

Lecture 15

Refinement Types and Type-Driven Synthesis

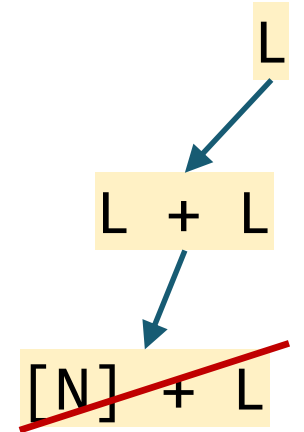
Nadia Polikarpova

Motivation

Goal: use deductive reasoning for top-down propagation

- prune unverifiable candidates early
- need synthesis-friendly verification technique!

Observation: type checkers are good at rejecting incomplete programs!



Running example

```
// Insert x into a sorted list xs  
insert :: x:e → xs:List e → List e  
insert x xs =
```

```
  match xs with
```

```
    Nil →
```



```
    Cons h t →
```

```
      if x ≤ h
```

```
      then Cons x xs
```



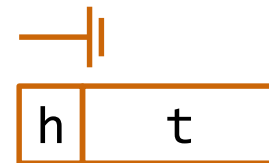
```
      else Cons h (insert x t)
```



```
data List e where
```

```
  Nil :: List e
```

```
  Cons :: h:e → t:List e → List e
```



Rejecting incomplete programs

[Pierce, Turner. TPLS'00]

```
// Insert x into a sorted list xs
insert :: x:e → xs:List e → List e
insert x xs =
  match xs with
    Nil → Cons xs ...
    ...
```



bidirectional
type-checking!

Expected
e
and got
List e

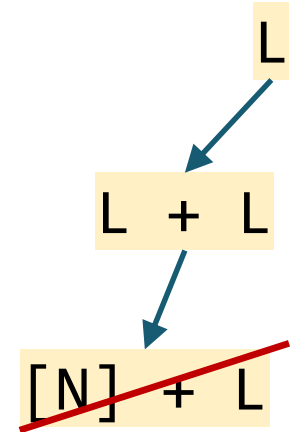
Motivation

Goal: use deductive reasoning for top-down propagation



- prune unverifiable candidates early
- need synthesis-friendly verification technique!

Observation: type checkers are good at rejecting incomplete programs!

Idea: can we use types as behavioral constraints for synthesis?



Conventional types are not enough

```
// Insert x into a sorted list xs
insert :: x:e → xs:List e → List e
insert x xs =
   match xs with
    Nil → Nil 
    Cons h t →
      if x ≤ h
      then Cons x xs
      else Cons h (insert x t)
```

Refinement types

[Rondon et al.'08, Kawaguchi et al.'09]

Nat

base types

$\text{max} :: x: \text{Int} \rightarrow y: \text{Int} \rightarrow \{ v: \text{Int} \mid x \leq v \wedge y \leq v \}$

dependent
function types

$\text{xs} :: \{ v: \text{List Nat} \}$

polymorphic
datatypes

data List α **where**

Nil :: { List α | $\text{Len } v = 0$ }

Cons :: $x: \alpha \rightarrow \{ \text{List } \alpha \mid \text{Len } v = \text{Len } xs + 1 \}$

measure Len :: List $\alpha \rightarrow \text{Int}$

$\text{Len Nil} = 0$

$\text{Len (Cons } _ \text{ xs)} = \text{Len } xs + 1$

Refinement types

$e ::= \text{true} \mid \text{false} \mid n \mid e + e$
 $\mid x \mid e \ e \mid \lambda x:T. e$

Terms

$T ::= \{v: B \mid e\}$ (basic types)
 $\mid x: T_1 \rightarrow T_2$ (function types)
 $\mid \alpha$ (type variables)

Types

$S ::= T \mid \forall \alpha. S$

Type schemas

T-num
$$\frac{(n = 0, 1, \dots)}{\Gamma \vdash n :: \{v: \text{Int} \mid v = n\}}$$

T-var
$$\frac{(x: T \in \Gamma)}{\Gamma \vdash x :: \{v: T \mid v = x\}}$$

T-app
$$\frac{\Gamma \vdash e_1 :: x: T \rightarrow T' \quad \Gamma \vdash e_2 :: T}{\Gamma \vdash e_1 \ e_2 :: T'[x \mapsto e_2]}$$

Example

Let's check that $\Gamma \vdash \text{double } 5 :: \text{Nat}$

- $\text{Nat} = \{v: \text{Int} \mid v \geq 0\}$
- $\Gamma = [\text{double}: x: \text{Int} \rightarrow \{v: \text{Int} \mid v = 2 * x\}]$

$$\text{T-num} \quad \frac{(n = 0, 1, \dots)}{\Gamma \vdash n :: \{v: \text{Int} \mid v = n\}}$$

$$\text{T-var} \quad \frac{(x: T \in \Gamma)}{\Gamma \vdash x :: \{v: T \mid v = x\}}$$

$$\text{T-abs} \quad \frac{\Gamma; x: T \vdash e :: T'}{\Gamma \vdash \lambda x: T. e :: T \rightarrow T'}$$

$$\text{T-app} \quad \frac{\Gamma \vdash e_1 :: x: T \rightarrow T' \quad \Gamma \vdash e_2 :: T}{\Gamma \vdash e_1 \ e_2 :: T' [x \mapsto e_2]}$$

We need subtyping!

Subtyping

Intuitively, T' is a subtype of T if all values of type T' also belong to T

- written $T' <: T$
- e.g. $\text{Nat} <: \text{Int}$ or $\{v: \text{Int} \mid v = 5\} <: \text{Nat}$

Defined via inference rules:

$$\text{Sub-base} \frac{[\![\Gamma]\!] \wedge e' \Rightarrow e}{\Gamma \vdash \{v: B \mid e'\} <: \{v: B \mid e\}}$$

$$\text{Sub-fun} \frac{\Gamma \vdash T_1 <: T'_1 \quad \Gamma; x: T_1 \vdash T'_2 <: T_2}{\Gamma \vdash x: T'_1 \rightarrow T'_2 <: x: T_1 \rightarrow T_2}$$

Conventional types are not enough

```
// Insert x into a sorted list xs
insert :: x:e → xs:List e → List e
insert x xs =
  ✓ match xs with
    Nil → Nil ←
    Cons h t →
      if x ≤ h
      then Cons x xs
      else Cons h (insert x t)
```

Refinement types

data *SList* *e* **where** sorted lists

Nil :: *SList* *e*


Cons :: *h*:*e* →

t:*SList* $\{v:e \mid v \geq h\}$ →
SList *e*



Refinement types as specs

[Rondon et al. PLDI'08]

```
// Insert x into a sorted list xs
insert :: x:e → xs:SList e →
        {v:SList e | 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

$\{v:SList\ e \mid elems\ v = elems\ xs \cup \{x\}\}$

and got

$\{v:SList\ e \mid elems\ xs \subseteq elems\ v\}$

Incomplete programs?

```
// Insert x into a sorted list xs
insert :: x:e → xs:SList e →
        {v:SList e | elems v = elems xs ∪ {x}}
insert x xs =
  ? match xs with
    Nil → Nil
    Cons h t → ...
```

Bidirectional type checking

$\{v:\text{SList } e \mid \text{elems } v = \{x\}\}$



insert x xs =
 match xs with
 Nil → Nil
 Cons h t → ...



Round-trip type checking

`insert :: x:α → xs:SList α → SList α`

`{v:e | v ≥ h}`



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

`h` `(insert x ...)`

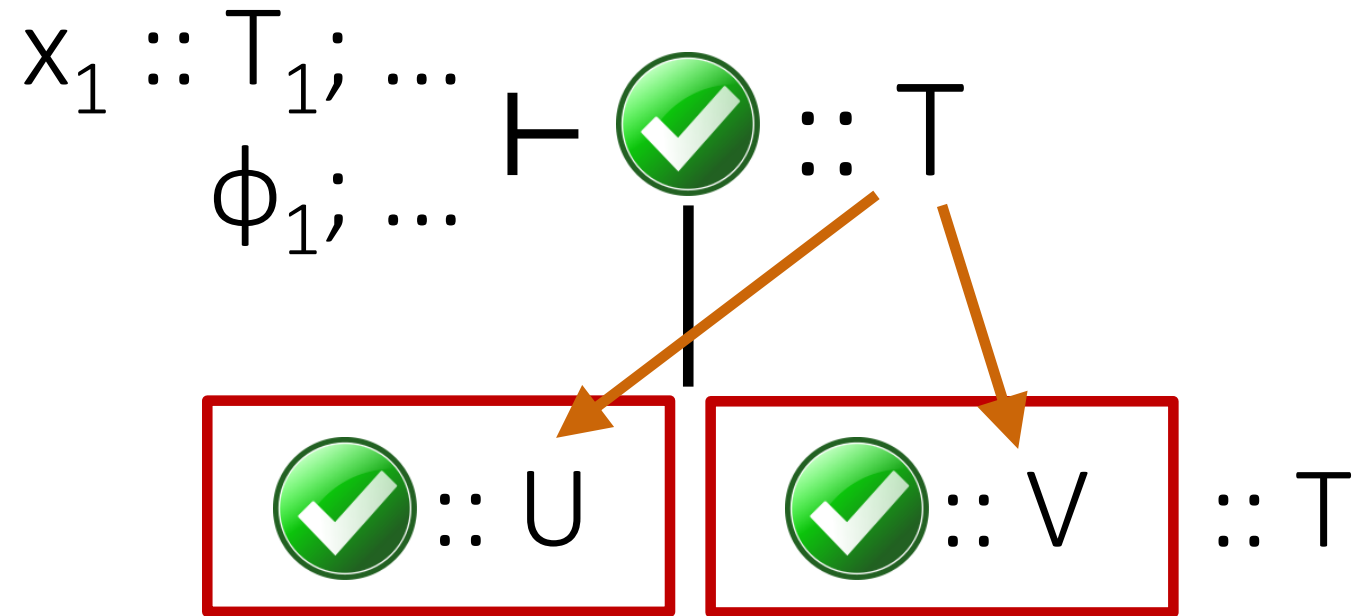


Type-driven Synthesis



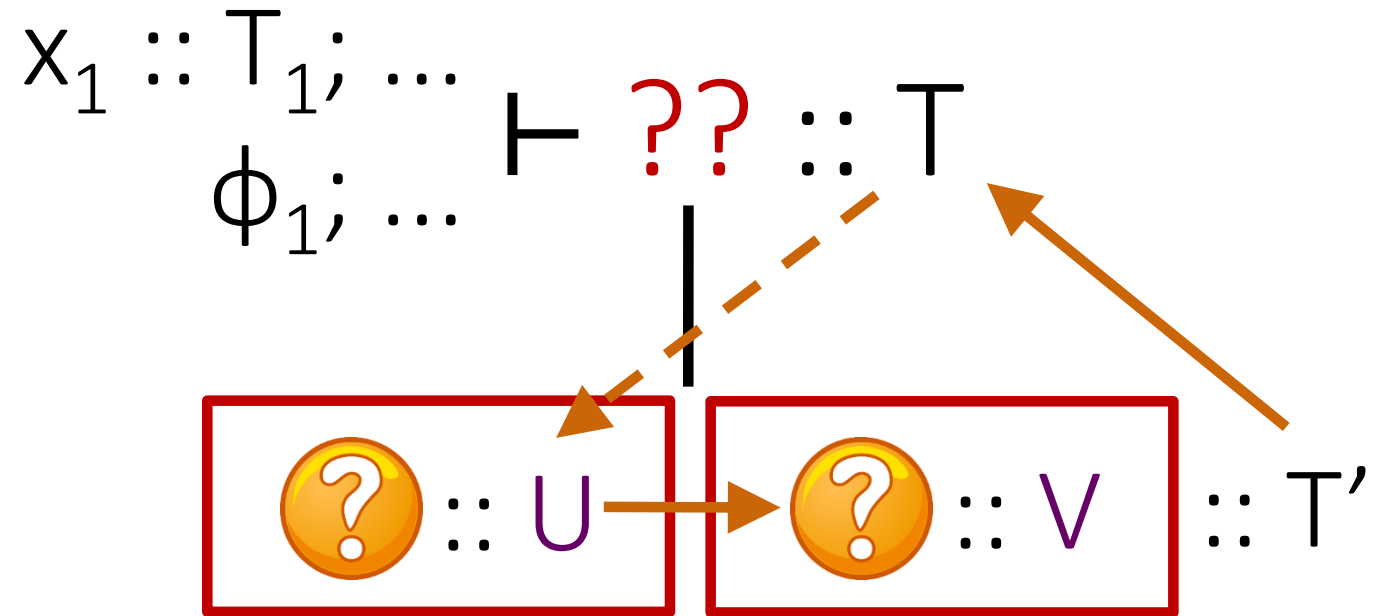
<http://tiny.cc/synquid>

Synthesis from refinement types



I. top-down enumerative search

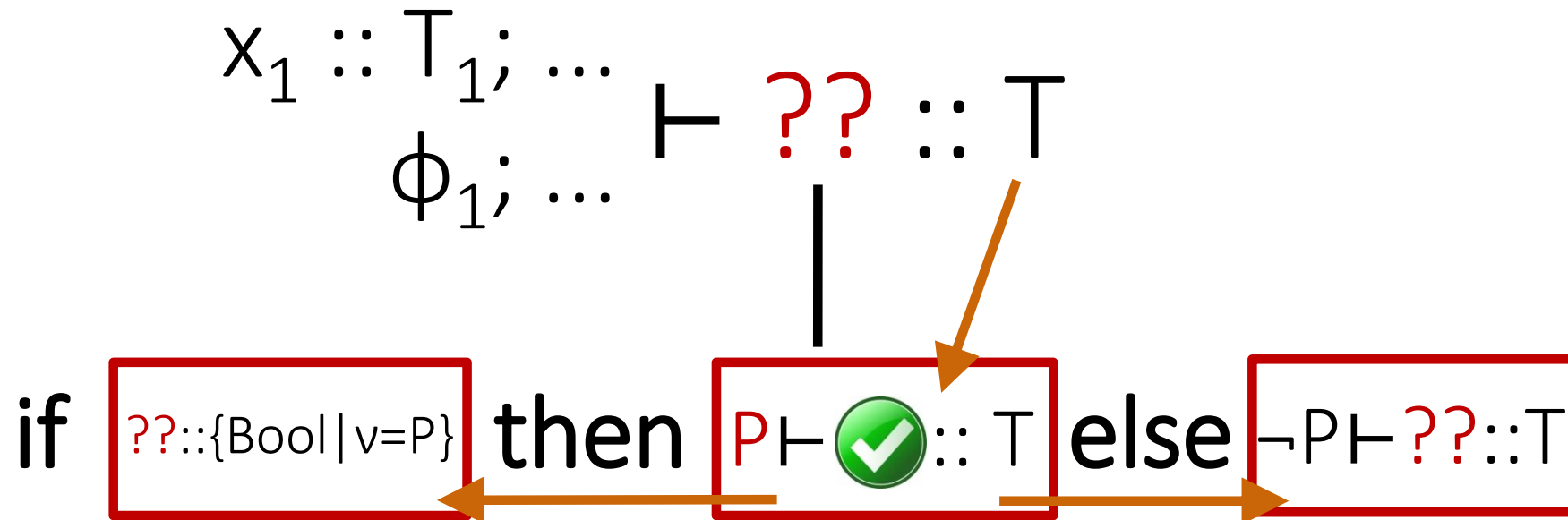
Synthesis from refinement types



I. top-down enumerative search

II. round-trip type checking

Synthesis from refinement types

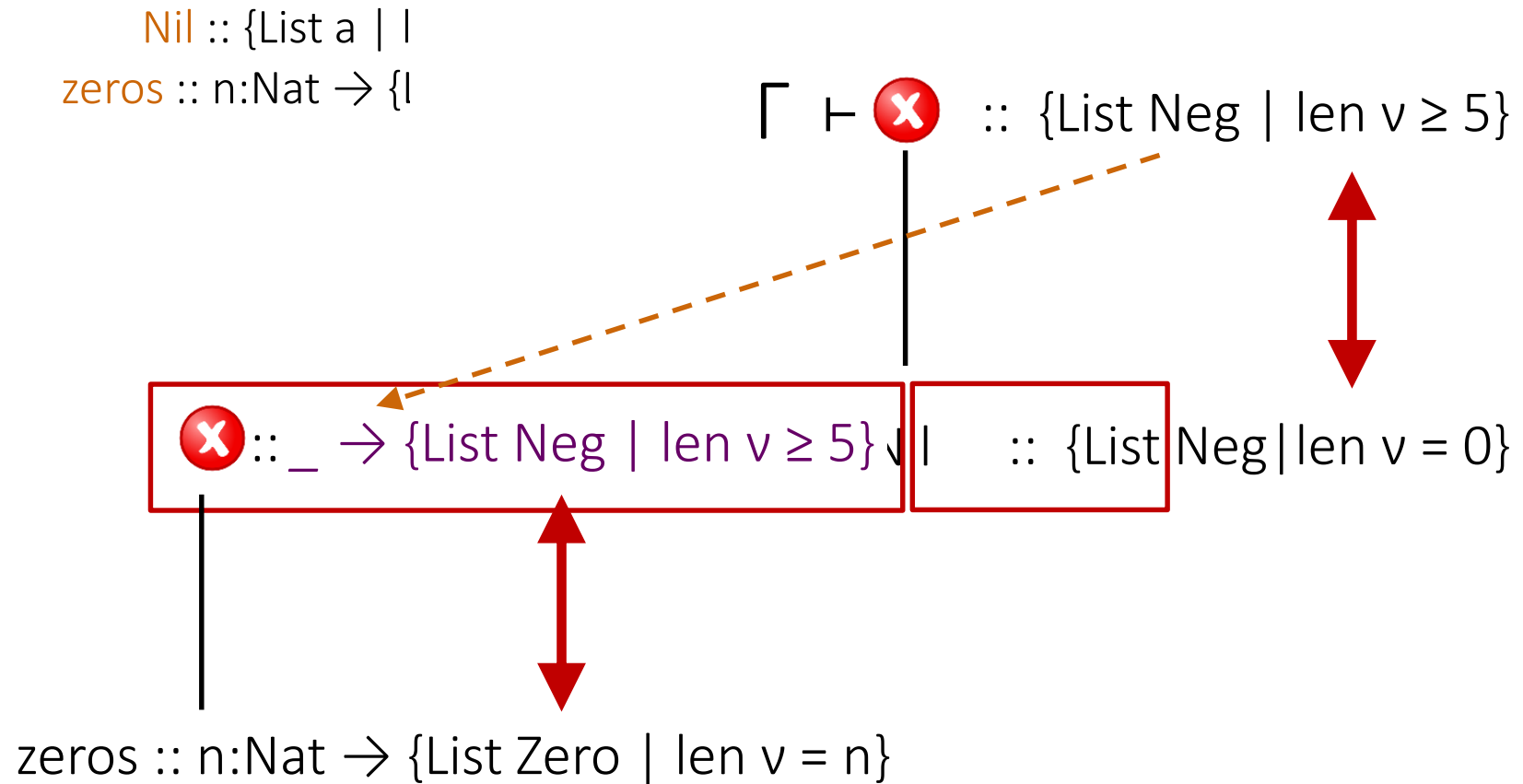


I. top-down enumerative search

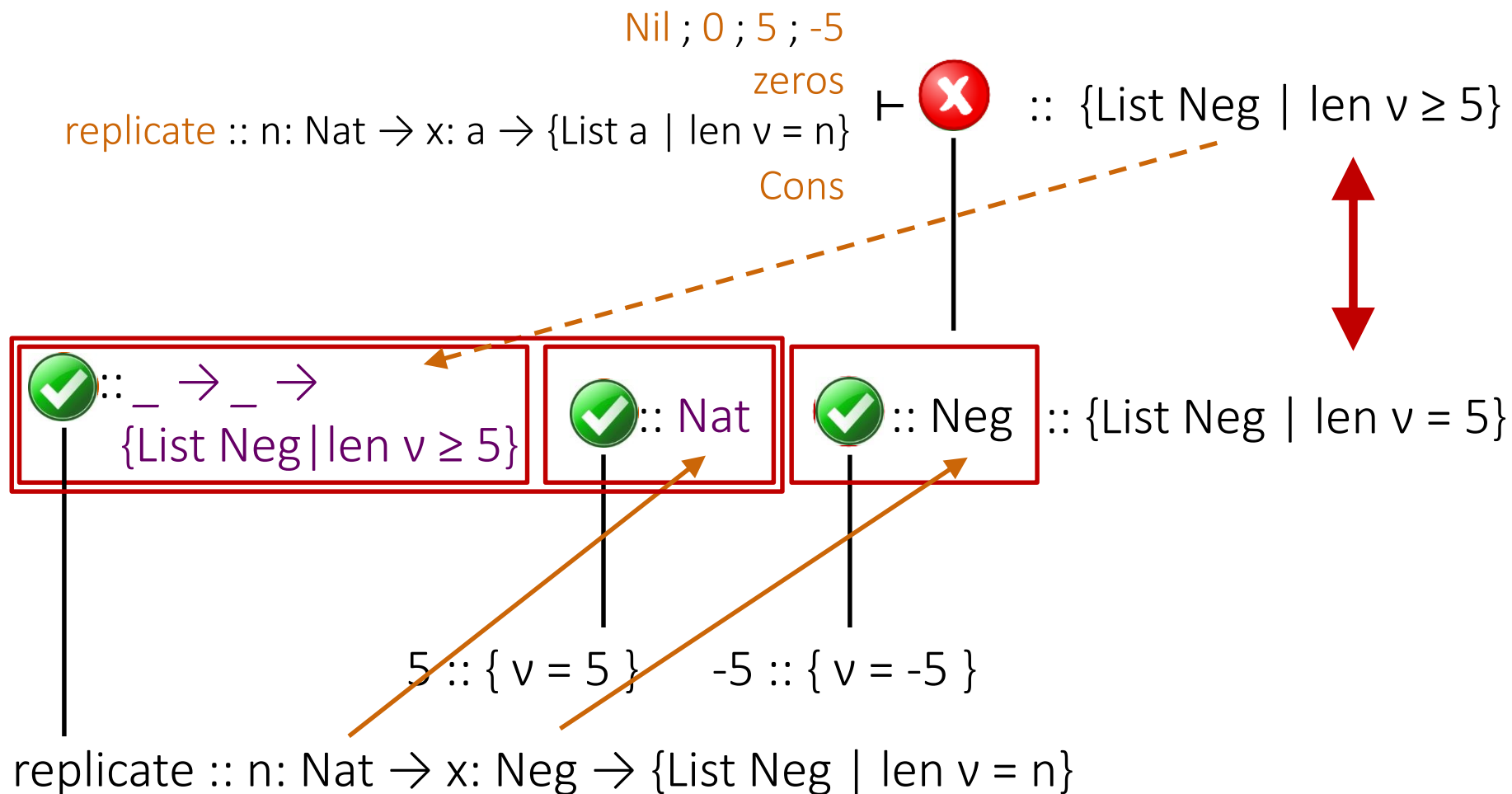
II. round-trip type checking

III. condition abduction

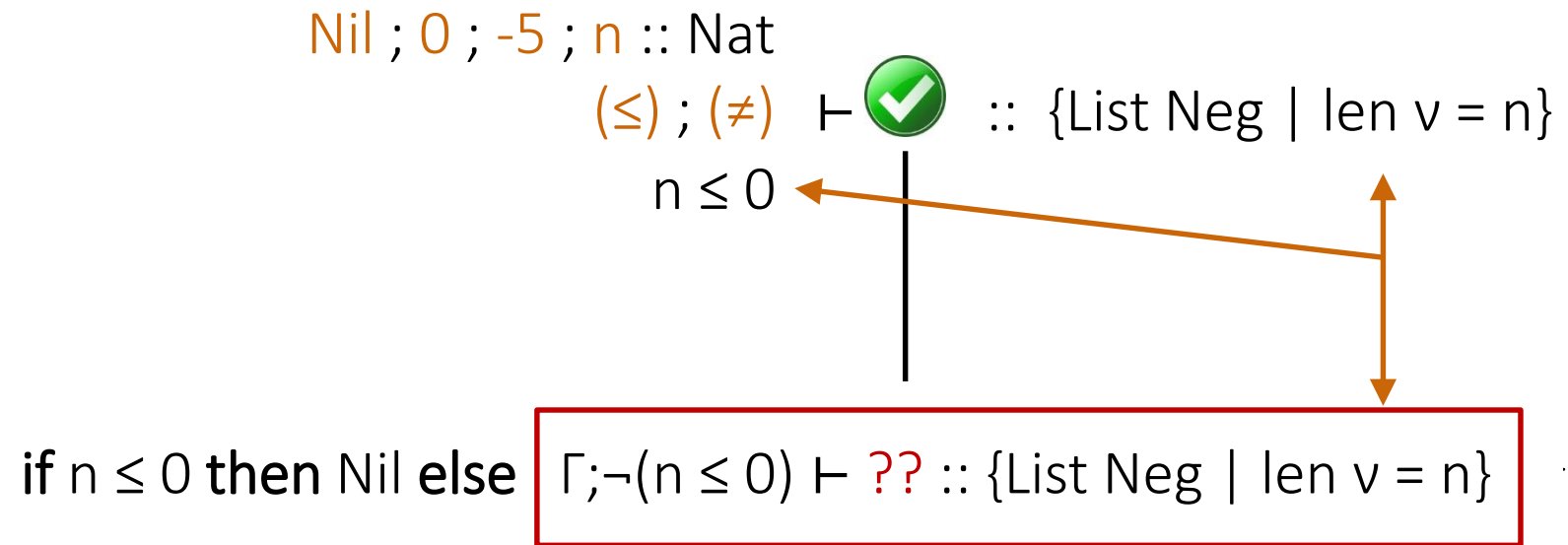
Example



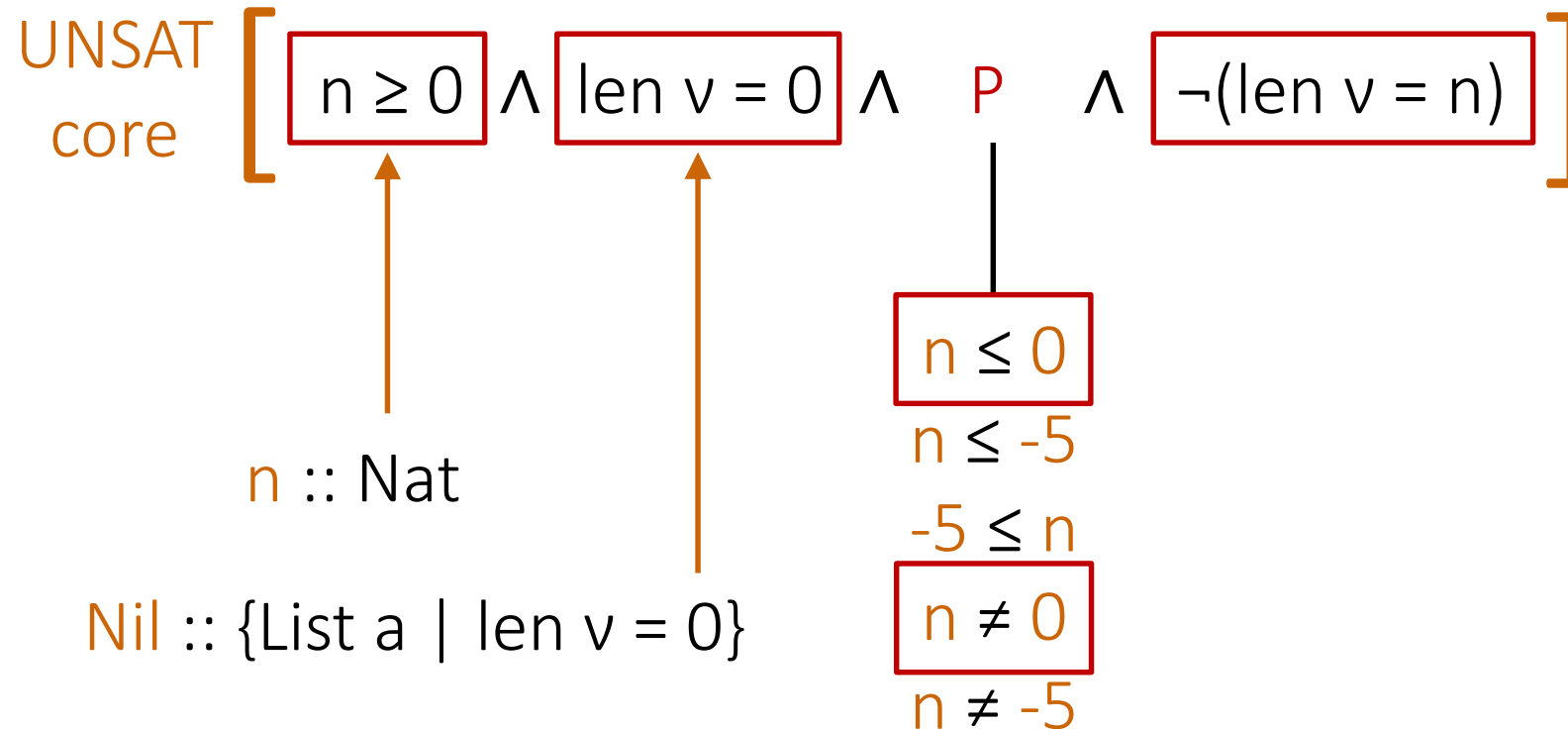
Example



Condition abduction



Liquid abduction



Synquid: contributions

Round-trip type system to reject incomplete programs

- + GFP Horn Solver
- Extensible beyond refinement types?

Refinement types can express complex properties in a simple way

- a bit more programmer-friendly than pre-/postconditions
- handles recursive, HO functions
- automatic verification for a larger class of programs due to polymorphism (e.g. sorted list insert)

Synquid: limitations

User interaction

- refinement types still less user-friendly than examples (?)
- 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

- $\{B_0 \mid \perp\} \rightarrow \{B_1 \mid \perp\} \rightarrow \{\text{List Pos} \mid \text{len } v = \textcolor{red}{2}5\}$

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 $(\text{len } v = 2 * (\text{len } xs) \wedge \text{len } v = 5 \rightarrow \text{UNSAT})$