

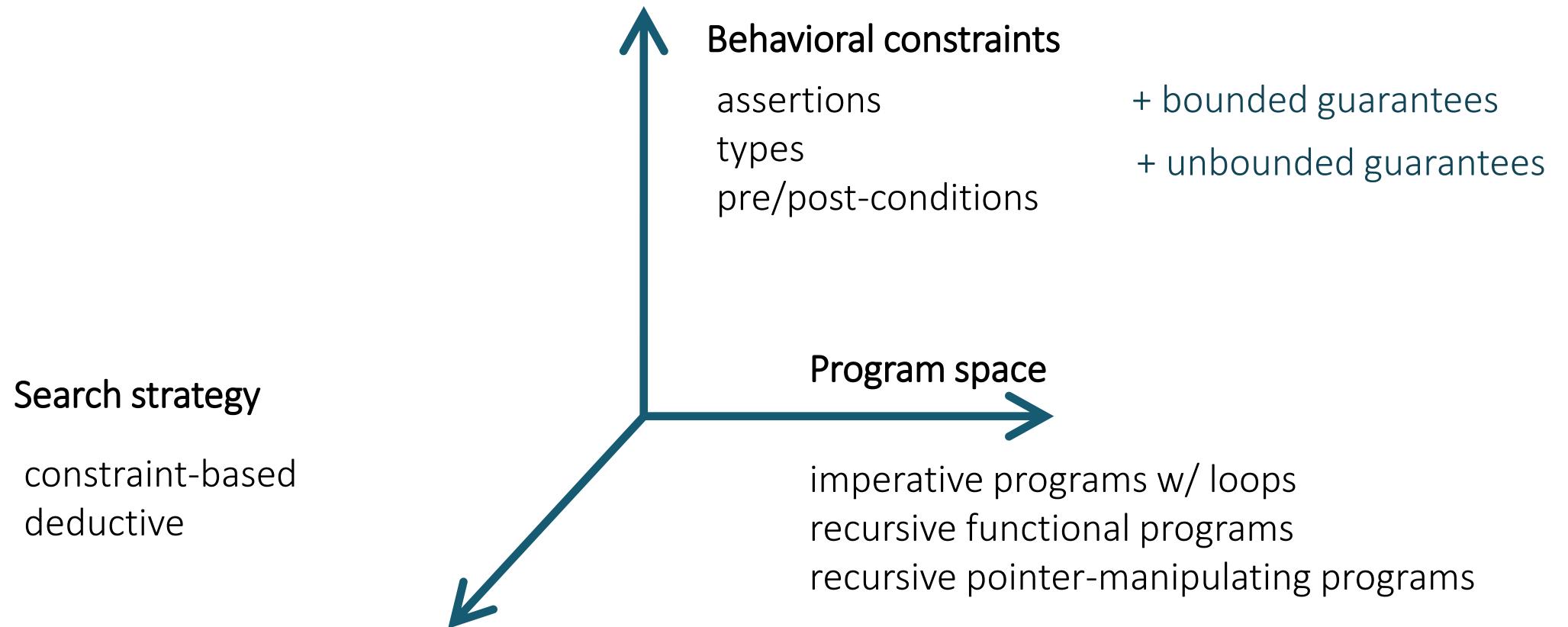
# Lecture 11

# Type-Driven Synthesis

*Nadia Polikarpova*

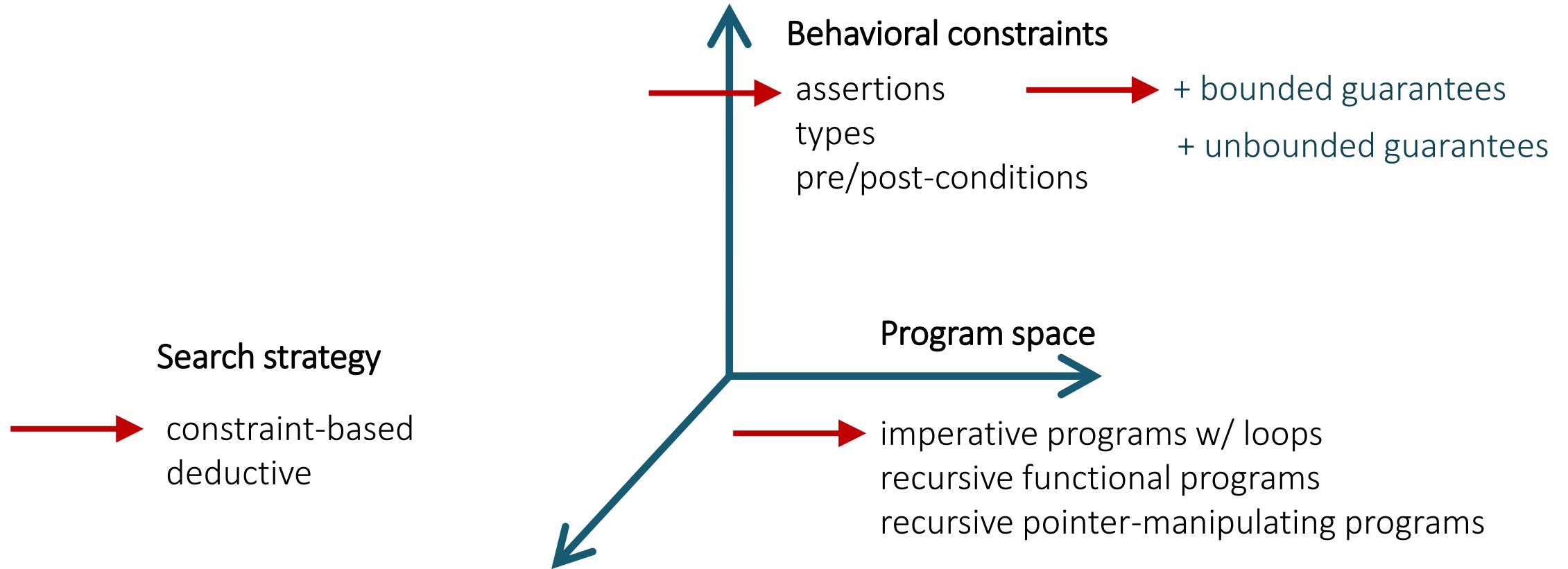
# Module II

---



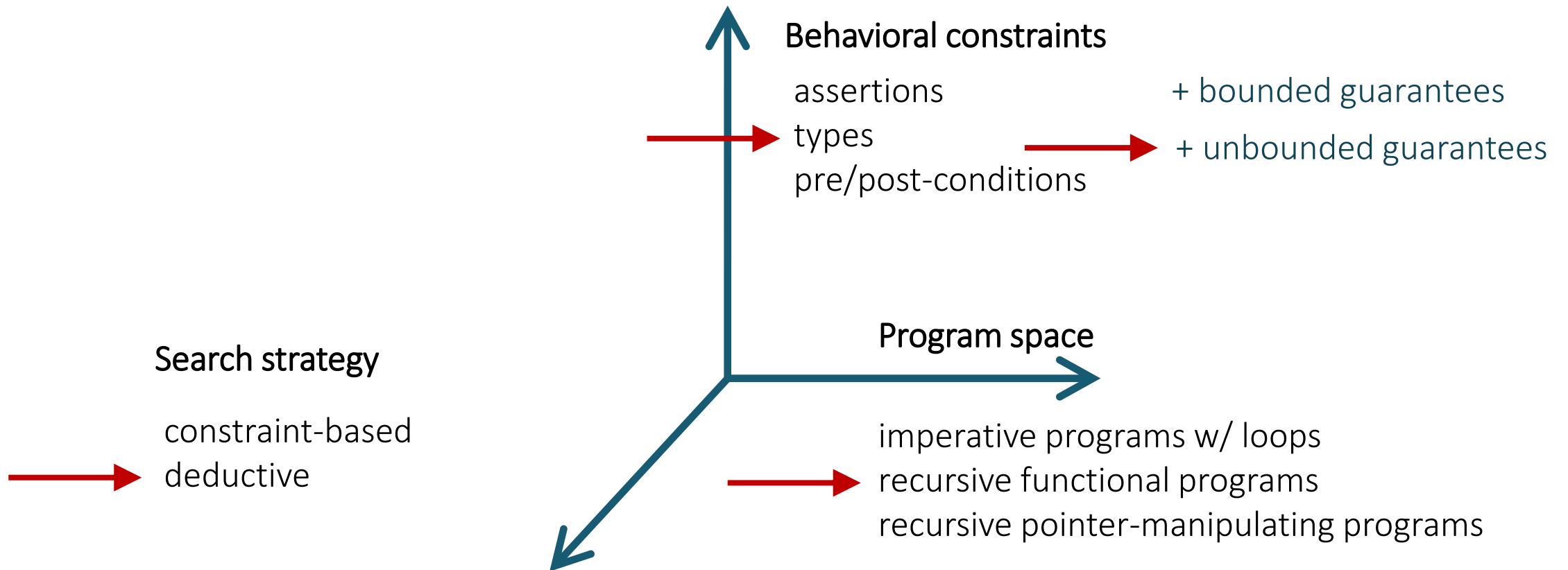
# Last week

---



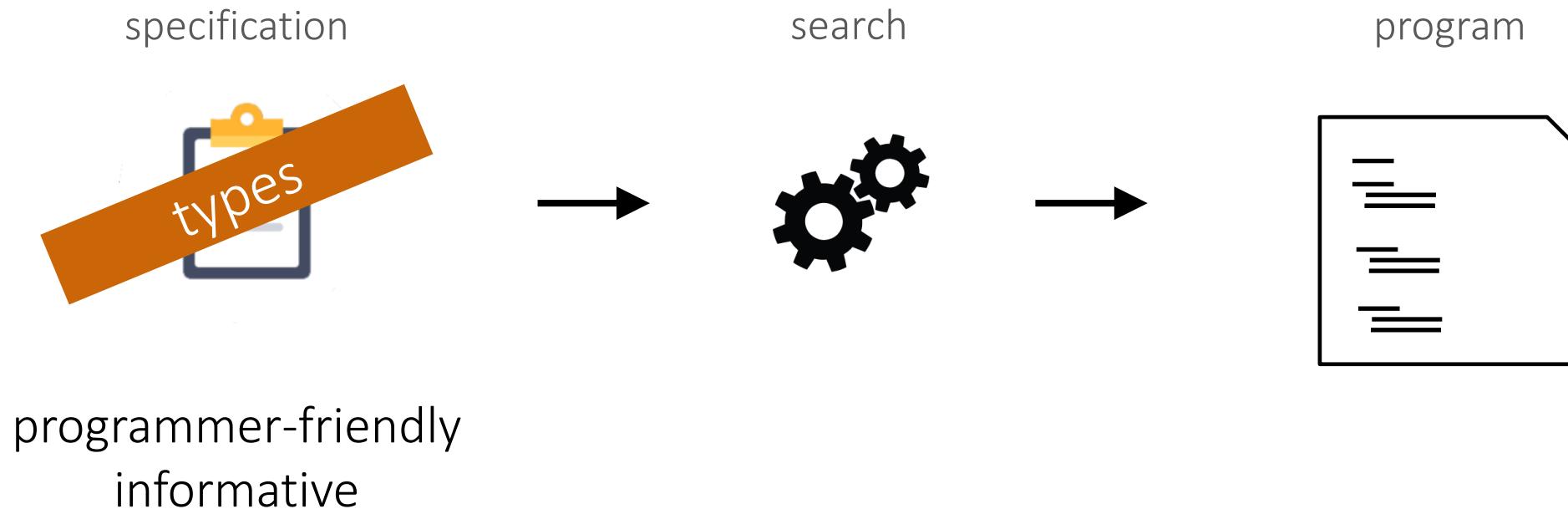
# This week

---



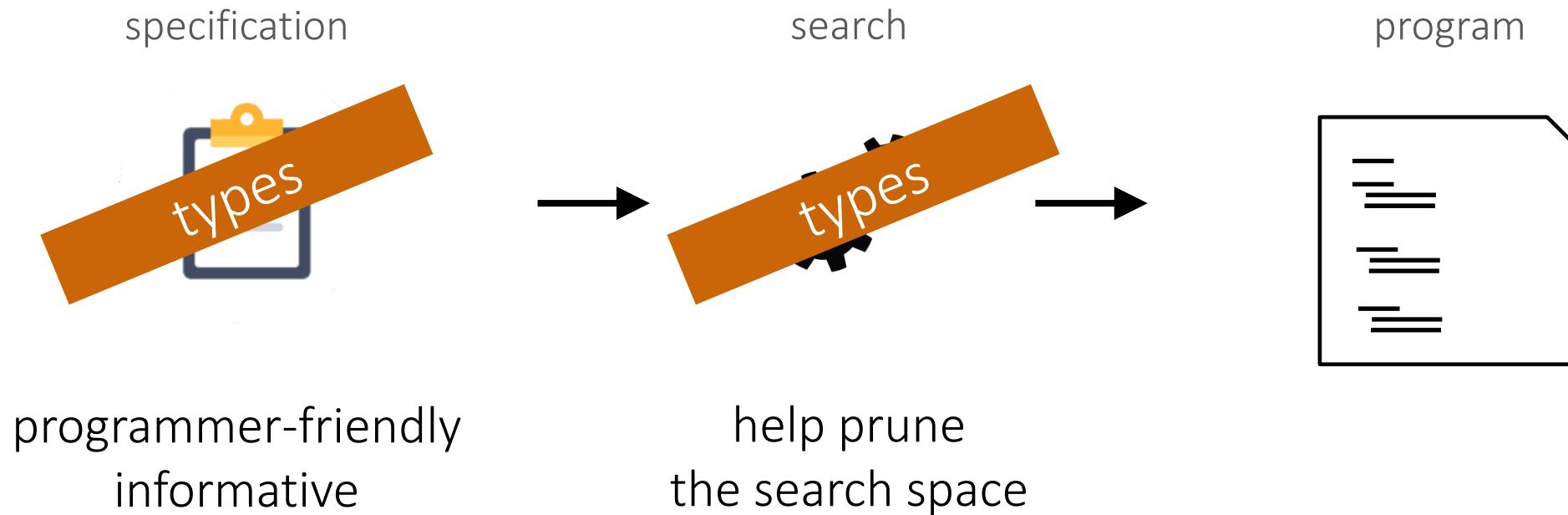
# Type-driven program synthesis

---



# Type-driven program synthesis

---



# Which program do I have in mind?

---

Char -> String -> [String]

split string at custom separator

a -> Int -> [a]

list with n copies of input value

# Which component to choose next?

---

specification

$$[a] \rightarrow \text{Int} \rightarrow [a]$$

concatenate n copies of input list

search state

$$f \ xs \ n = \text{concat} \ (\textcolor{red}{??} \ :: \ [[a]])$$

components

splitStr :: Char -> String -> [String]

replicate :: b -> Int -> [b]

inc :: Int -> Int

log2 :: Double -> Double

concat :: [[c]] -> [c]

# Which component to choose next?

---

specification

$[a] \rightarrow \text{Int} \rightarrow [a]$

concatenate n copies of input list

search state

$f \ xs \ n = \text{concat} \ (\text{replicate} \ xs \ n)$

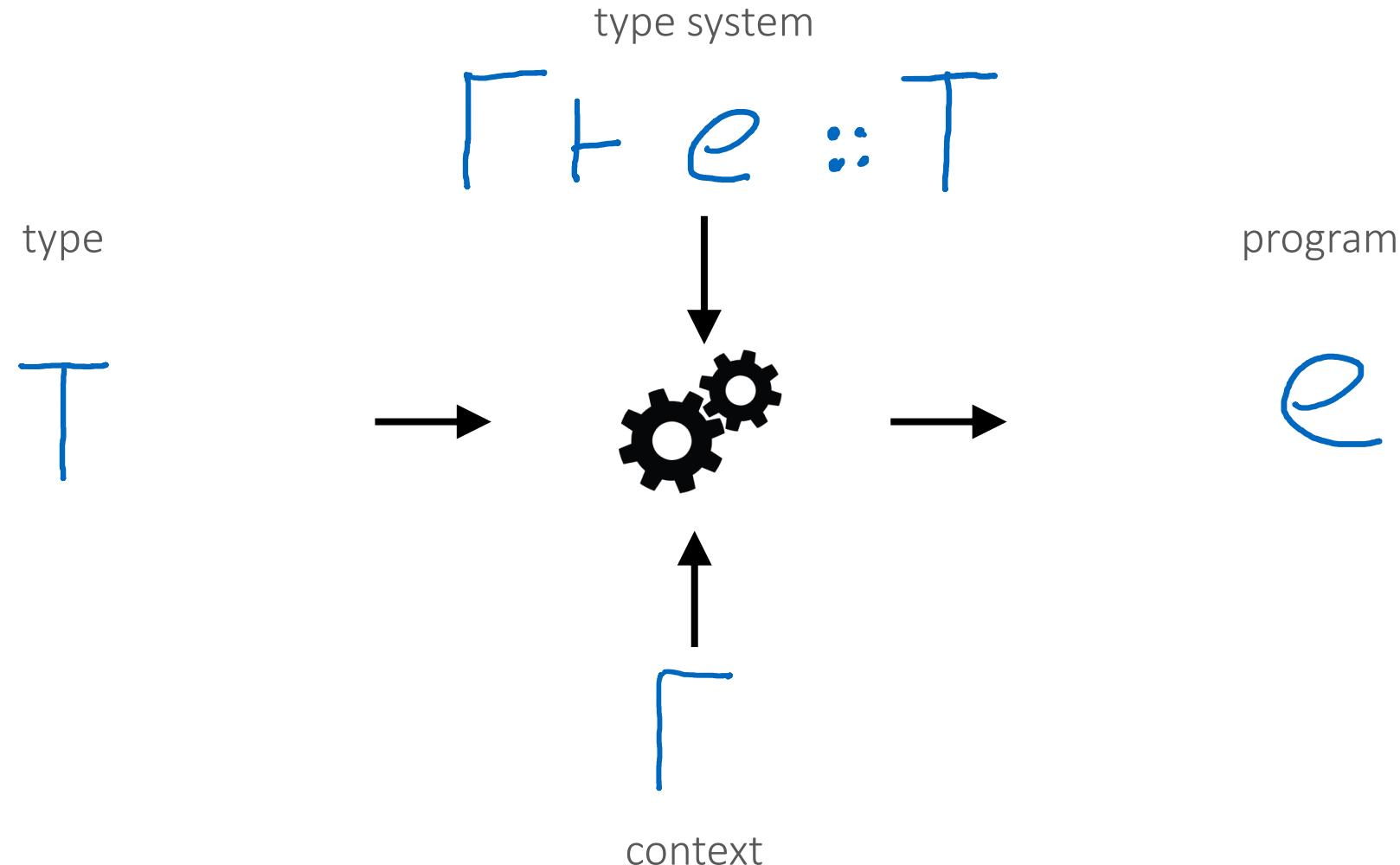


components

```
splitStr :: Char -> String -> [String]
replicate :: b -> Int -> [b]
inc :: Int -> Int
log2 :: Double -> Double
concat :: [[c]] -> [c]
```

# Type-driven program synthesis

---



# This week

---

intro to type systems

enumerating well-typed terms

synthesis with types and examples

polymorphic types

refinement types

synthesis with refinement types

# This week

---

intro to type systems

enumerating well-typed terms

synthesis with types and examples

polymorphic types

refinement types

synthesis with refinement types

# What is a type system?

---

Deductive system for proving facts about programs and types

Defined using *inference rules* over *judgments*

typing judgement

program / term  
↓  
context  $\rightarrow \Gamma \vdash e :: T \leftarrow$  type

“under context Gamma, term e has type T”

# A simple type system: syntax

---

$$e ::= 0 \mid e + \mid x \mid e \cdot e \mid \lambda x. e \quad \text{-- expressions}$$

example program: increment

$$\lambda x. x + 1$$

# A simple type system: syntax

---

$e ::= 0 \mid e + \mid x \mid e e \mid \lambda x. e$  -- expressions

$T ::= \text{Int} \mid T \rightarrow T$  -- types

$\Gamma ::= \cdot \mid x:T, \Gamma$  -- contexts

# Inference rules = typing rules

---

$$\frac{t\text{-zero}}{\Gamma \vdash 0 :: \text{Int}}$$
$$\frac{t\text{-succ} \quad \Gamma \vdash e :: \text{Int}}{\Gamma \vdash e + 1 :: \text{Int}}$$
$$\frac{t\text{-var} \quad x : T \in \Gamma}{\Gamma \vdash x :: T}$$

$$t\text{-abs} \quad \frac{\Gamma, x : T_1 \vdash e :: T_2}{\Gamma \vdash \lambda x. e :: T_1 \rightarrow T_2}$$

$$t\text{-app} \quad \frac{\Gamma \vdash e_1 :: T' \rightarrow T \quad \Gamma \vdash e_2 :: T'}{\Gamma \vdash e_1 e_2 :: T}$$

# Typing derivations

---

A derivation of  $\Gamma \vdash e :: T$  is a tree where

1. the root is  $\Gamma \vdash e :: T$
2. children are related to parents via inference rules
3. all leaves are axioms

# Typing derivations

---

let's build a derivation of

$$\cdot \vdash \lambda x. x + 1 :: \text{Int} \rightarrow \text{Int}$$

we say that  $\lambda x. x + 1$  is **well-typed** in the empty context  
and has type  $\text{Int} \rightarrow \text{Int}$

# Typing derivations

---

$$\begin{array}{c} \frac{}{\Gamma \vdash 0 :: \text{Int}} \quad \text{t-zero} \\ \frac{\Gamma \vdash e :: \text{Int}}{\Gamma \vdash e + 1 :: \text{Int}} \quad \text{t-plus} \\ \frac{t-var}{\Gamma \vdash x :: T} \quad \frac{}{\Gamma, x : T_1 \vdash e :: T_2} \quad \text{t-abs} \\ \frac{\Gamma \vdash e_1 :: T' \rightarrow T \quad \Gamma \vdash e_2 :: T'}{\Gamma \vdash e_1 e_2 :: T} \quad \text{t-app} \end{array}$$

$\cdot \vdash \lambda x. x + 1 :: \text{Int} \rightarrow \text{Int}$

# Typing derivations

---

is  $(\lambda x. x) + 1$  well-typed (in the empty context)?

no! no way to build a derivation of  $\cdot \vdash (\lambda x. x) + 1 :: _$

we say that  $(\lambda x. x) + 1$  is ill-typed

# Let's add lists!

---

$e ::= \dots | [] | e:e | \text{match } e \text{ with } [] \rightarrow e | x:x \rightarrow e$

$T ::= \text{Int} | \text{List} | T \rightarrow T$

# Example program: head with default

---

$$\lambda x. \text{match } x \text{ with } nil \rightarrow 0 \mid y: ys \rightarrow y$$

# Typing rules

$$\frac{t\text{-nil}}{\Gamma \vdash [] :: \text{List}}$$

$$\frac{t\text{-cons} \quad \Gamma \vdash e_1 :: \text{Int} \quad \Gamma \vdash e_2 :: \text{List}}{\Gamma \vdash e_1 : e_2 :: \text{List}}$$

what should the t-match rule be?

$$\frac{t\text{-match} \quad \begin{array}{c} \Gamma \vdash e_0 :: \boxed{1} \\ \Gamma \vdash e_1 :: \boxed{2} \quad \Gamma \vdash \boxed{4} \quad + e_2 :: \boxed{3} \end{array}}{\Gamma \vdash \text{match } e_0 \text{ with } [] \rightarrow e_1 \mid x:xs \rightarrow e_2 :: T}$$

# Typing rules

---

$$\frac{t\text{-nil}}{\Gamma \vdash [] :: \text{List}}$$

$$\frac{t\text{-cons} \quad \Gamma \vdash e_1 :: \text{Int} \quad \Gamma \vdash e_2 :: \text{List}}{\Gamma \vdash e_1 : e_2 :: \text{List}}$$

$$\frac{t\text{-match} \quad \Gamma \vdash e_0 :: \text{List} \quad \Gamma \vdash e_1 :: T \quad \Gamma, x:\text{Int}, xs:\text{List} \vdash e_2 :: T}{\Gamma \vdash \text{match } e_0 \text{ with } [] \rightarrow e_1 \mid x:xs \rightarrow e_2 :: T}$$

# Example: head with default

---

- $\vdash \lambda x. \text{match } x \text{ with } nil \rightarrow 0 \mid y: ys \rightarrow y :: \text{List} \rightarrow \text{Int}$

# Let's add recursion!

---

*instead of*  
 $\lambda x.e$

$$e ::= \dots \mid \text{fix } f(x) . e$$

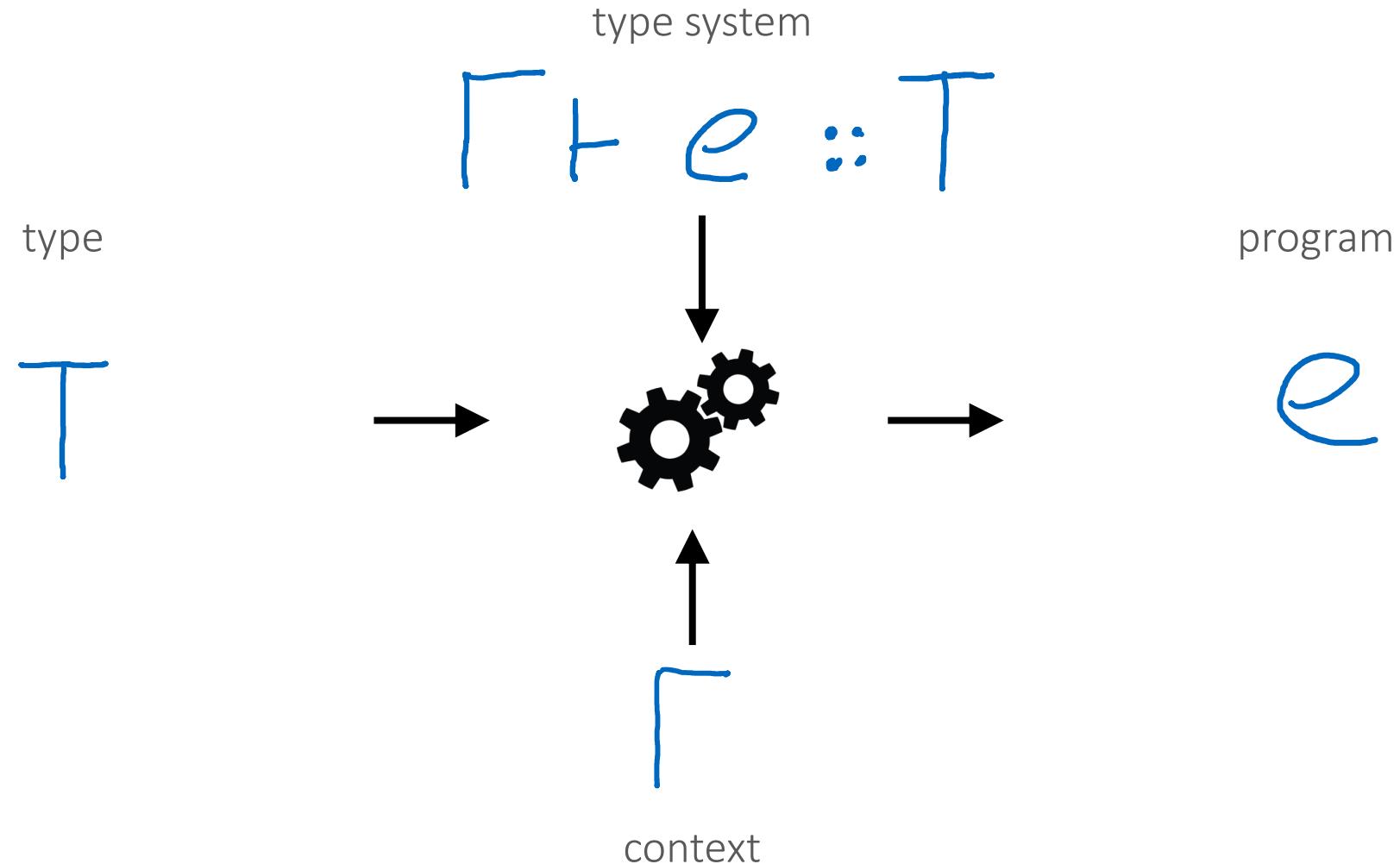
$$\frac{\text{t-fix} \quad \Gamma, x:T_1, f: T_1 \rightarrow T_2 \vdash e :: T_2}{\Gamma \vdash \text{fix } f(x) . e :: T_1 \rightarrow T_2}$$

# Example: length

---

$\cdot \vdash \text{fix } f(x) . \text{match } x \text{ with } nil \rightarrow 0 \mid y: ys \rightarrow f(ys) + 1$   
 $:: \text{List} \rightarrow \text{Int}$

# Type system → synthesis



# This week

---

intro to type systems

enumerating well-typed terms

synthesis with types and examples

polymorphic types

refinement types

synthesis with refinement types

# Enumerating well-typed terms

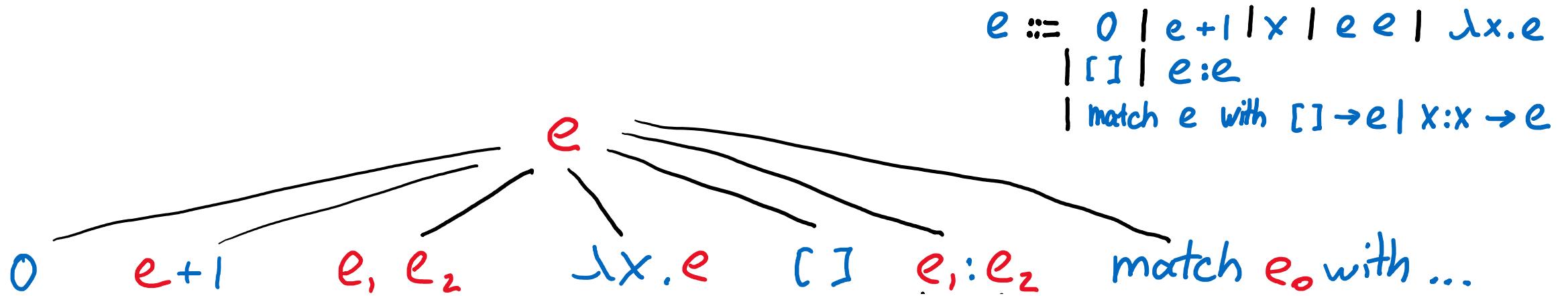
---

how should I enumerate all terms of type  $\text{List} \rightarrow \text{List}$ ?  
(up to depth 2, in the empty context)

naïve idea: syntax-guided enumeration

1. enumerate all terms *generated by the grammar*
2. type-check each term and throw away ill-typed ones

# Syntax-guided enumeration



31 complete programs enumerated  
only 2 have the type  $\text{List} \rightarrow \text{List}$ !  
can we do better?

# Enumerating well-typed terms

---

how should I enumerate all terms of type  $\text{List} \rightarrow \text{List}$ ?  
(up to depth 2, in the empty context)

better idea: type-guided enumeration  
enumerate all derivations *generated by the type systems*  
extract terms from derivations (well-typed by construction)

# Three ways to look at typing judgments

---

$$\Gamma \vdash e :: T$$

term and type are known: type checking

$$\Gamma \vdash e :: \color{red}{T}$$

term known, type unknown: type inference

$$\Gamma \vdash \color{red}{e} :: T$$

type known, term unknown: program synthesis  
(also: type inhabitation)

# Synthesis as proof search

---

input: synthesis goal  $\Gamma \vdash ? :: T$

output: derivation of  $\Gamma \vdash e :: T$  for some  $e$

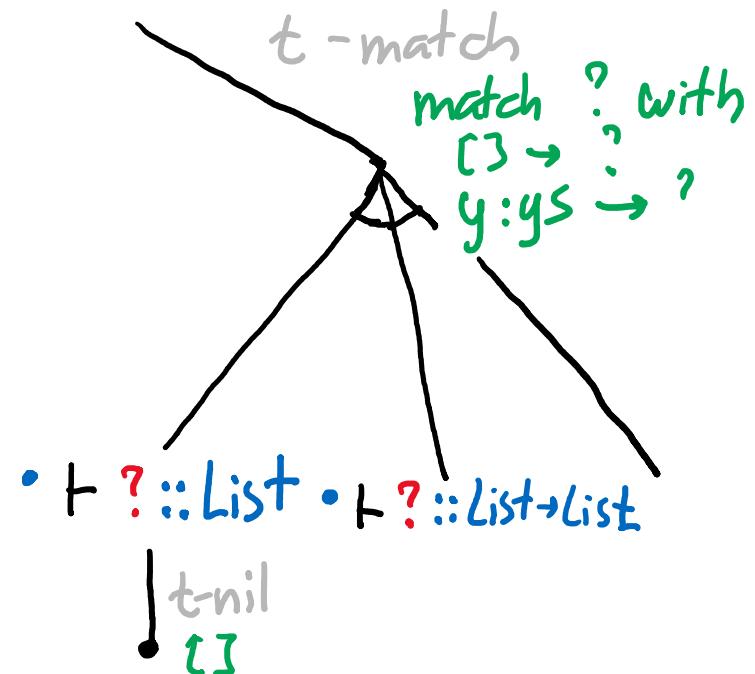
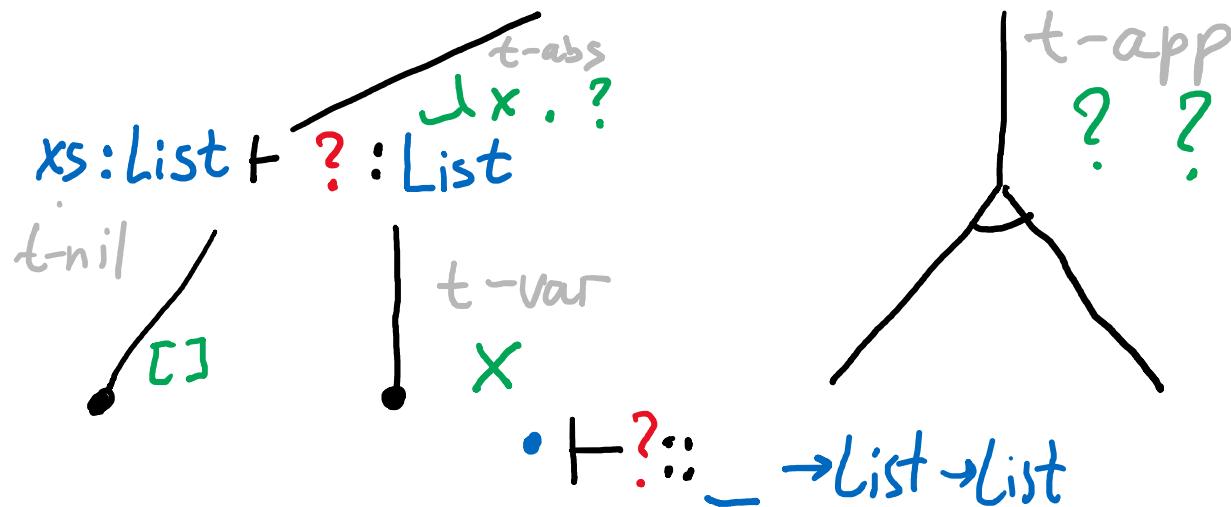
search strategy: recursively transform synthesis goal using typing rules until you find a complete derivation

# Type-guided enumeration

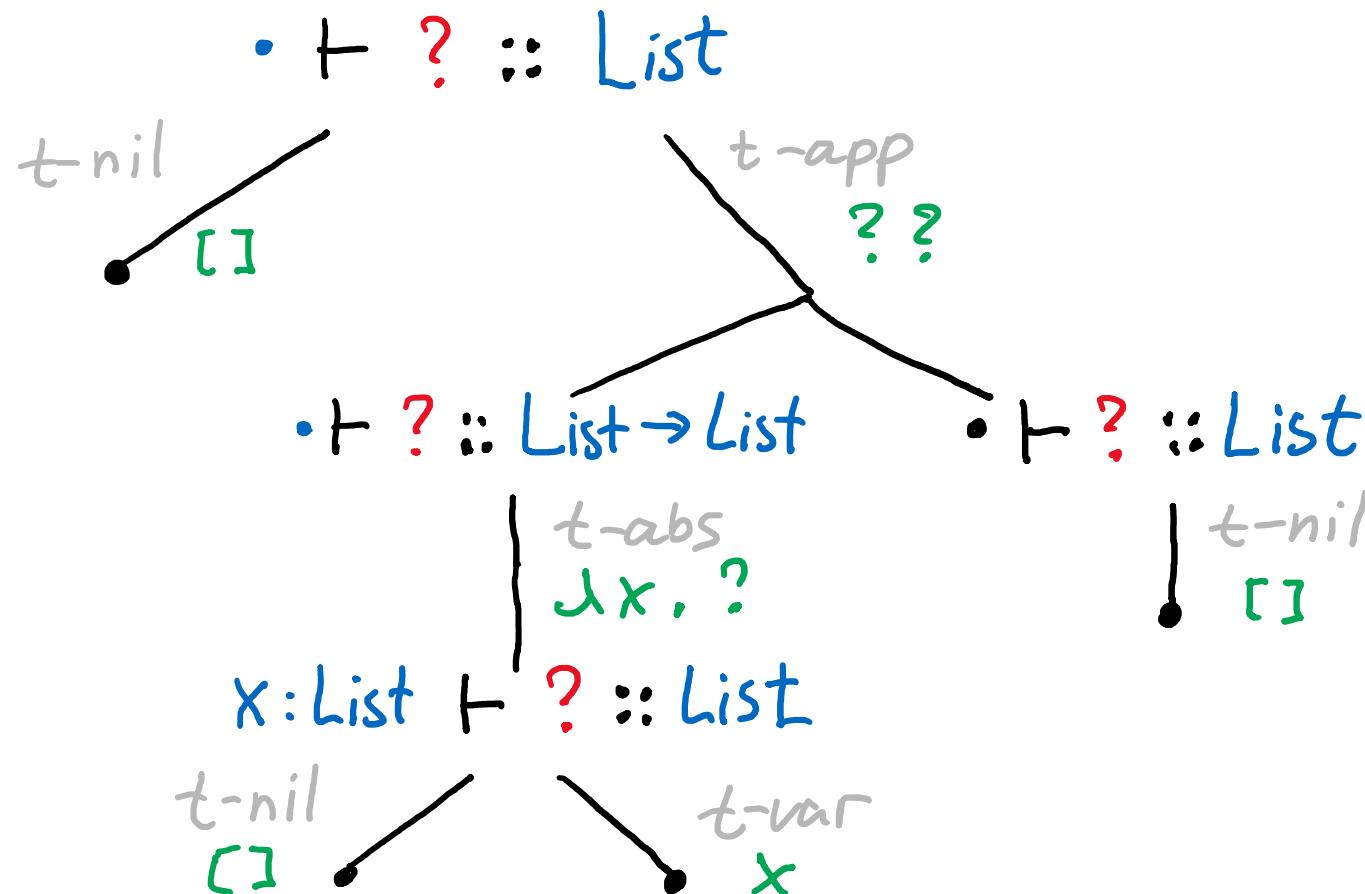
only 2 programs fully constructed!  
all other programs *rejected early*

$$\begin{array}{c}
 \frac{x:T \in \Gamma}{\Gamma \vdash x : T} \quad t-zero \quad \frac{\Gamma \vdash e : Int}{\Gamma \vdash 0 : Int} \quad t+ \frac{\Gamma \vdash e : Int}{\Gamma \vdash e + 1 : Int} \\
 \\[10pt]
 t-abs \quad \frac{\Gamma, x:T_1 \vdash e : T_2}{\Gamma \vdash \lambda x. e : T_1 \rightarrow T_2} \quad t-app \quad \frac{\Gamma \vdash e_1 : T' \rightarrow T \quad \Gamma \vdash e_2 : T'}{\Gamma \vdash e_1 e_2 : T} \\
 \\[10pt]
 t-nil \quad \frac{}{\Gamma \vdash [] : List} \quad t-cons \quad \frac{\Gamma \vdash e_1 : Int \quad \Gamma \vdash e_2 : List}{\Gamma \vdash e_1 :: e_2 : List} \\
 \\[10pt]
 t-match \quad \frac{\Gamma \vdash e_0 : List \quad \Gamma \vdash e_1 : T \quad \Gamma, x:Int, xs:List \vdash e_2 : T}{\Gamma \vdash \text{match } e_0 \text{ with } [] \rightarrow e_1 \mid x:xs \rightarrow e_2 : T}
 \end{array}$$

- $\vdash ? : List \rightarrow List$



# What's wrong with this search?



Enumerated 3 programs:

*nil*

$(\lambda x. x) \text{ nil}$

$(\lambda x. \text{ nil}) \text{ nil}$

They are all equivalent!

# Redundant programs

---

$$(\lambda x. e_1) e_2 \xrightarrow{\beta\text{-reduction}} e_1 [x := e_2]$$

Generating programs on the left  
is a waste of time!

$$\begin{array}{ccc} \text{match } [] \text{ with} & & \text{match-nil} \\ [] \rightarrow e_1 & \longrightarrow & e_1 \\ y:ys \rightarrow e_2 & & \end{array}$$

Idea: only generate programs *in  
normal form*

$$\begin{array}{ccc} \text{match } e_1 : e_2 \text{ with} & & \text{match-cons} \\ [] \rightarrow e_3 & \longrightarrow & e_4 [y := e_1, ys := e_2] \\ y:ys \rightarrow e_4 & & \end{array}$$

Restrict type system  
to make redundant programs *ill-typed*

# Normal-form programs

---

$$e ::= x \mid ei$$

elimination forms

$$i ::= 0 \mid i+1 \mid \lambda x. i \mid [] \mid i:i$$

| match e with [] → i | x:x → i

introduction forms

$$B ::= \text{Int} \mid \text{List}$$

base types

$$T ::= B \mid T \rightarrow T$$

types

# Bidirectional typing judgments

---

$$\Gamma \vdash i \Leftarrow T$$

“under context Gamma, i checks against type T”

$$\Gamma \vdash e \Rightarrow T$$

“under context Gamma, e generates type T”

[Pierce, Turner. Local Type Inference. 2000]

# Bidirectional typing rules

$$\text{e-var} \frac{x:T \in \Gamma}{\Gamma \vdash x : T}$$

$$\text{e-app} \frac{\Gamma \vdash e \Rightarrow T' \rightarrow T \quad \Gamma \vdash i \Leftarrow T'}{\Gamma \vdash e i \Rightarrow T}$$

$$\text{i-e} \frac{\Gamma \vdash e \Rightarrow B}{\Gamma \vdash e \Leftarrow B}$$

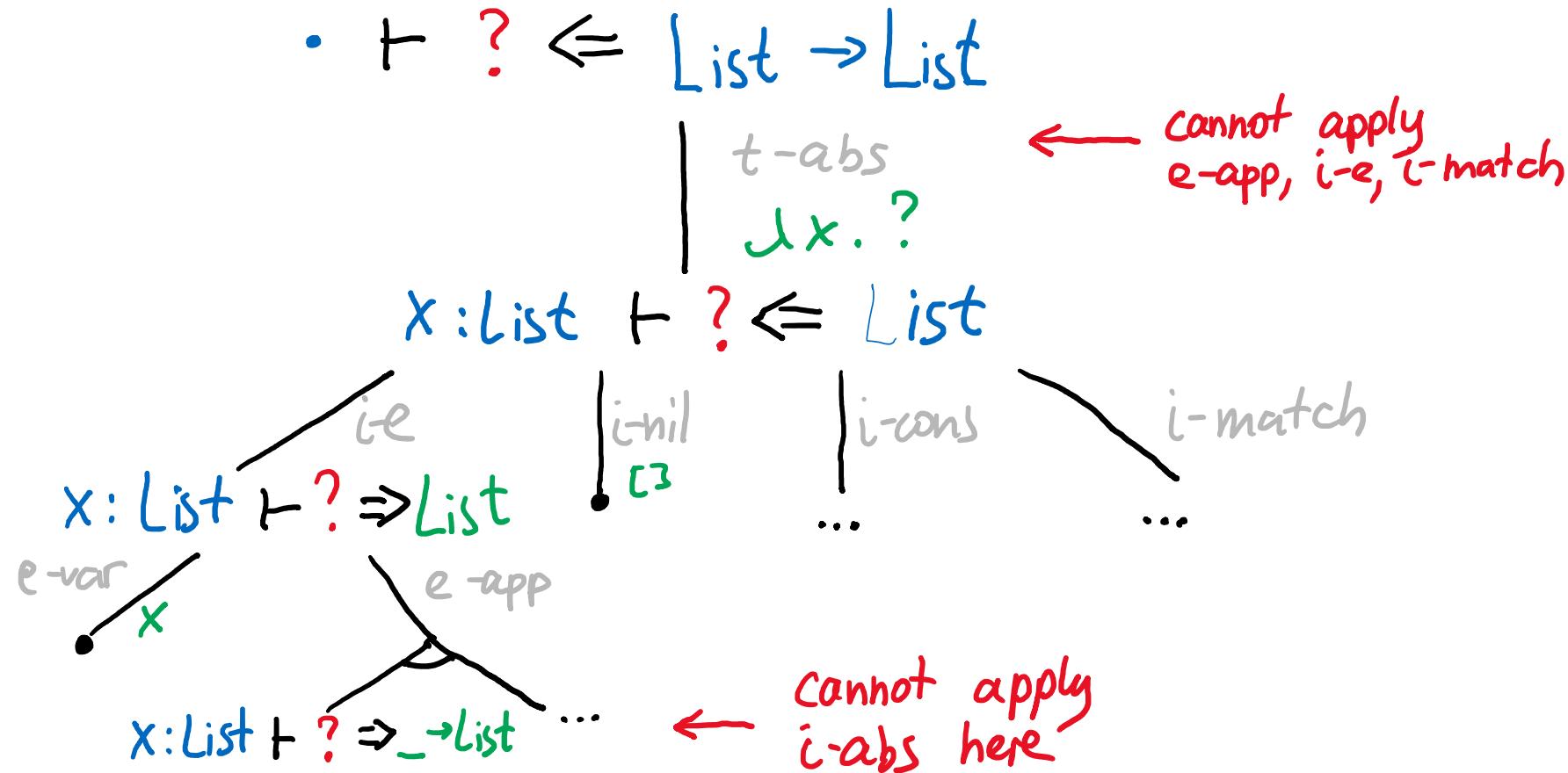
$$\text{i-nil} \frac{}{\Gamma \vdash [] \Leftarrow \text{List}}$$

$$\text{i-cons} \frac{\Gamma \vdash i_1 \Leftarrow \text{Int} \quad \Gamma \vdash i_2 \Leftarrow \text{List}}{\Gamma \vdash i_1 : i_2 \Leftarrow \text{List}}$$

$$\text{i-abs} \frac{\Gamma, x:T_1 \vdash i \Leftarrow T_2}{\Gamma \vdash \lambda x. i \Leftarrow T_1 \rightarrow T_2}$$

$$\text{i-match} \frac{\Gamma \vdash e \Rightarrow \text{List} \quad \Gamma \vdash i_1 \Leftarrow B \quad \Gamma, y:\text{Int}, ys:\text{List} \vdash i_2 \Leftarrow B}{\Gamma \vdash \text{match } e \text{ with } [] \Rightarrow i_1 \mid y:ys \Rightarrow i_2 \Leftarrow B}$$

# Type-guided enumeration



# This week

---

intro to type systems

enumerating well-typed terms

synthesis with types and examples

polymorphic types

refinement types

synthesis with refinement types

# Simple types are not enough

---

specification

“duplicate every  
element in a list”

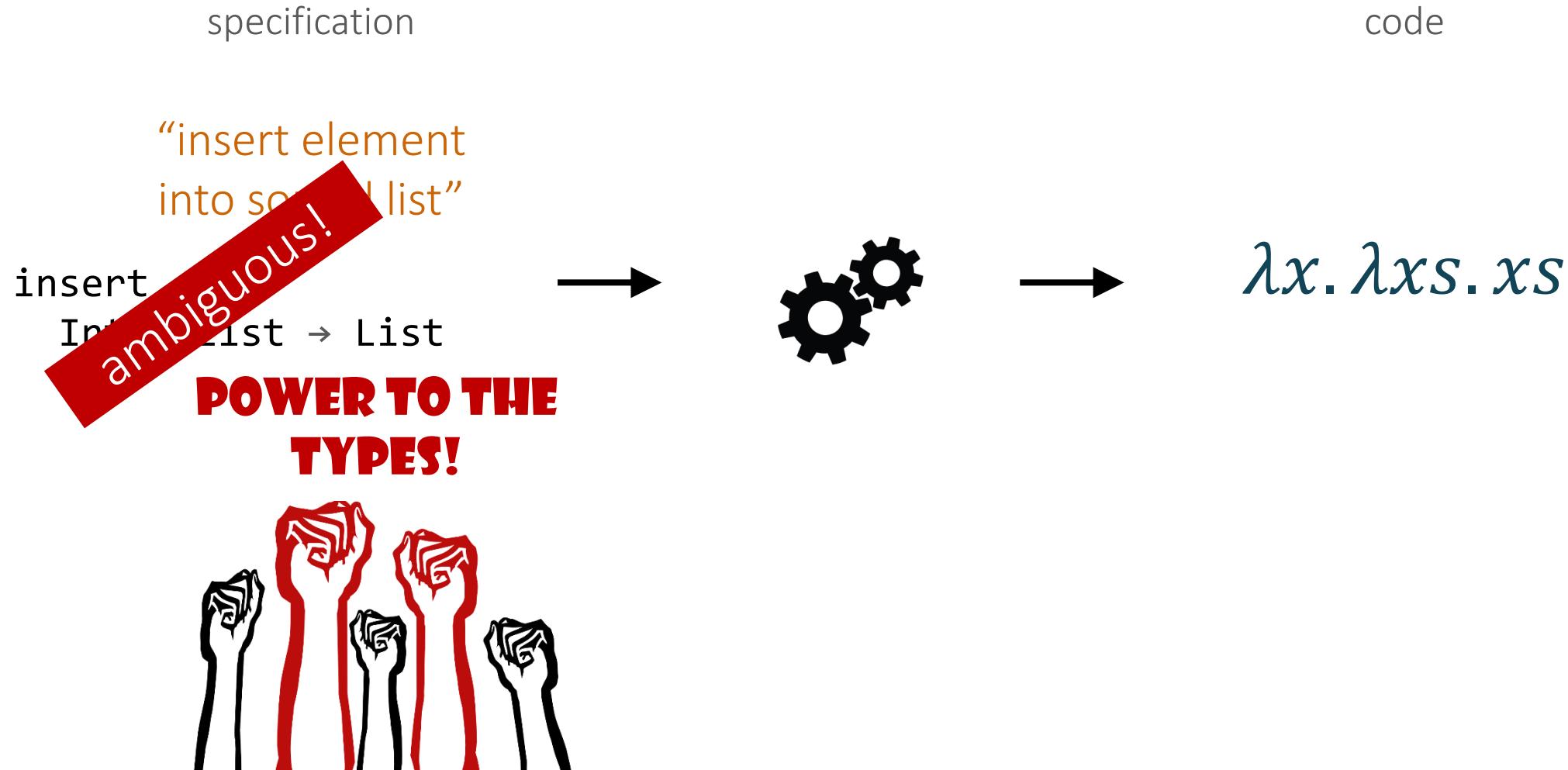
stutter :: List  $\rightarrow$  List  $\rightarrow$



code

$\lambda xs. xs$

# Simple types are not enough



# Type-driven synthesis in 3 easy steps

---

1. Annotate types with extra specs  
examples, logical predicates, resources, ...
2. Design a type system for annotated types  
propagate as much info as possible from conclusion to premises
3. Perform type-directed enumeration as before

# This week

---

intro to type systems

enumerating well-typed terms

**synthesis with types and examples**

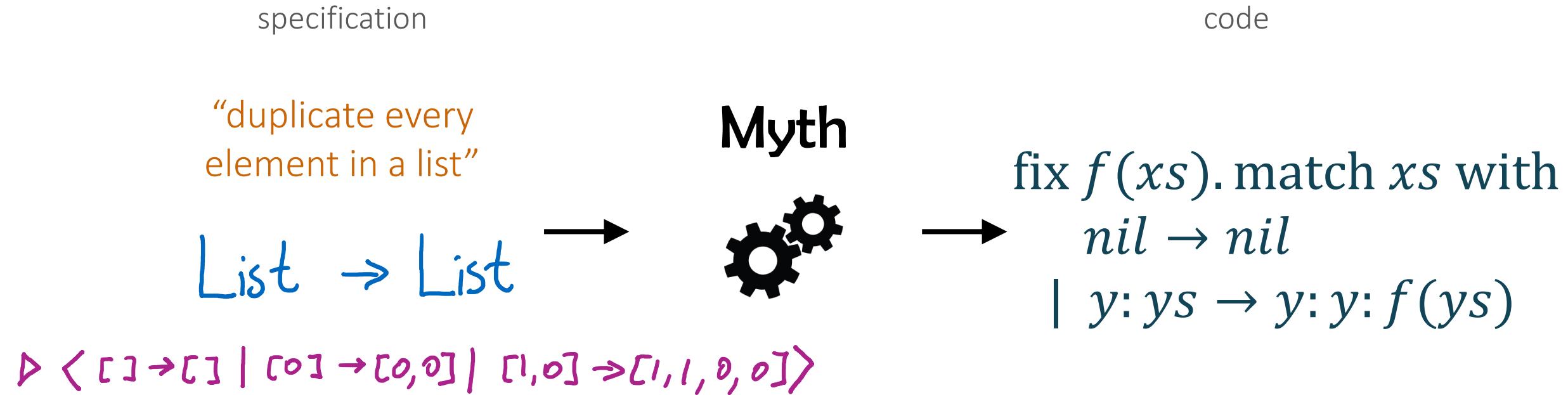
polymorphic types

refinement types

synthesis with refinement types

# Type + examples

---



[Osera, Zdancewic , Type-and-Example-Directed Program Synthesis. 2015]

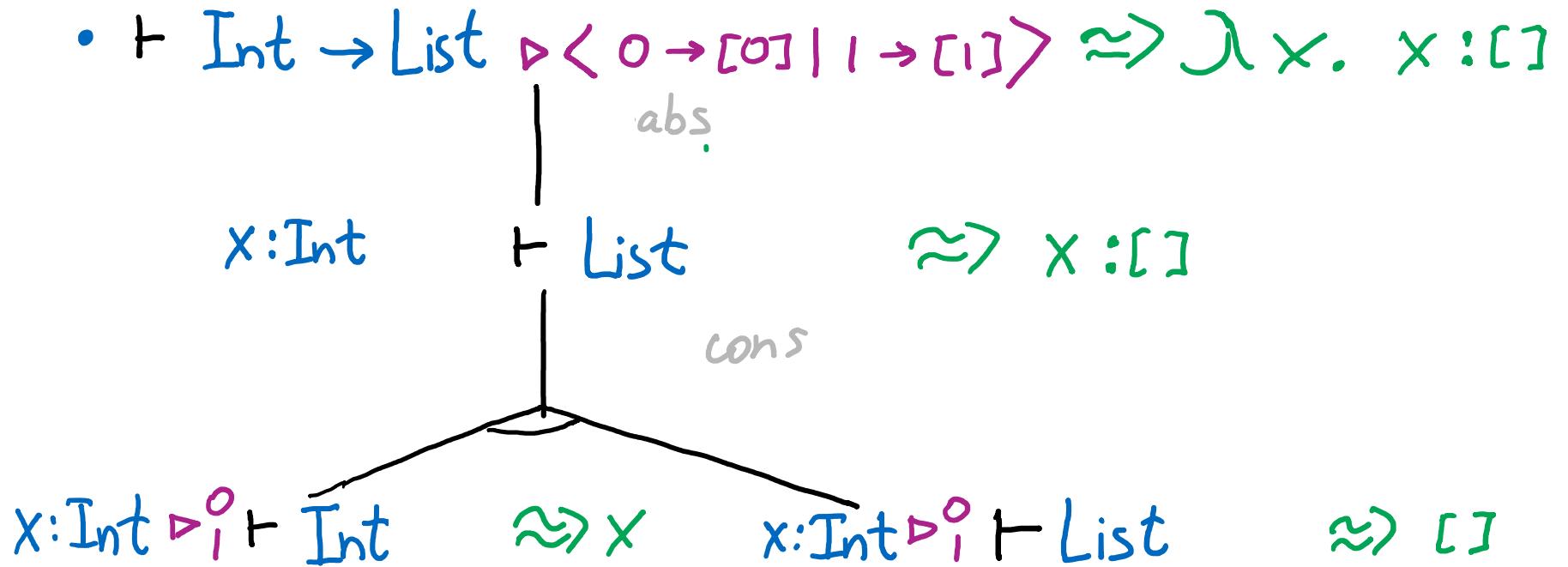
# Types + examples: syntax

---

$V ::= 0 \mid v+ \mid [] \mid V:V \mid \overline{V \rightarrow V} \mid \text{fix } f(x).e$	values
$X ::= \begin{matrix} v_1 \\ \vdots \\ v_n \end{matrix}$	vectors of examples
$R ::= T \triangleright X$	type refined with examples
$\Gamma ::= \cdot \mid x:R, \Gamma$	context

# Example: singleton

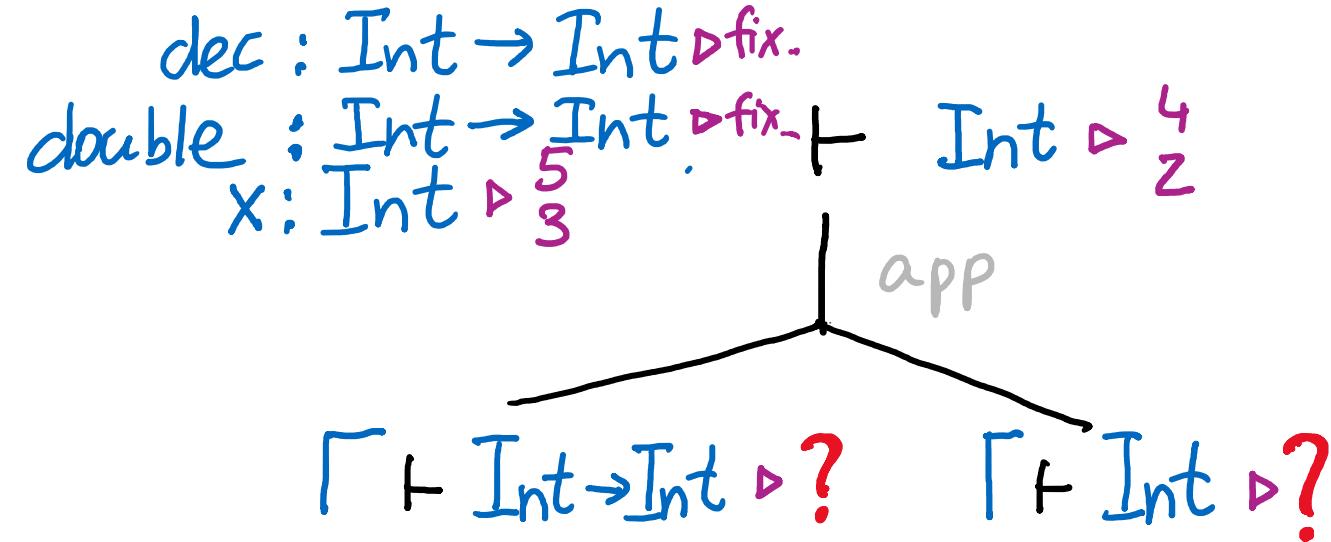
---



no search! simply propagate the spec top-down

# The hard part: application

---



no way to propagate spec through application 😞  
revert to guess-and-check

# types + examples: judgments

---

$$\Gamma \vdash R \approx i$$

i-refine

“under Gamma, refined type R generates i”

$$\Gamma \vdash T \rightsquigarrow e > X$$

e-guess

“under Gamma, type T generates e,  
which evaluates to X”

# Inference rules

---

$$\frac{i\text{-nil}}{\Gamma \vdash \text{List} \triangleright \boxed{\quad} \stackrel{?}{\sim} []}$$

$$\frac{i\text{-cons}}{\Gamma \vdash \text{Int} \triangleright \boxed{\quad} \stackrel{?}{\sim} i \quad \Gamma \vdash \text{List} \triangleright \boxed{\quad} \stackrel{?}{\sim} i'} \Gamma \vdash \text{List} \triangleright \boxed{\quad} \stackrel{?}{\sim} i:i'$$

# Inference rules

---

rules for i-refine judgements  
propagate examples top-down

$$\frac{i\text{-nil}}{\Gamma \vdash \text{List} \triangleright \frac{}{\vdash [] \approx []}} \quad \frac{i\text{-cons}}{\Gamma \vdash \text{Int} \triangleright \frac{v_1 : v_n \approx i}{\Gamma \vdash \text{List} \triangleright \frac{v_1 : v'_1 \approx i' \\ v_n : v'_n \approx i'}{v_1 : v'_1 \approx i' \\ v_n : v'_n \approx i'}}}$$

# Inference rules

---

rules for e-guess judgements  
perform type-directed enumeration  
and propagate examples bottom-up

$$\text{e-var} \frac{x:T \triangleright X \in \Gamma}{\Gamma \vdash T \rightsquigarrow x \triangleright X}$$

$$\text{e-app} \frac{\Gamma \vdash T' \rightarrow T \rightsquigarrow e \triangleright X \quad \Gamma \vdash T' \triangleright \cdot \approx i}{\Gamma \vdash T \rightsquigarrow e \ i \triangleright X(i)}$$

# Type-driven synthesis in 3 easy steps

---

1. Annotate types with examples
2. Design a type system for annotated types
3. Perform type-directed enumeration as before

# This week

---

intro to type systems

enumerating well-typed terms

synthesis with types and examples

polymorphic types

refinement types

synthesis with refinement types

# Polymorphic types

---

$\forall \alpha. \alpha \rightarrow \text{List } \alpha$

# Polymorphic types for synthesis

---

•  $\vdash ? :: \text{Int} \rightarrow \text{List Int}$

$\lambda x. \text{nil}$

$\lambda x. [0], \lambda x. [1], \dots$

$\lambda x. [x]$

$\lambda x. [\text{double } 0], \lambda x. [\text{dec } 0]$

$\lambda x. [0,0], \lambda x. [0,1], \dots$

$\lambda x. [x, x]$

•  $\vdash ? :: \forall d. d \rightarrow \text{List } d$

which of these programs  
match the polymorphic type?

# Polymorphic types for synthesis

---

•  $\vdash ? :: \text{Int} \rightarrow \text{List Int}$

$\lambda x. \text{nil}$

$\lambda x. [0], \lambda x. [1], \dots$

$\lambda x. [x]$

$\lambda x. [\text{double } 0], \lambda x. [\text{dec } 0]$

$\lambda x. [0,0], \lambda x. [0,1], \dots$

$\lambda x. [x, x]$

•  $\vdash ? :: \forall d. d \rightarrow \text{List } d$

1.  $\lambda x. \text{nil}$

eliminate ambiguity!

2.  $\lambda x. [x]$

prune the search!

3.  $\lambda x. [x, x]$

# Polymorphic types

---

$$B ::= \text{Int} \mid \text{List } B \mid \lambda$$

base types

$$T ::= B \mid T \rightarrow T$$

types

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

type schemas (polytypes)

$$\Gamma ::= \cdot \mid x : S, \Gamma \mid \lambda \lambda, \Gamma$$

contexts

# Judgments

---

$$\Gamma \vdash i \leqslant S$$

type checking:

“under context Gamma, i checks against a schema S”

$$\Gamma \vdash e \Rightarrow T$$

type inference:

“under context Gamma, e generates type T”

# Typing rules

$$\frac{i\text{-nil}}{\Gamma \vdash [] \Leftarrow \text{List } B}$$

$$\frac{i\text{-cons} \quad \Gamma \vdash i_1 \Leftarrow B \quad \Gamma \vdash i_2 \Leftarrow \text{List } B}{\Gamma \vdash i_1 : i_2 \Leftarrow \text{List } B}$$

$$\frac{i\text{-gen} \quad \Gamma, d \vdash i \Leftarrow S}{\Gamma \vdash i \Leftarrow \forall d. S}$$

$$\frac{e\text{-var} \quad x : \overline{\forall d. T} \in \Gamma}{\Gamma \vdash x \Rightarrow \overline{[\alpha \mapsto T']} T}$$

how do we guess  $T'$ ?  
Hindley-Milner type inference!

# This week

---

intro to type systems

enumerating well-typed terms

synthesis with types and examples

polymorphic types

refinement types

synthesis with refinement types

# Refinement types

---

Nat

base types

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

dependent  
function types

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

polymorphic  
datatypes

```
data List α where
  Nil  :: { List α | len v = 0 }
  Cons :: x: α → { List α | len v = len
                           xs + 1 }
```

```
measure Len :: List α → Int
  Len Nil = 0
  Len (Cons _ xs) = Len xs + 1
```

# Refinement types

---

$$B ::= \text{Int} \mid \text{List } B \mid d$$

base types

$$T ::= \{\nu:B. \mid \varphi\} \mid x:T \rightarrow T$$

types

$$S ::= T \mid \#d. S$$

type schemas (polytypes)

$$\Gamma ::= \cdot \mid x:S, \Gamma \mid d, \Gamma$$

contexts

# Example: increment

---

$$\begin{aligned} \text{Nat} &= \{\nu: \text{Int} \mid \nu \geq 0\} \\ \Gamma &= [\text{inc}: y: \text{Int} \rightarrow \{\nu: \text{Int} \mid \nu = y + 1\}] \end{aligned}$$

*we need subtyping!*

$$\Gamma \vdash \lambda x. \text{inc } x \Leftarrow \text{Nat} \rightarrow \text{Nat}$$

# 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  $\{\nu: \text{Int} \mid \nu = 5\} <: \text{Nat}$

$$\text{sub-base} \quad \frac{[\![\Gamma]\!] \wedge \phi' \Rightarrow \phi}{\Gamma \vdash \{\nu: B \mid \phi'\} <: \{\nu: B \mid \phi\}}$$

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

# Typing rules

---

$$\frac{i : e \quad \Gamma \vdash e \Rightarrow T \quad \Gamma \vdash T \leq \{B | \varphi\}}{\Gamma \vdash e \Leftarrow \{B | \varphi\}}$$

$$e\text{-app} \frac{\Gamma \vdash e \Rightarrow y:T_1 \rightarrow T_2 \quad \Gamma \vdash i \Leftarrow T_1}{\Gamma \vdash e \ i \Rightarrow [y \mapsto i]T_2}$$

# Example: increment

$\frac{e\text{-var}}{\Gamma, x:\text{Nat} \vdash \text{inc} \Rightarrow y:\text{Int} \rightarrow \{\text{Int} \mid v=y+1\}}$	$\frac{e\text{-var}}{\Gamma, x:\text{Nat} \vdash x \Rightarrow \text{Nat}} \quad \frac{e\text{-var}}{\Gamma, x:\text{Nat} \vdash \text{Nat} <: \text{Int}}$
$\frac{e\text{-app}}{\Gamma, x:\text{Nat} \vdash \text{inc } x \Rightarrow \{\text{Int} \mid v=x+1\}}$	$\frac{e\text{-app}}{\Gamma, x:\text{Nat} \vdash x \Leftarrow \text{Int}} \quad \frac{e\text{-app}}{\Gamma, x:\text{Nat} \vdash \{\text{Int} \mid v=x+1\} <: \text{Nat}}$
$\frac{i\text{-abs}}{\Gamma \vdash \lambda x. \text{inc } x \Leftarrow \text{Nat} \rightarrow \text{Nat}}$	

## subtyping constraints

$x:\text{Nat} \vdash \{v:\text{Int} \mid v = x + 1\} <:\text{Nat}$

$$x \geq 0 \wedge v = x + 1 \Rightarrow v \geq 0$$

## implications

$$\nu \geq 0 \Rightarrow \text{true}$$

*SMT solver: VALID!*

# Refinement type checking

---

**idea:** separate type checking into  
**subtyping constraint generation** and **subtyping constraint solving**

1. Generate a constraint for every subtyping premise in derivation
2. Reduce subtyping constraints to implications
3. Use SMT solver to check implications

# This week

---

intro to type systems

enumerating well-typed terms

synthesis with types and examples

polymorphic types

refinement types

synthesis with refinement types

# Synthesis from refinement types

---

specification

“duplicate every  
element in a list”

```
stutter ::  
  xs:List a →  
  {v:List a | len v =  
    2 * len xs}
```



code

```
match xs with  
  Nil → Nil  
  Cons h t →  
    Cons h (Cons h (stutter t))
```

[Polikarpova, Kuraj, Solar-Lezama, Program Synthesis from Polymorphic Refinement Types. 2016]

# Synthesis from refinement types

---

specification

“insert element  
into sorted list”

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



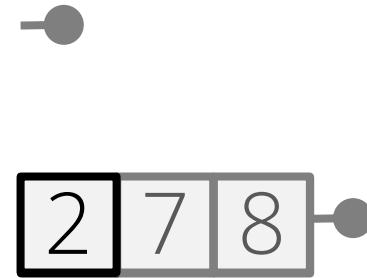
code

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

# 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
```

# Type-driven synthesis in 3 easy steps

---

1. Annotate types with logical predicates
2. Design a type system for annotated types
3. Perform type-directed enumeration as before

# Type-directed enumeration

---

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



```
insert = ??
```

# Type-directed enumeration

---

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



```
insert x xs = ??
```

context:  
x: a  
xs: SList a

# Type-directed enumeration

---

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

context:  
x: a  
xs: SList a



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

# Type-directed enumeration

---

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



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

context:  
x: a  
xs: SList a  
elems xs = {}

# Type-directed enumeration

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



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



context:  
x: a  
xs: SList a  
elems xs = {}

Constraints:  
 $\forall x: \{\} = \{\} \cup \{x\}$

SMT solver: INVALID!

# The hard part: application

---

```
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 ??)
```

should this program be rejected?  
yes!  
cannot guarantee output is sorted!

# The hard part: application

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


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



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

insert ::  $x:\tau \rightarrow$   
 $xs:SList \tau \rightarrow$   
 $SList \tau$

SMT solver: INVALID!

context:  
 $x: a$   
 $xs: SList a$   
 $h: a$   
 $t: SList \{a \mid h \leq v\}$

# Round-trip type-checking (RTTC)

---

$$\Gamma \vdash i \leqslant S$$

type checking:

“under context Gamma, i checks against schema S”

$$\Gamma \vdash e \leqslant T \Rightarrow T'$$

type strengthening:

“under context Gamma, e checks against type T and generates a stronger type T’”

# RTTC rules

---

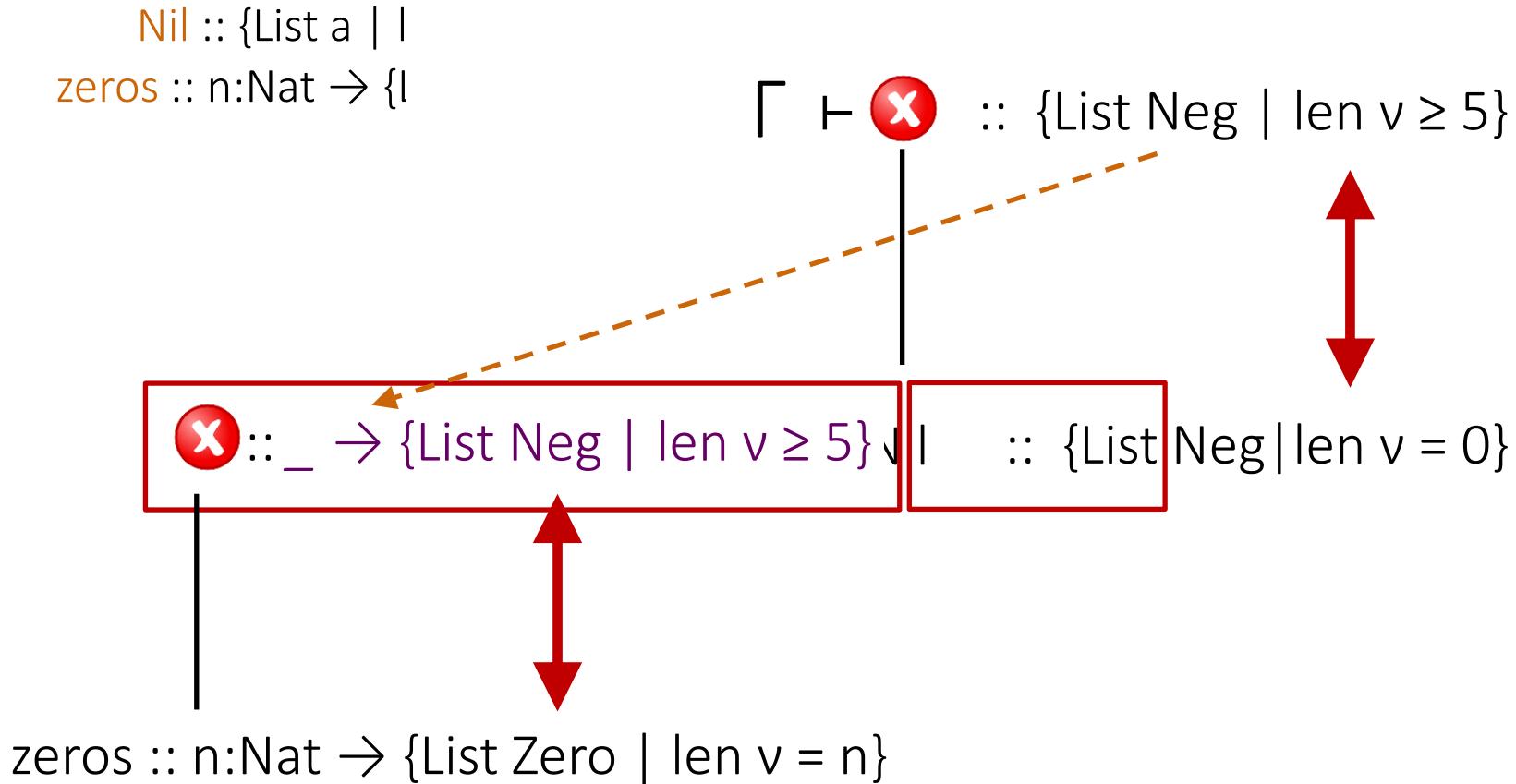
$$\text{ie } \frac{\Gamma \vdash e \Leftarrow \{B|\varphi\} \Rightarrow T}{\Gamma \vdash e \Leftarrow \{B|\varphi\}}$$

$$e-\forall\forall\Gamma \frac{x:T' \in \Gamma \quad \Gamma \vdash T' \leq : T}{\Gamma \vdash x \Leftarrow T \Rightarrow T'}$$

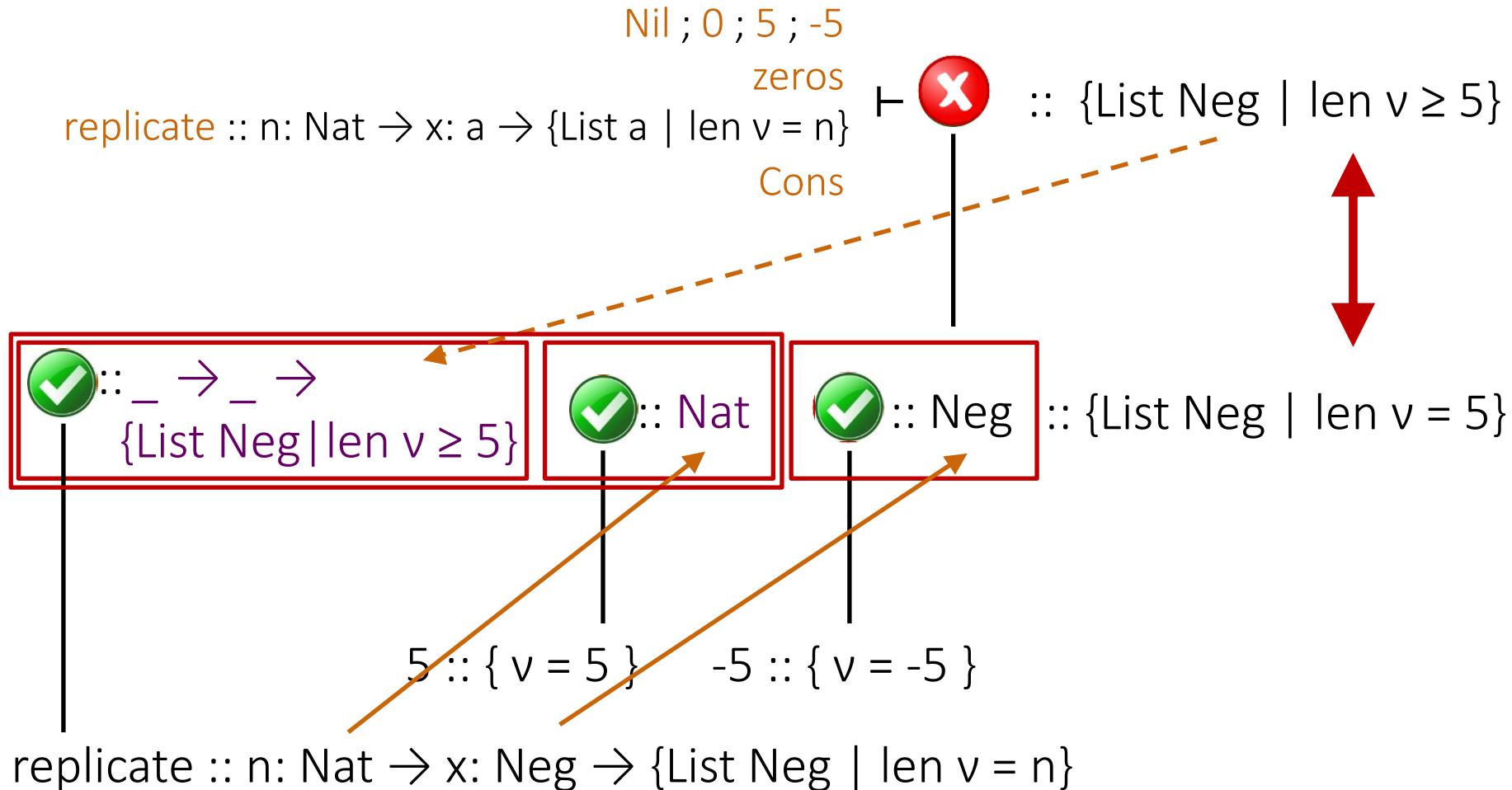
$$e\text{-app} \frac{\Gamma \vdash e \Leftarrow \perp \Rightarrow T \Rightarrow y:T_1 \rightarrow T_2 \quad \Gamma \vdash i \Leftarrow T_1}{\Gamma \vdash e i \Leftarrow T \Rightarrow [y \mapsto i]T_2}$$

# Example

---

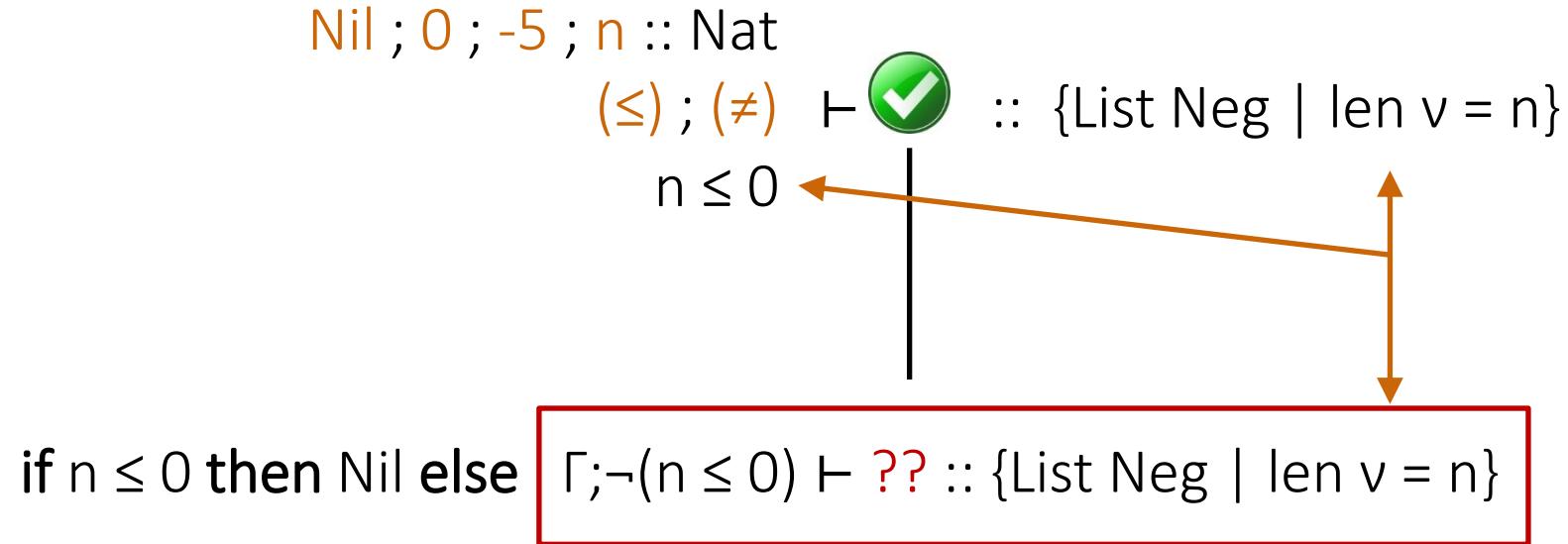


# Example



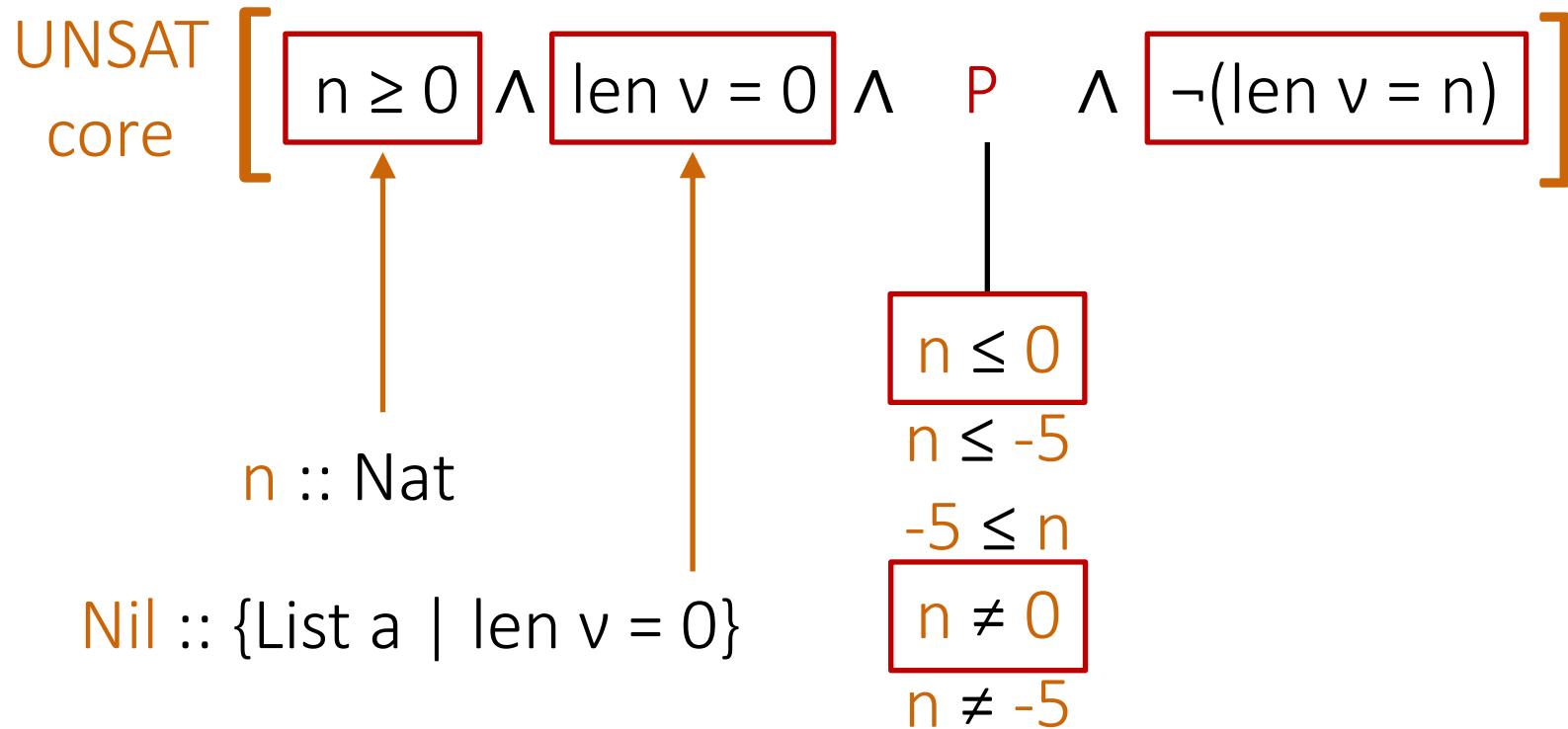
# Condition abduction

---



# 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

- $\{B_0 \mid \perp\} \rightarrow \{B_1 \mid \perp\} \rightarrow \{\text{List Pos} \mid \text{len } v = \underline{25}\}$

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