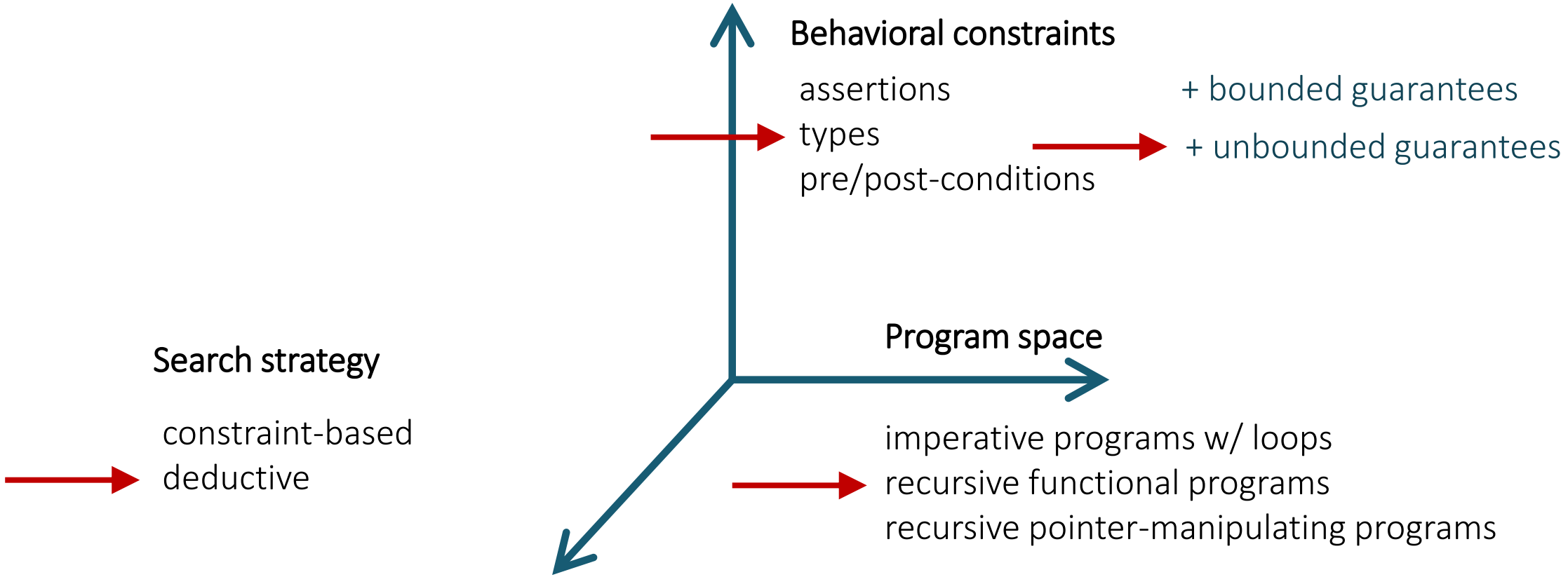
Lecture 12
Type-Driven Synthesis

Nadia Polikarpova

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

Tuesday:

- 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

Specification for insert

Input:

x

xs: sorted list

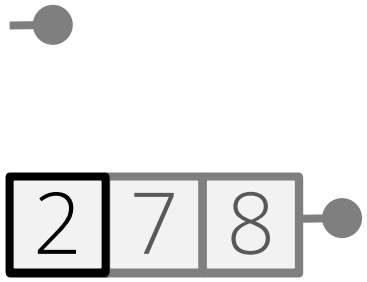
Output:

ys: sorted list

elems ys = elems xs ∪ {x}

Refinement types: sorted lists

```
data List a where
  Nil :: List a
  Cons :: h:a →
          t:List a →
          List a
```



all you need
is one simple predicate!

Refinement type for insert

```
insert :: x:a  →  xs:List a   →
                  List a
```

Refinement types as specs

```
// Insert x into a sorted list xs
insert :: x:a  →  xs:SList a  →
          {v:SList a | elems v = elems xs ∪ {x}}
insert x xs =
  match xs with
    Nil → Nil
    Cons h t →
      if x ≤ h
        then Cons x xs
        else Cons h (insert x t)
```

Expected
{v:SList e|elems v = elems xs ∪ {x}}
and got
{v:SList e|elems xs ⊆ elems v}

Insert in Synquid

specification

```
insert :: x:a →
  xs:SList a →
  {v:SList a | elems v =
    elems xs ∪ {x}}
```

code

```
match xs with
 Nil → Cons x Nil
 Cons h t →
   if x ≤ h
     then Cons x xs
     else Cons h (insert x t)
```

Agenda

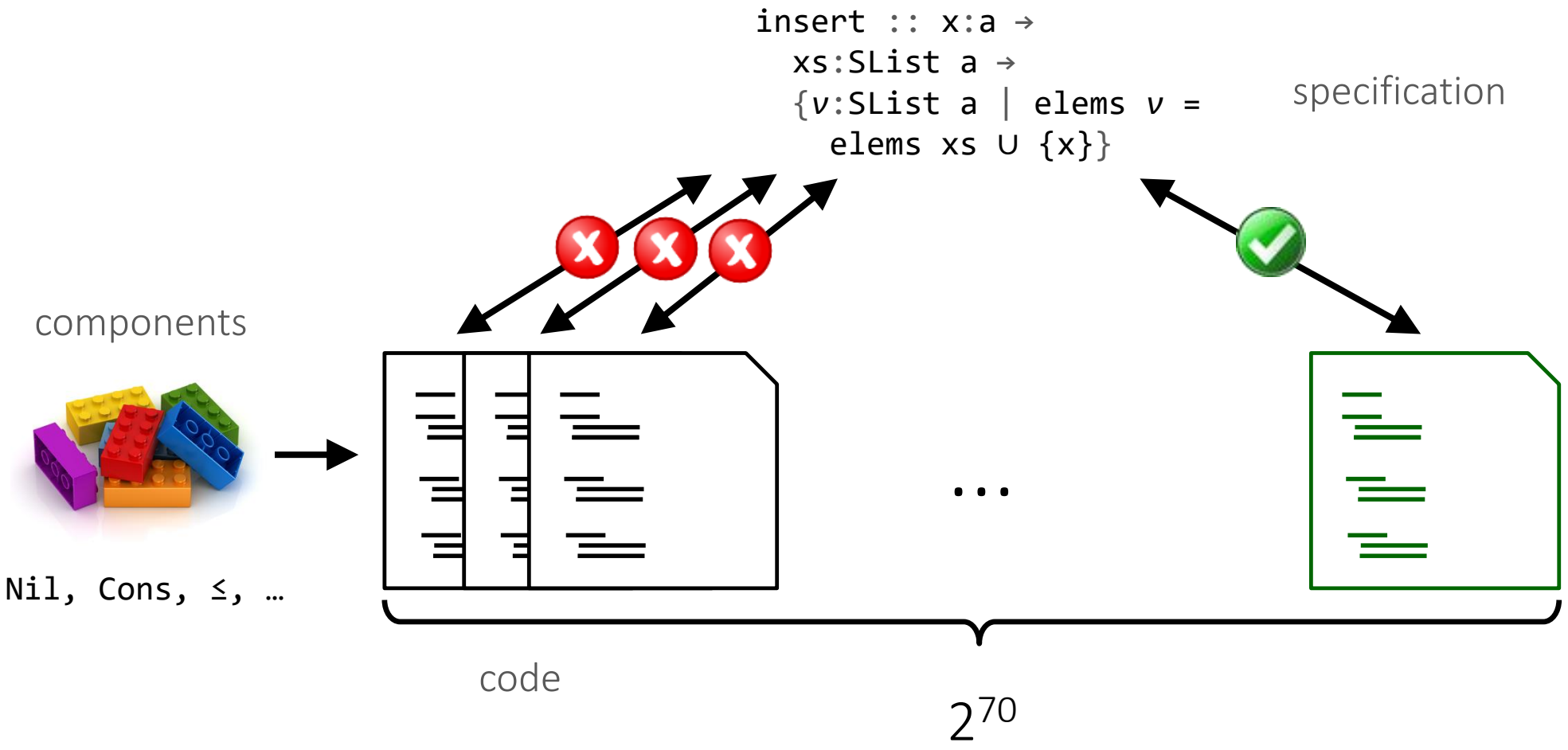
Tuesday:

- 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

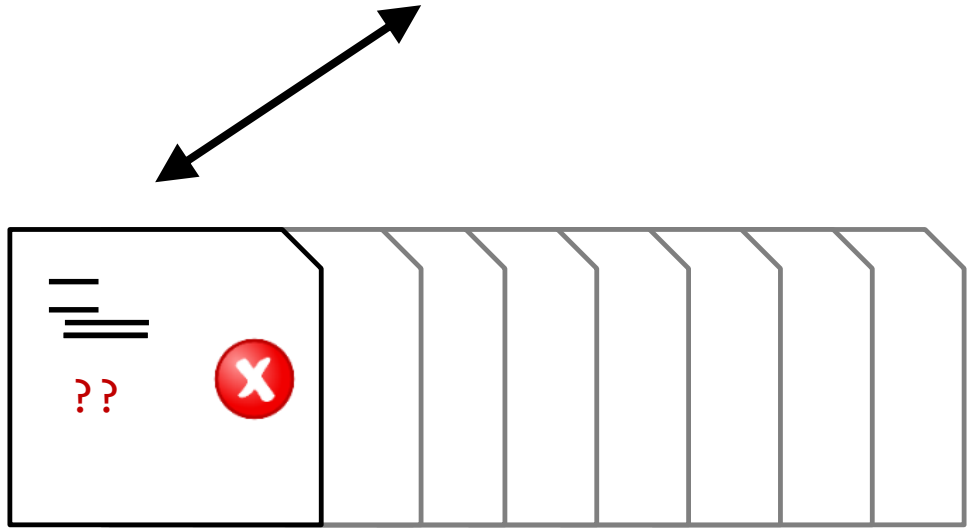
Enumerate and check



Idea: reject hopeless programs

```
insert :: x:a →
  xs:SList a →
  {v:SList a | elems v =
    elems xs ∪ {x}}
```

specification



Rejecting hopeless programs

```
x:a  →  xs:SList a  →
  {v:SList a | elems v = elems xs ∪ {x}}
```

```
insert x xs =  ??
  match xs with
    Nil →  ??
    Cons h t →
```

2^{50}

hopeless:
output must always contain x!

Bidirectional type checking

```
{v:SList a | elems v = elems xs ∪ {x}}
```

```
insert x xs =
  match xs with
    Nil → Nil
    Cons h t → ??
```

Constraints:
∀x: {} = {} ∪ {x}

SMT solver: INVALID!

Round-trip type checking

```
x:a  →  xs:SList a  →
  {v:SList a | elems v = elems xs ∪ {x}}
```

```
insert x xs =
  match xs with
    Nil → Cons x Nil
    Cons h t →
      Cons h (insert x ??)
```

hopeless:
cannot guarantee
output is sorted!

Round-trip type checking

$\{\nu{:}a \mid h \leq \nu\}$

```
insert x xs =
  match xs with
    Nil → Cons x Nil
    Cons h t →
      Cons h (insert x ??)
```

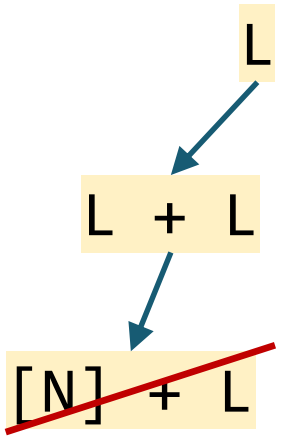
Constraints:
$\forall x, h: h \leq x$

SMT solver: INVALID!

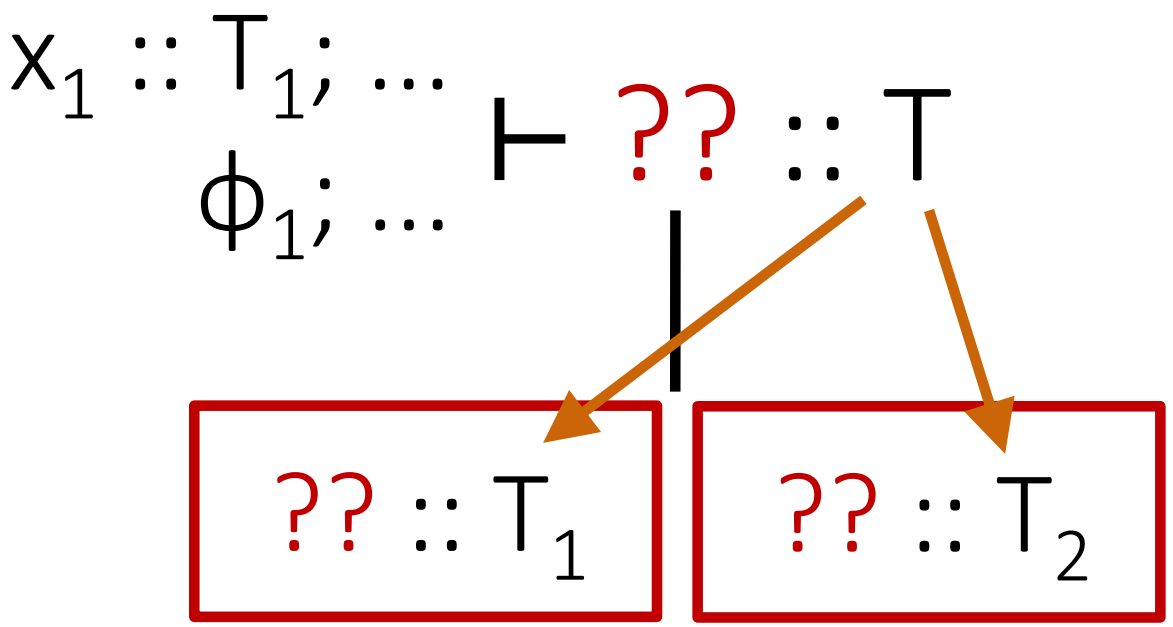
```
insert :: x:τ →
  xs:SList τ →
  SList τ
```

Type-driven synthesis

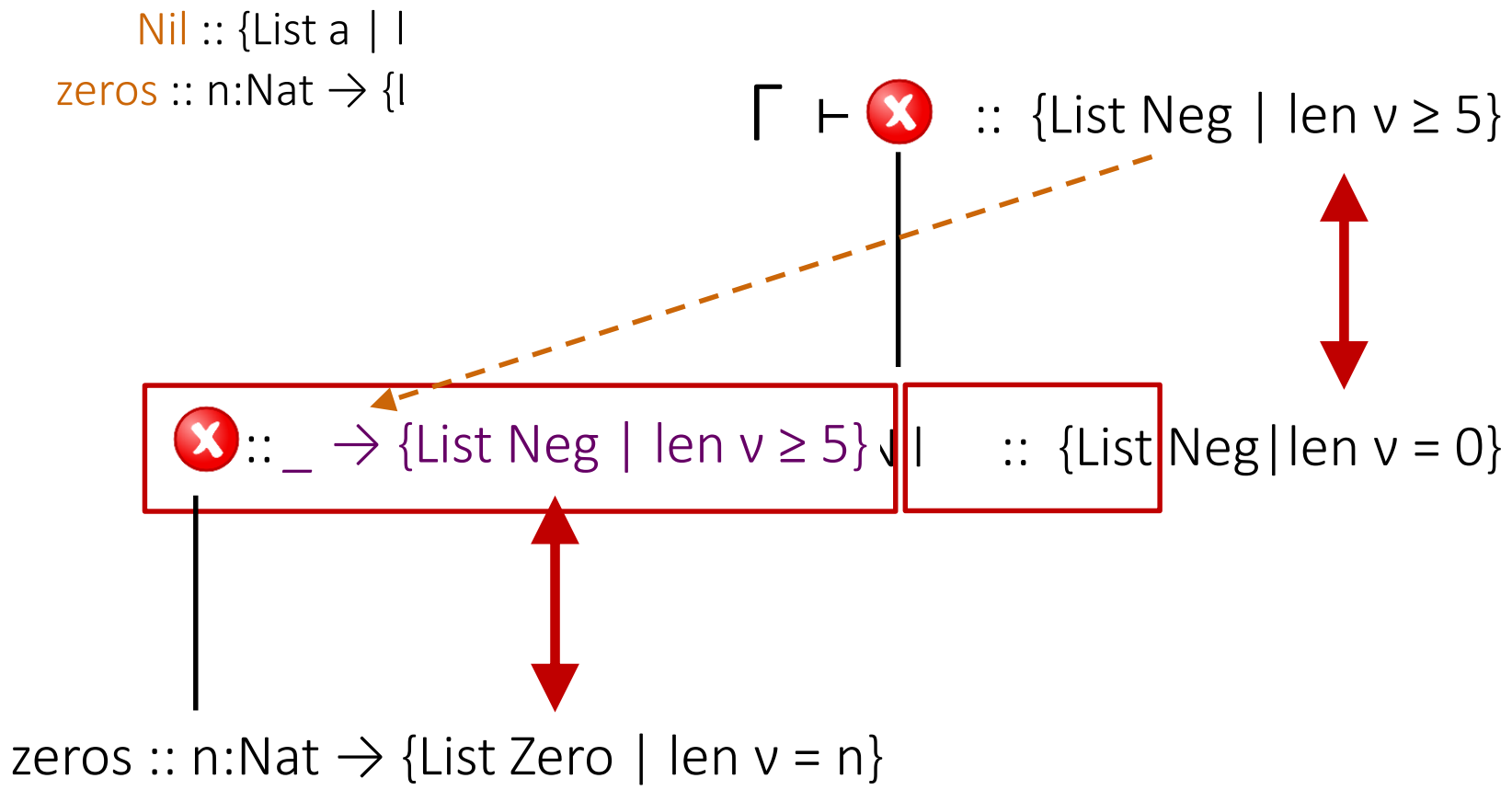
Top-down enumerative search
+
Use type-checking for top-down propagation



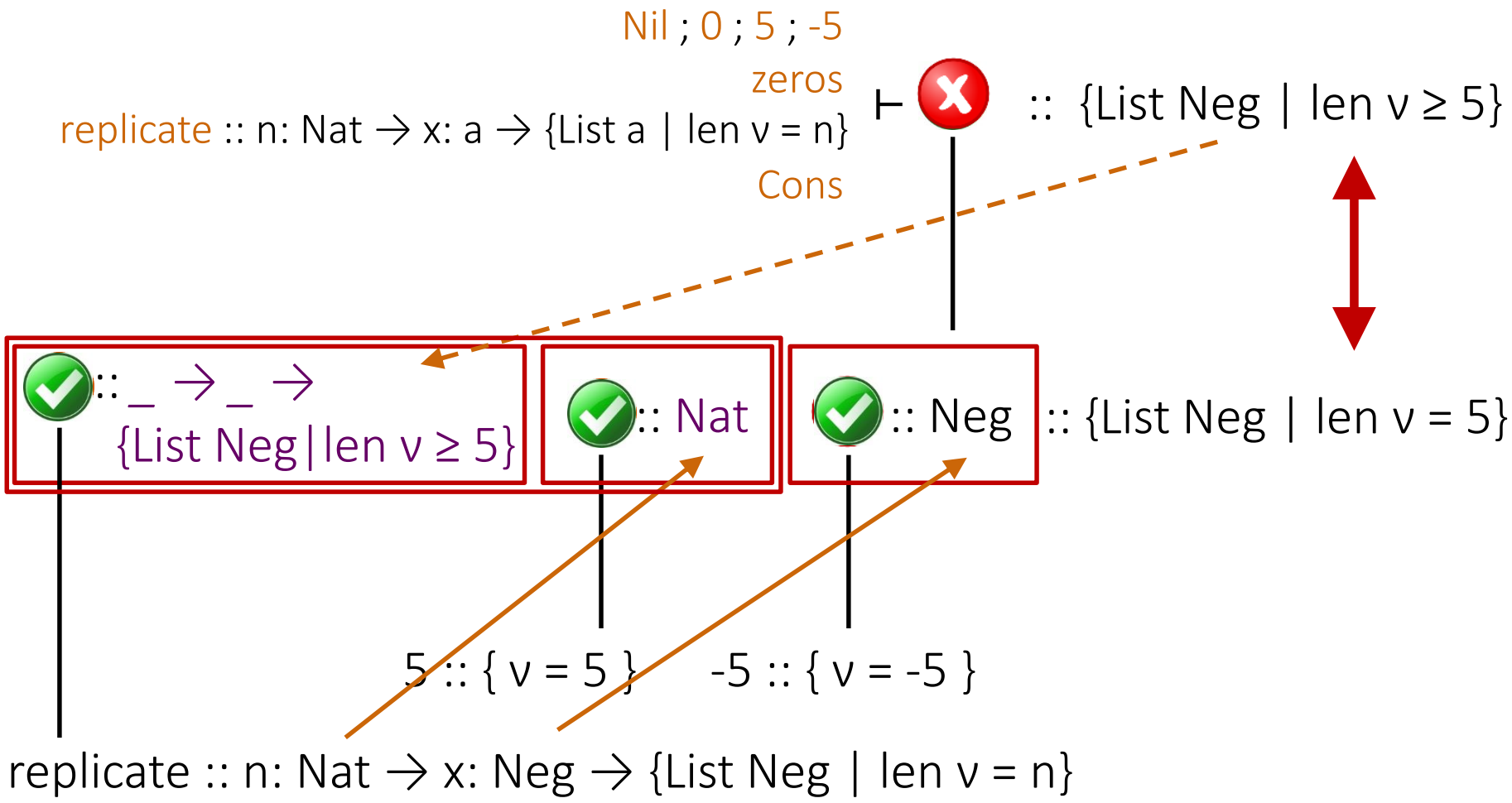
Synthesis from refinement types



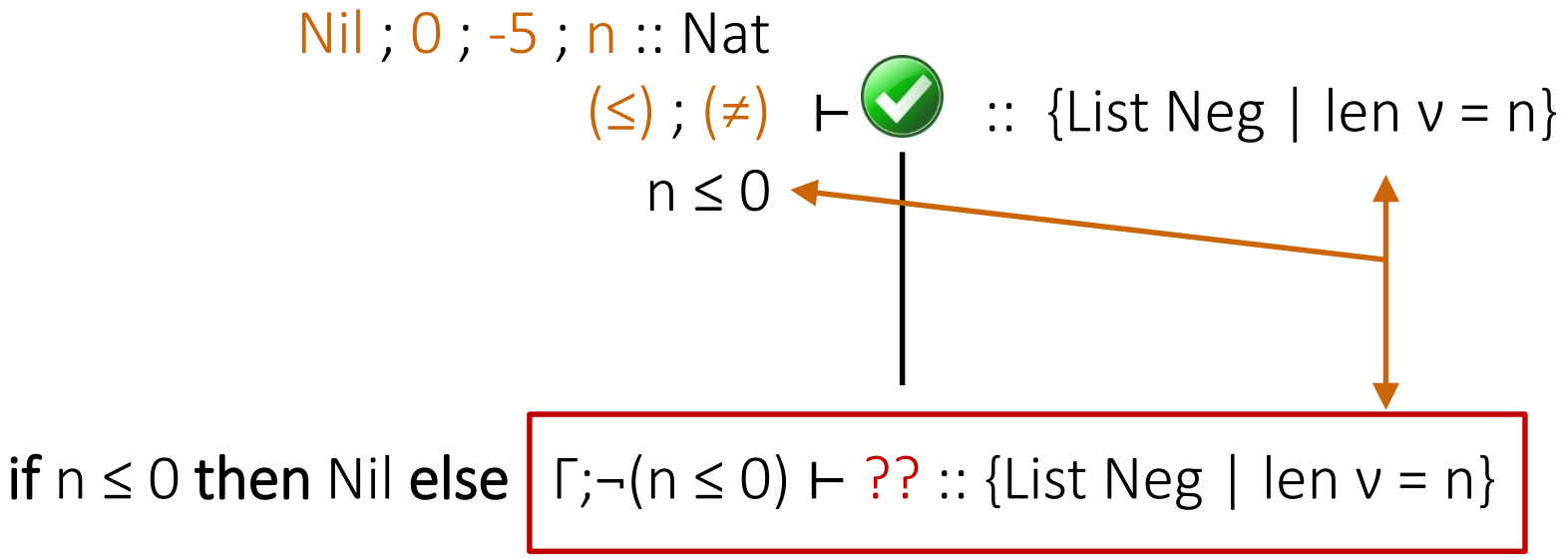
Example


Nil :: {List a | l
zeros :: n:Nat → {l
⌜ ⊢ :: {List Neg | len v ≥ 5}
:: _ → {List Neg | len v ≥ 5}
:: {List Neg | len v = 0}
zeros :: n:Nat → {List Zero | len v = n}

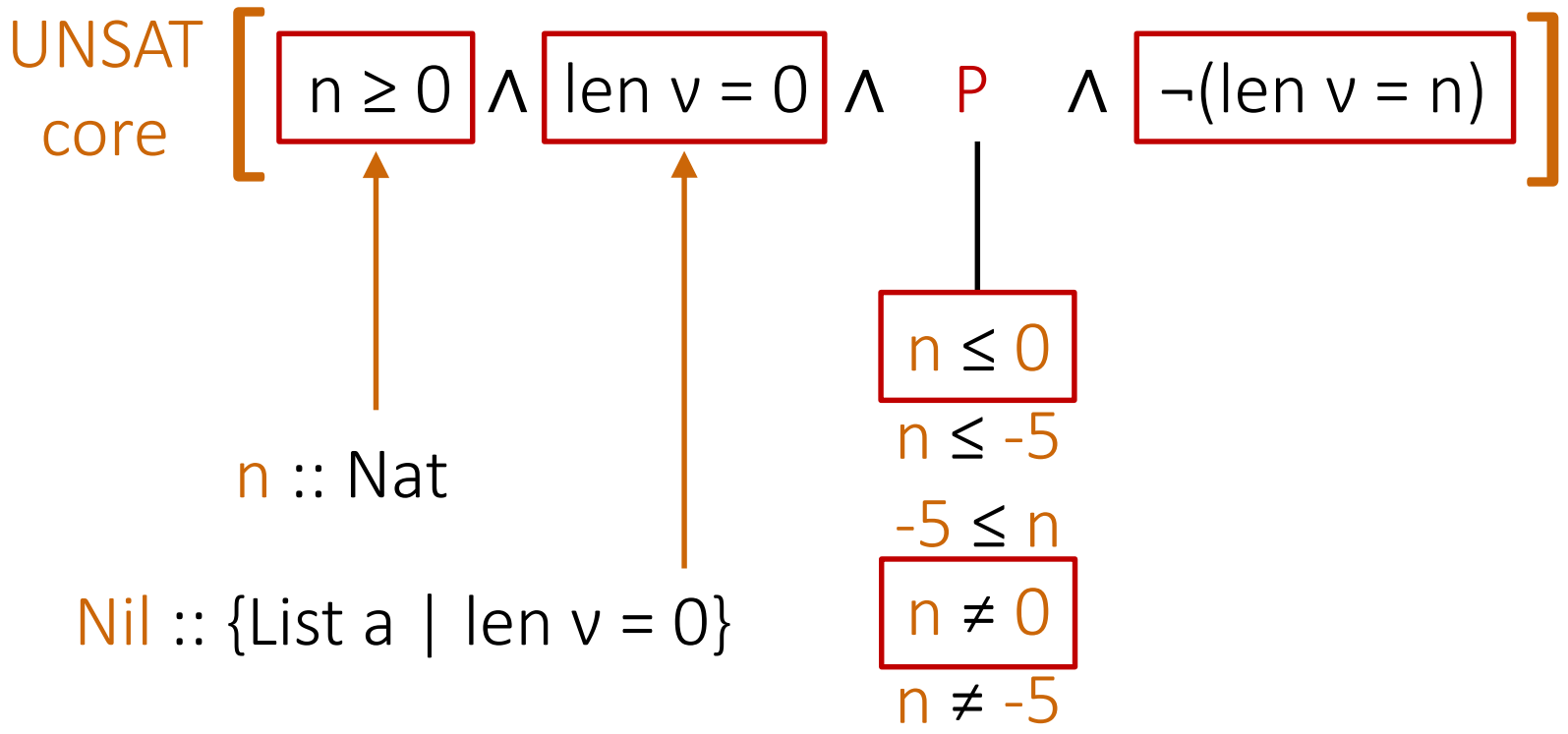
Example


Nil ; 0 ; 5 ; -5
zeros
replicate :: n: Nat → x: a → {List a | len v = n}
Cons
⊢
:: {List Neg | len v ≥ 5}
:: _ → _ →
{List Neg | len v ≥ 5}
:: Nat
:: Neg
:: {List Neg | len v = 5}
5 :: { v = 5 }
-5 :: { v = -5 }
replicate :: n: Nat → x: Neg → {List Neg | len v = n}

Condition abduction



Liquid abduction



Synquid: contributions

Unbounded correctness guarantees

Round-trip type system to reject incomplete programs

- + GFP Horn Solver

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 (how to mitigate?)

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

Typo in the example in Section 3.2

- $\{B_0 \mid \bot\} \rightarrow \{B_1 \mid \bot\} \rightarrow \{\text{List Pos} \mid \text{len } \nu = \cancel{2}5\}$

Can RTTC reject 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)