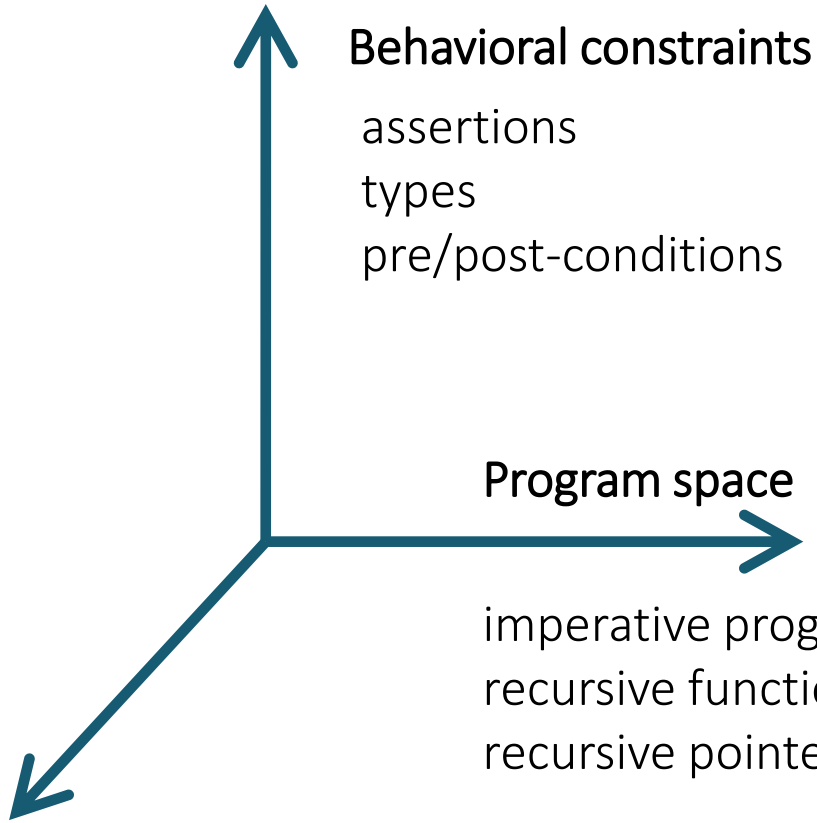
Lecture 11
Type-Driven Synthesis

Nadia Polikarpova

Module II



Behavioral constraints

assertions
types
pre/post-conditions

+ bounded guarantees

+ unbounded guarantees

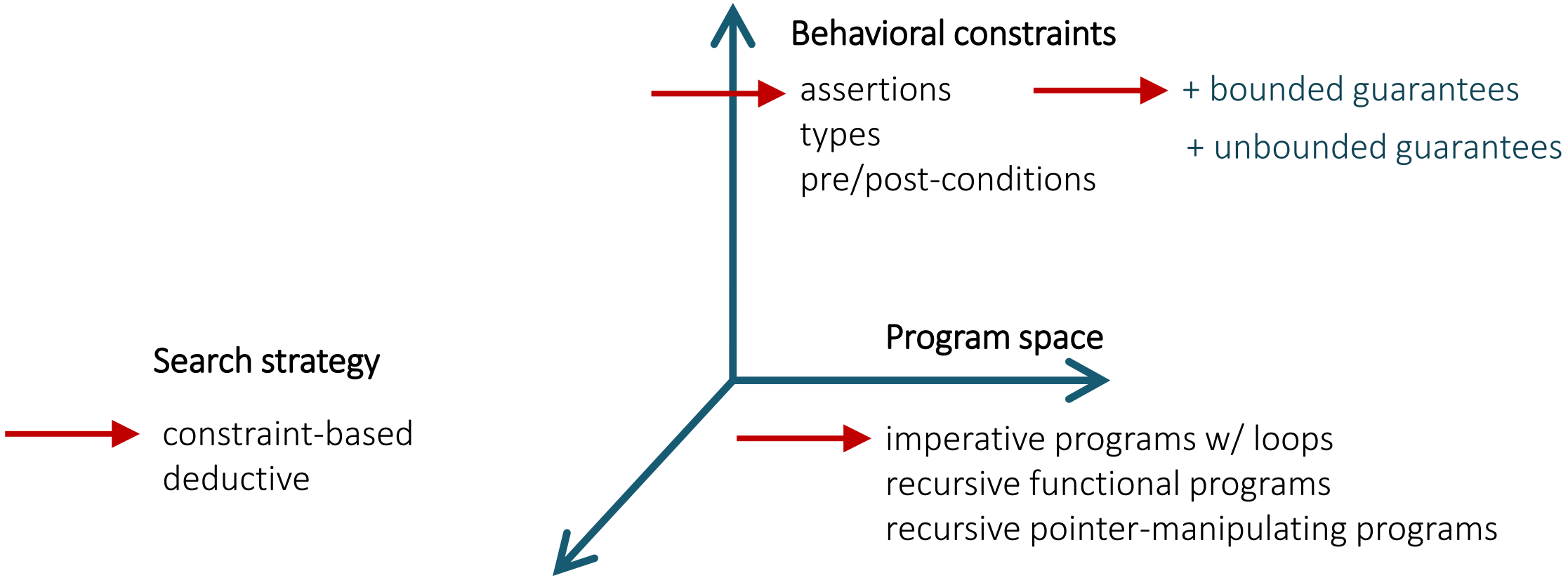
Program space

imperative programs w/ loops
recursive functional programs
recursive pointer-manipulating programs

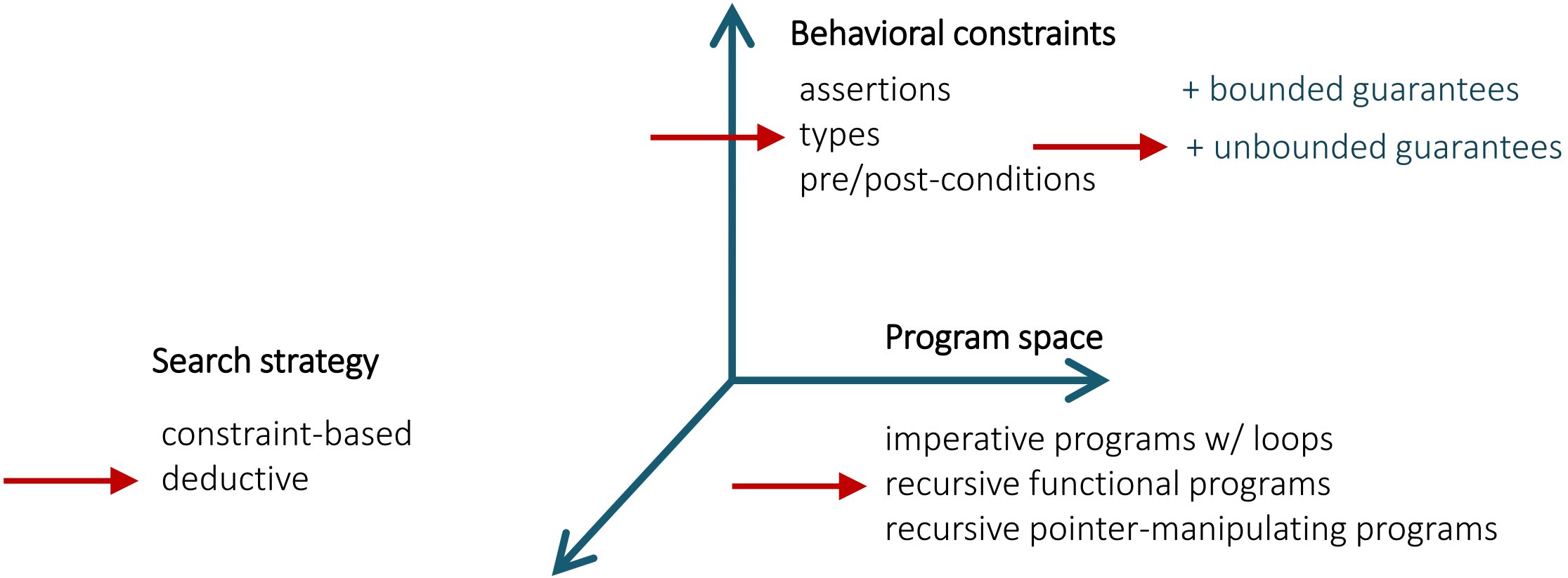
Search strategy

constraint-based
deductive

Last 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

This week

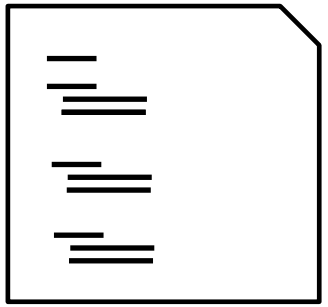

Behavioral constraints
assertions
types
pre/post-conditions
+ bounded guarantees
+ unbounded guarantees
Program space
imperative programs w/ loops
recursive functional programs
recursive pointer-manipulating programs
Search strategy
constraint-based
deductive

Type-driven program synthesis

specification

search

program



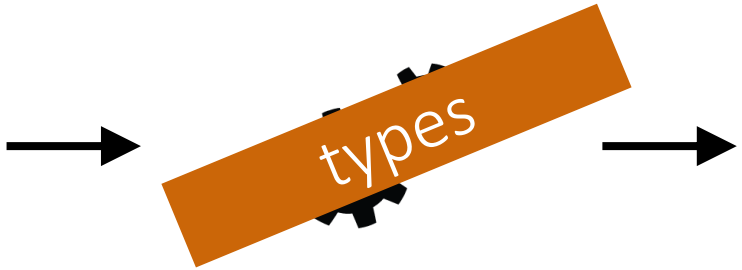
programmer-friendly
informative

Type-driven program synthesis

specification

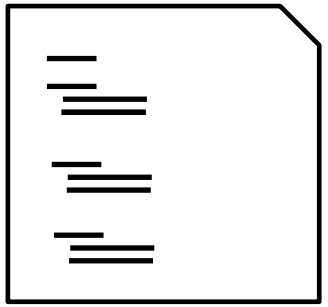
programmer-friendly
informative

search



help prune
the search space

program



Which program do I have in mind?

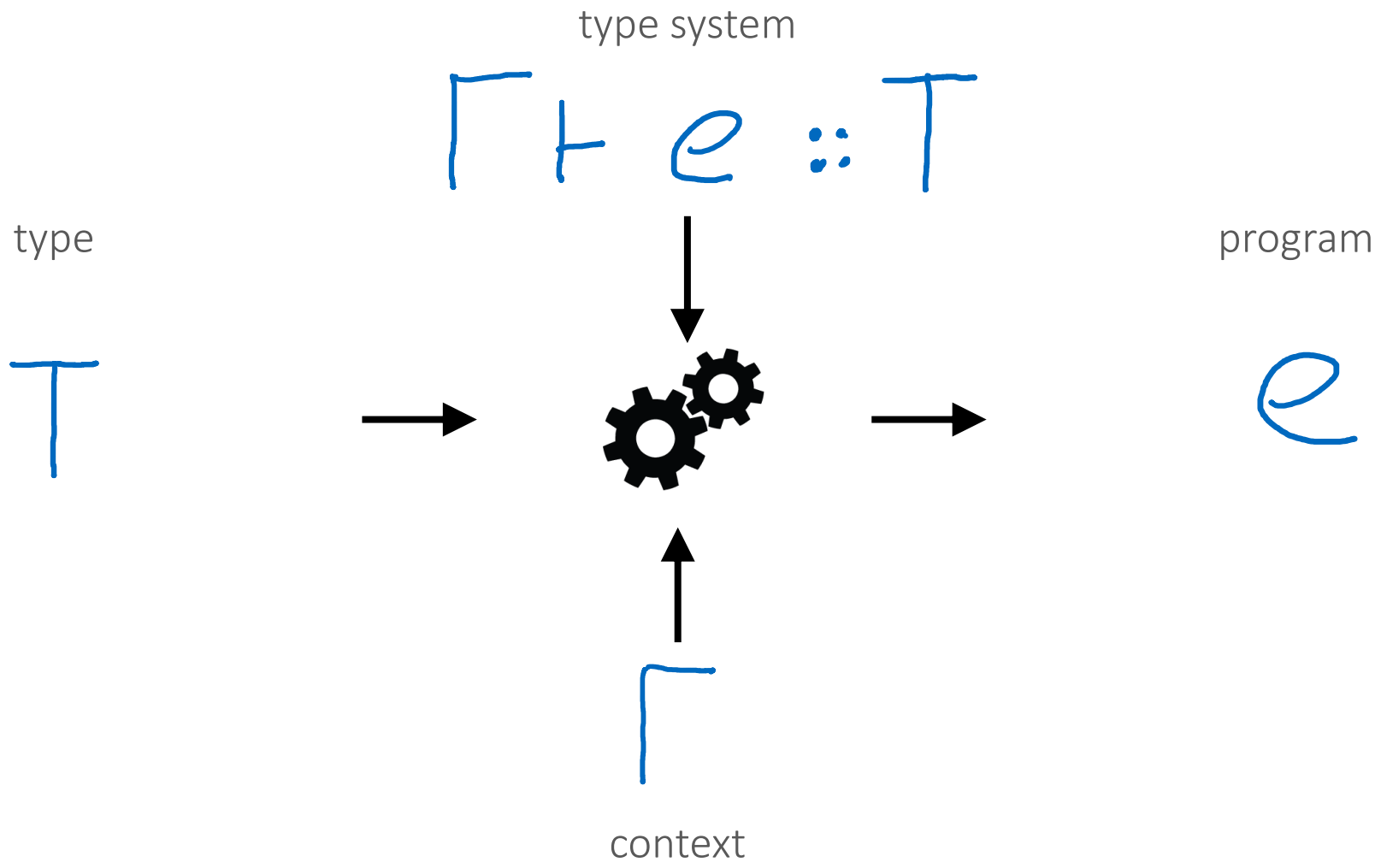
```
Char -> String -> [String]
```

split string at custom separator

```
a -> Int -> [a]
```

list with n copies of input value

Type-driven program synthesis



This week

intro to type systems

enumerating well-typed terms

bidirectional type systems

synthesis with types and examples

polymorphic types

refinement types

synthesis with refinement types

This week

intro to type systems

enumerating well-typed terms

bidirectional type systems

synthesis with types and examples

polymorphic types

refinement types

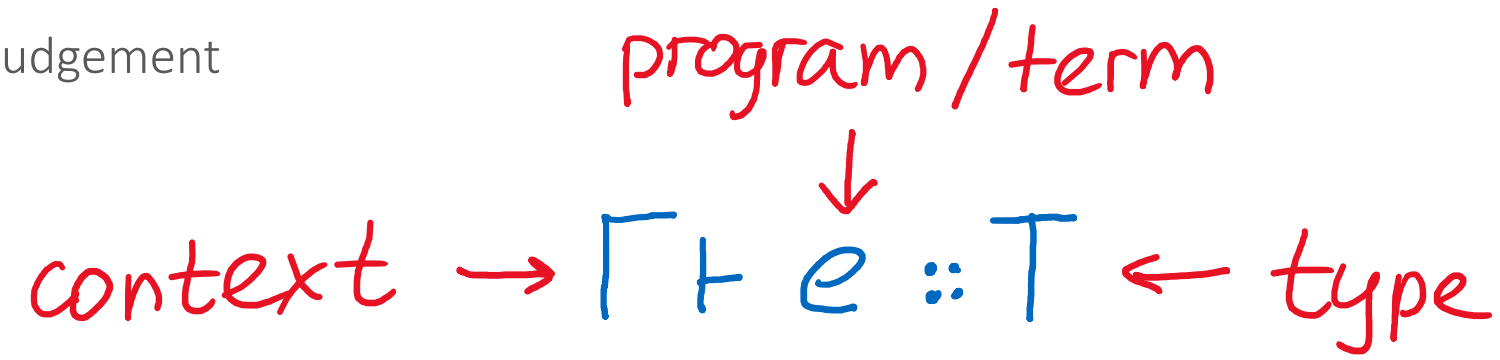
synthesis with refinement types

What is a type system?

Deductive system for proving facts about programs and types

Defined using *inference rules* over *judgments*

typing judgement



"under context Gamma, term e has type T"

A simple type system: syntax

$e ::= 0 \mid e + 1 \mid x \mid e\ e \mid \lambda x.e$ -- expressions

example program: increment by two $\lambda x.(x + 1) + 1$

A simple type system: syntax

$e ::= 0 \mid e+1 \mid x \mid e\ e \mid \lambda x.e$ -- expressions

$T ::= \mathrm{Int} \mid T \to T$ -- types

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

Inference rules = typing rules

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

$$\text{t-abs}\ \frac{\Gamma, x : T_1 \vdash e :: T_2}{\Gamma \vdash \lambda x.e :: T_1 \to T_2} \qquad \text{t-app}\ \frac{\Gamma \vdash e_1 :: T' \to T \qquad \Gamma \vdash e_2 :: T'}{\Gamma \vdash e_1\ e_2 :: T}$$

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

$$\cdot \vdash \lambda x.\, x + 1 :: \mathrm{Int} \to \mathrm{Int}$$

we say that $\lambda x.\, x + 1$ is well-typed in the empty context

and has type $\mathrm{Int} \to \mathrm{Int}$

Typing derivations

$$\text{t-zero}\ \frac{}{\Gamma \vdash 0 :: \text{Int}} \qquad \text{t-succ}\ \frac{\Gamma \vdash e :: \text{Int}}{\Gamma \vdash e+1 :: \text{Int}}$$

$$\text{t-var}\ \frac{x : T \in \Gamma}{\Gamma \vdash x :: T} \qquad \text{t-abs}\ \frac{\Gamma, x : T_1 \vdash e :: T_2}{\Gamma \vdash \lambda x.e :: T_1 \to T_2}$$

$$\text{t-app}\ \frac{\Gamma \vdash e_1 :: T' \to T \qquad \Gamma \vdash e_2 :: T'}{\Gamma \vdash e_1\, e_2 :: T}$$

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

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 ::= ... | [] | e:e | match e with [] → e | x:x → e

T ::= Int | List | T → T

Example program: head with default

$$\lambda x.\,\text{match } x \text{ with } nil \to 0 \mid y\colon ys \to y$$

Typing rules

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

what should the t-match tule be?

$$\text{t-match}\ \frac{\Gamma \vdash e_0 :: \square^{1} \quad \Gamma \vdash e_1 :: \square^{2} \quad \Gamma\ \square^{4} \vdash e_2 :: \square^{3}}{\Gamma \vdash \text{match } e_0 \text{ with } [] \to e_1 \mid x : xs \to e_2 :: T}$$

Typing rules

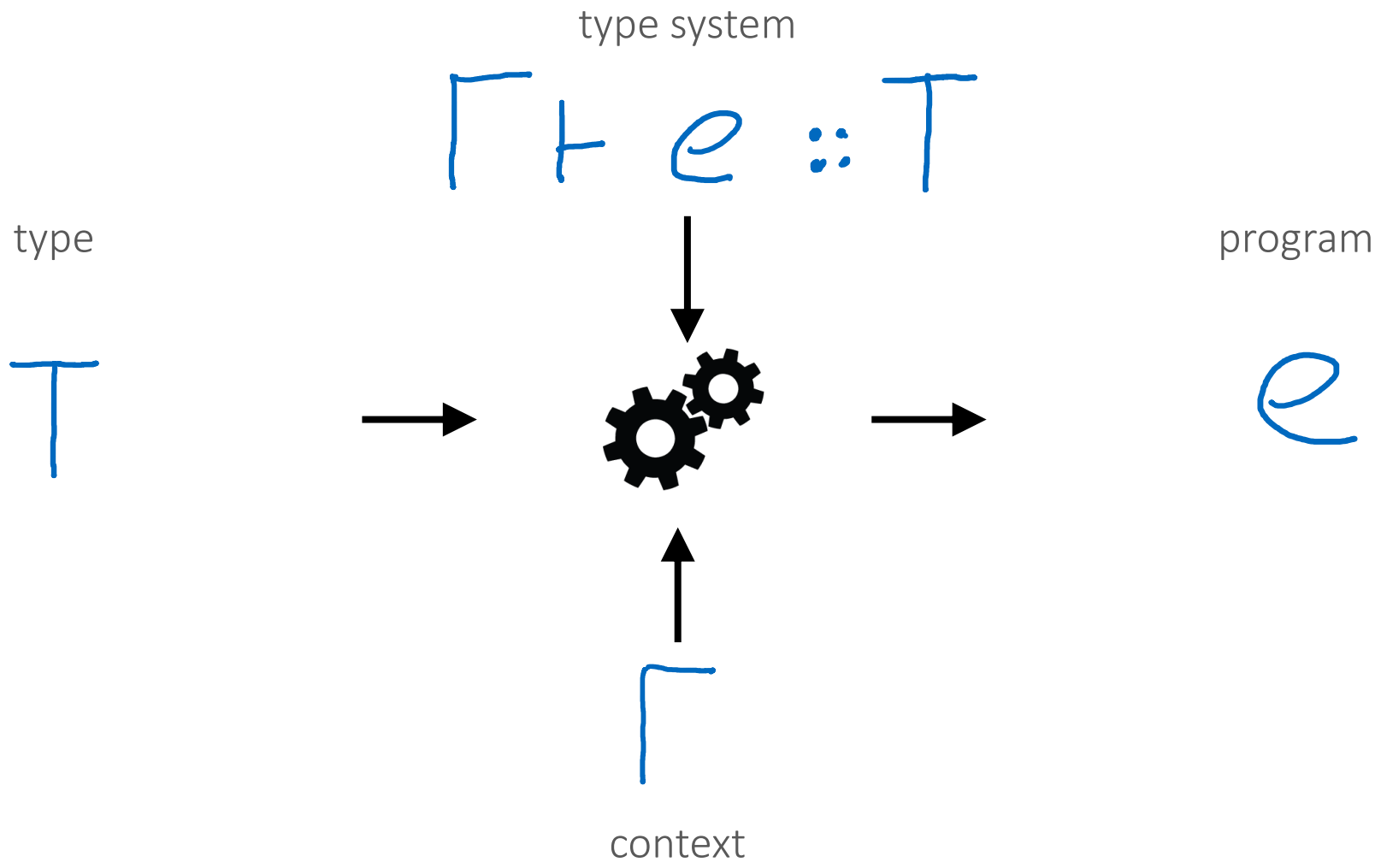
$$\text{t-nil}\ \frac{}{\Gamma \vdash [] :: \text{List}} \qquad \text{t-cons}\ \frac{\Gamma \vdash e_1 :: \text{Int} \quad \Gamma \vdash e_2 :: \text{List}}{\Gamma \vdash e_1 : e_2 :: \text{List}}$$

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

Example: head with default

$$\cdot \vdash \lambda x.\,\mathrm{match}\ x\ \mathrm{with}\ nil \to 0 \mid y{:}\,ys \to y\ \ ::\ \mathrm{List} \to \mathrm{Int}$$

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

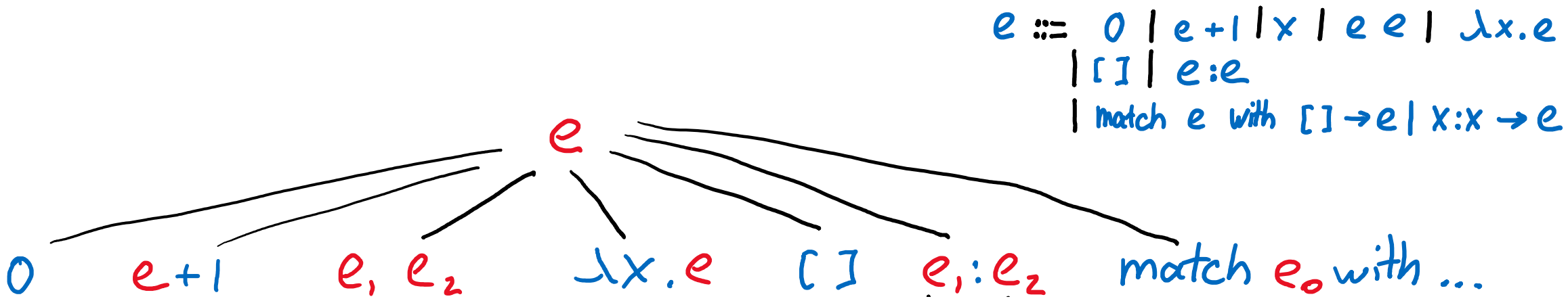
how should I enumerate all terms of type $\mathrm{List} \to \mathrm{List}$?

(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 List → List!

can we do better?

Enumerating well-typed terms

how should I enumerate all terms of type $\mathrm{List} \to \mathrm{List}$?

(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)

Synthesis as proof search

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

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

search strategy: top-down enumeration of derivation trees

like syntax-guided top-down enumeration but

derivation trees instead of ASTs

typing rules instead of grammar

Type-guided enumeration

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

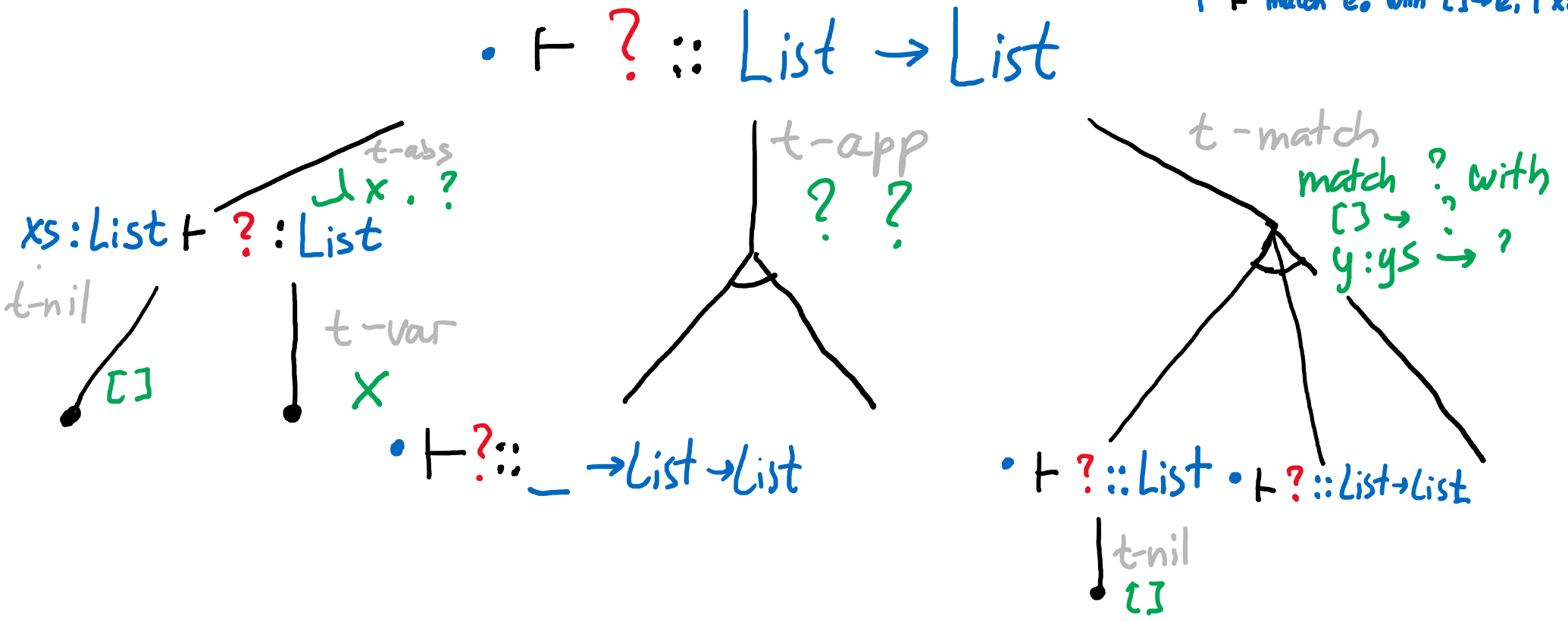
$$\text{t-abs}\ \frac{\Gamma, x:T_1 \vdash e :: T_2}{\Gamma \vdash \lambda x.e :: T_1 \to T_2} \qquad \text{t-app}\ \frac{\Gamma \vdash e_1 :: T' \to T \qquad \Gamma \vdash e_2 :: T'}{\Gamma \vdash e_1\ e_2 :: T}$$

$$\text{t-nil}\ \frac{}{\Gamma \vdash [] :: \text{List}} \qquad \text{t-cons}\ \frac{\Gamma \vdash e_1 :: \text{Int} \qquad \Gamma \vdash e_2 :: \text{List}}{\Gamma \vdash e_1 : e_2 :: \text{List}}$$

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

only 2 programs fully constructed!

all other programs *rejected early*



This week

intro to type systems

enumerating well-typed terms

bidirectional type systems

synthesis with types and examples

polymorphic types

refinement types

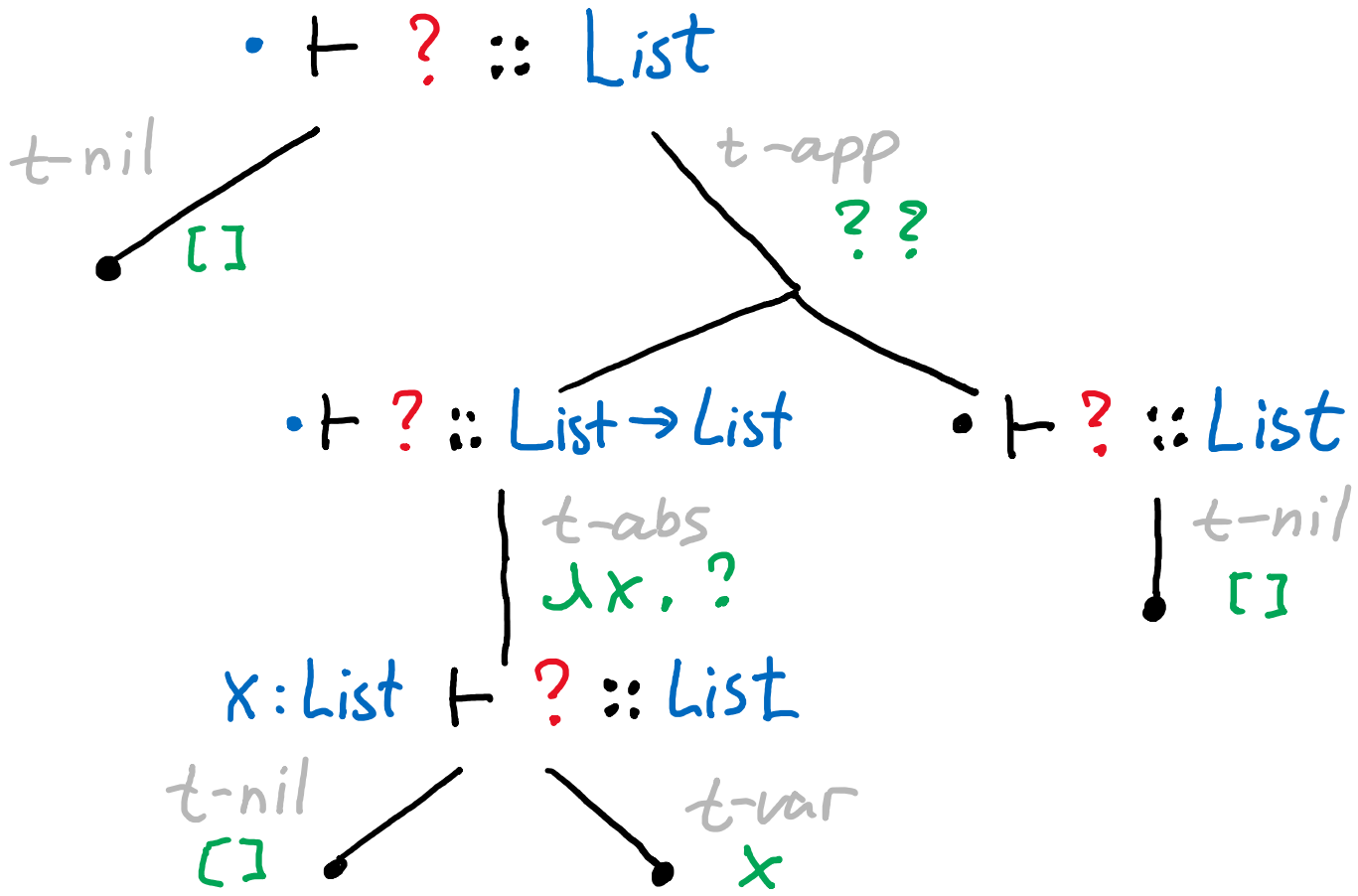
synthesis with refinement types

Bidirectional type system

Makes top-down propagation of types explicit

Helps with equivalence reduction

What's wrong with this search?



Enumerated 3 programs:

nil

$(\lambda x.\, x)\ nil$

$(\lambda x.\, nil)\ nil$

They are all equivalent!

Redundant programs

β-reduction

$(\lambda x.\, e_1)\ e_2 \longrightarrow e_1\,[x := e_2]$

match-nil

```
match [] with
  [] → e1
  y:ys → e2
```

$\longrightarrow e_1$

match-cons

```
match e1 : e2 with
  [] → e3
  y:ys → e4
```

$\longrightarrow e_4\,[y := e_1, ys := e_2]$

Generating programs on the left is a waste of time!

Idea: only generate programs *in normal form*

Restrict type system to make redundant programs ill-typed

Normal-form programs

$$e ::= x \mid e\ i$$

elimination forms

$$i ::= 0 \mid i+1 \mid \lambda x.\, i \mid [\,] \mid i : i$$
$$\mid \text{match } e \text{ with } [\,] \to i \mid x : x \to i$$

introduction forms

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

base types

$$T ::= B \mid T \to T$$

types

Bidirectional typing judgments

"under context Gamma, i checks against type T"

"under context Gamma, e generates type T"

[Pierce, Turner. Local Type Inference. 2000]

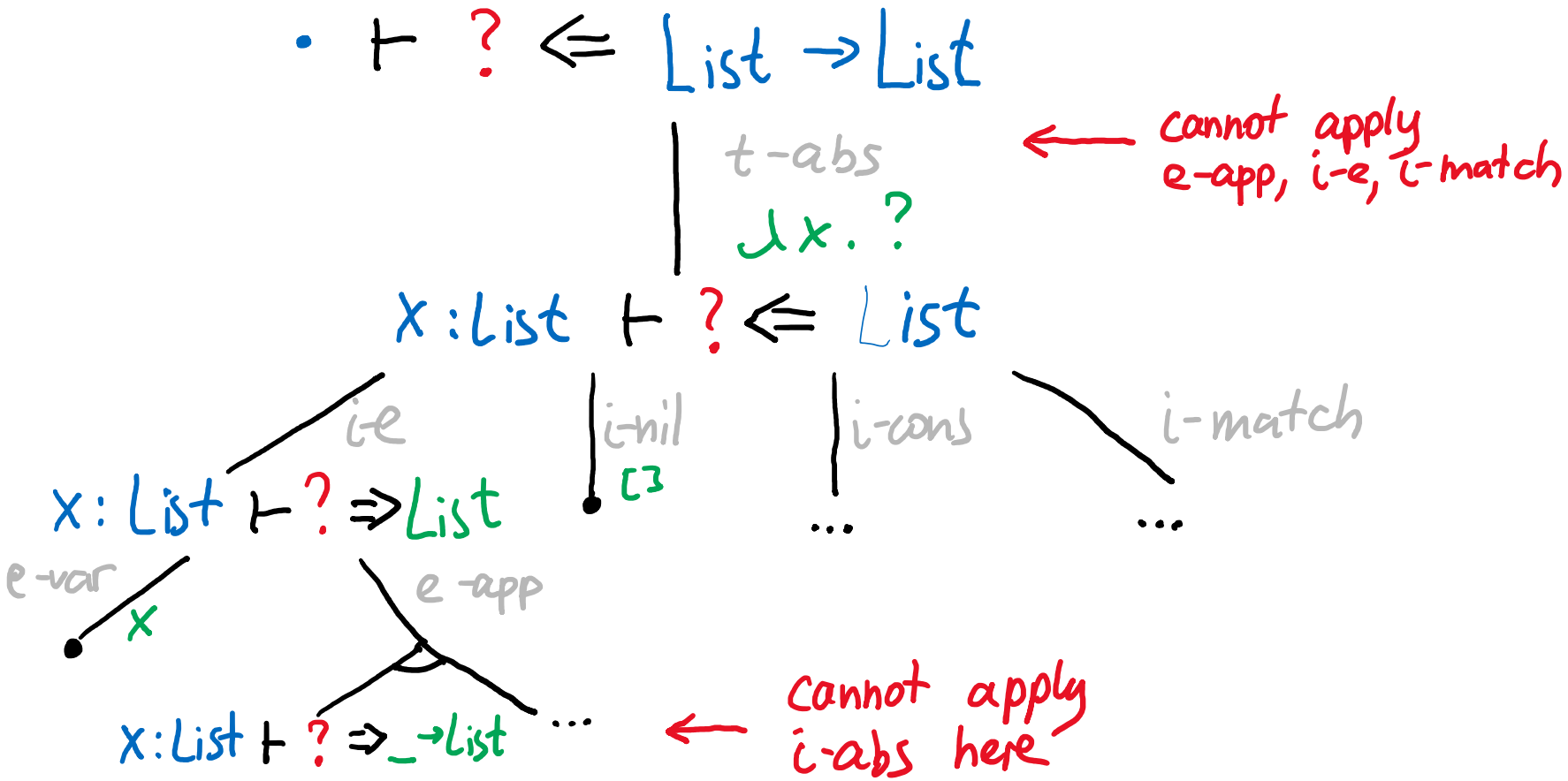
Bidirectional typing rules

$$\text{e-var}\ \frac{x : T \in \Gamma}{\Gamma \vdash x \Rightarrow T} \qquad \text{e-app}\ \frac{\Gamma \vdash e \Rightarrow T' \to T \quad \Gamma \vdash i \Leftarrow T'}{\Gamma \vdash e\ i \Rightarrow T}$$

$$\text{i-e}\ \frac{\Gamma \vdash e \Rightarrow B}{\Gamma \vdash e \Leftarrow B} \qquad \text{i-nil}\ \frac{}{\Gamma \vdash [] \Leftarrow \text{List}} \qquad \text{i-cons}\ \frac{\Gamma \vdash i_1 \Leftarrow \text{Int} \quad \Gamma \vdash i_2 \Leftarrow \text{List}}{\Gamma \vdash i_1 : i_2 \Leftarrow \text{List}}$$

$$\text{i-abs}\ \frac{\Gamma, x : T_1 \vdash i \Leftarrow T_2}{\Gamma \vdash \lambda x.\, i \Leftarrow T_1 \to T_2} \qquad \text{i-match}\ \frac{\Gamma \vdash e \Rightarrow \text{List} \quad \Gamma \vdash i_1 \Leftarrow B \quad \Gamma, y : \text{Int}, ys : \text{List} \vdash i_2 \Leftarrow B}{\Gamma \vdash \text{match}\ e\ \text{with}\ [] \Rightarrow i_1 \mid y : ys \Rightarrow i_2 \Leftarrow B}$$

Type-guided enumeration



This week

intro to type systems

enumerating well-typed terms

bidirectional type systems

synthesis with types and examples

polymorphic types

refinement types

synthesis with refinement types

Simple types are not enough

specification

code

"duplicate every element in a list"

```
stutter :: List → List
```

$\lambda xs.\, xs$

Simple types are not enough

specification

code

"insert element into sorted list"

ambiguous!

```
insert
  Int → List → List
```

$\lambda x.\, \lambda xs.\, xs$

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

bidirectional type systems

synthesis with types and examples

polymorphic types

refinement types

synthesis with refinement types

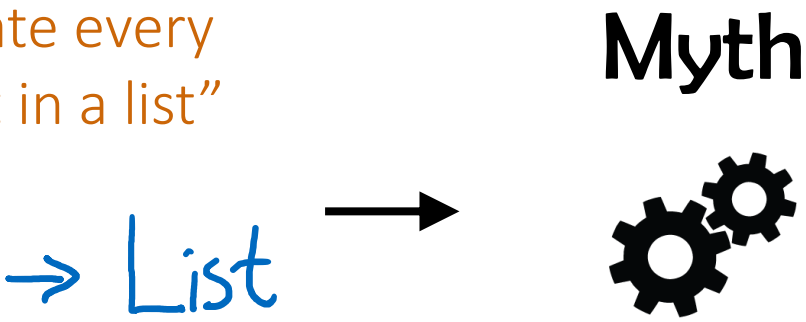
Type + examples

specification

code

"duplicate every element in a list"

List → List



$$\text{fix } f\,(xs).\,\text{match } xs \text{ with}$$
$$\quad nil \rightarrow nil$$
$$\mid\ y{:}ys \rightarrow y{:}y{:}f\,(ys)$$

▷ ⟨ [] → [] | [0] → [0,0] | [1,0] → [1,1,0,0] ⟩

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

Types + examples: syntax

partial function

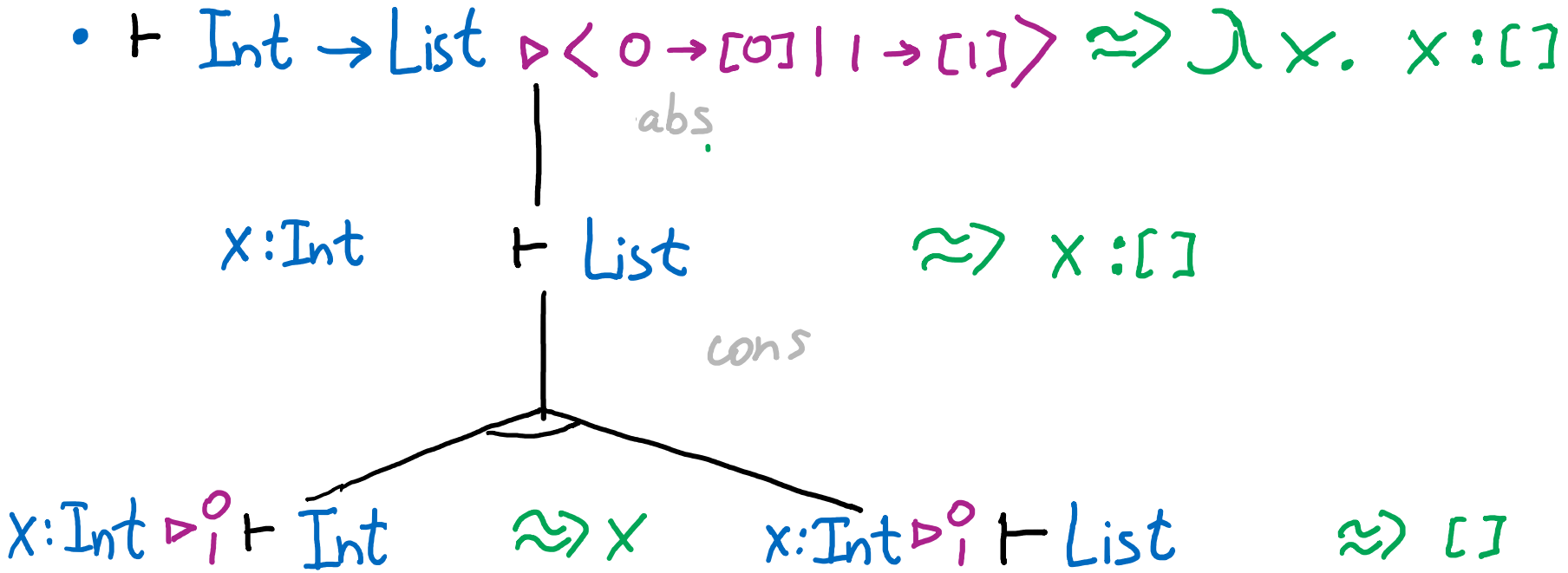
$$v ::= 0 \mid v+1 \mid [] \mid v:v \mid \overline{v \to v}$$ values

$$X ::= \begin{matrix} v_1 \\ \vdots \\ v_n \end{matrix}$$ 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

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

bidirectional type systems

synthesis with types and examples

polymorphic types

refinement types

synthesis with refinement types

Polymorphic types

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

Polymorphic types for synthesis

- $\vdash ? :: \mathtt{Int} \to \mathtt{List\ Int}$

$\lambda x.\, nil$

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

$\lambda x.\, [x]$

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

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

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

- $\vdash ? :: \forall \alpha.\ \alpha \to \mathtt{List}\ \alpha$

which of these programs match the polymorphic type?

Polymorphic types for synthesis

- $\vdash ? :: \mathrm{Int} \to \mathrm{List\ Int}$

$\lambda x.\, nil$

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

$\lambda x.\, [x]$

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

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

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

- $\vdash ? :: \forall \alpha.\ \alpha \to \mathrm{List}\ \alpha$

1. $\lambda x.\, nil$

eliminate ambiguity!

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

prune the search!

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

Polymorphic types

$B ::= \mathrm{Int} \mid \mathrm{List}\ B \mid \alpha$ base types

$T ::= B \mid T \to T$ types

$S ::= T \mid \forall \alpha . S$ type schemas (polytypes)

$\Gamma ::= \cdot \mid x : S, \Gamma \mid \alpha, \Gamma$ contexts

Judgments

$\Gamma \vdash i \Leftarrow S$

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

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

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

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

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

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

This week

intro to type systems

enumerating well-typed terms

bidirectional type systems

synthesis with types and examples

polymorphic types

refinement types

synthesis with refinement types

Refinement types

```
Nat
```

base types

```
max :: x: Int → y: Int → { v: Int | x ≤ v ∧ y ≤ v }
```

dependent function types

```
xs :: { v: List Nat }
```

polymorphic datatypes

[Rondon, Kawaguchi, Jhala. PLDI'2008]

Refinement types: measures

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

syntactic sugar:

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

example: duplicate every element in a list

```
stutter :: ??
```

Refinement types: sorted lists

```
data SList α where
    Nil  ::  SList α
    Cons  ::  x: α  → xs: SList {α | x ≤ ν }
                    → SList α
```

example: insert an element into a sorted list

```
insert :: ??
```

Refinement types

$B ::= \mathrm{Int} \mid \mathrm{List}\ B \mid \alpha$ — base types

$T ::= \{\nu : B \mid \varphi\} \mid x : T \to T$ — types

$S ::= T \mid \forall \alpha . S$ — type schemas (polytypes)

$\Gamma ::= \cdot \mid x : S, \Gamma \mid \alpha, \Gamma$ — contexts

Example: increment

$$\mathrm{Nat} = \{\nu\colon \mathrm{Int} \mid \nu \geq 0\}$$
$$\Gamma = [\mathrm{inc}\colon y\colon \mathrm{Int} \to \{\nu\colon \mathrm{Int} \mid \nu = y + 1\}]$$

we need subtyping!

$$\Gamma \vdash \lambda x.\, \mathrm{inc}\; x \Leftarrow \mathrm{Nat} \to \mathrm{Nat}$$

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

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

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

$\text{Pos} <: \text{Nat}$ ✓

$\text{Int} \to \text{Int} <: \text{Int} \to \text{Nat}$ ✗

$\text{Int} \to \text{Int} <: \text{Nat} \to \text{Int}$ ✓

$\text{x:Int} \to \{\text{Int} \mid \nu = x + 1\} <: \text{Nat} \to \text{Nat}$ ✓

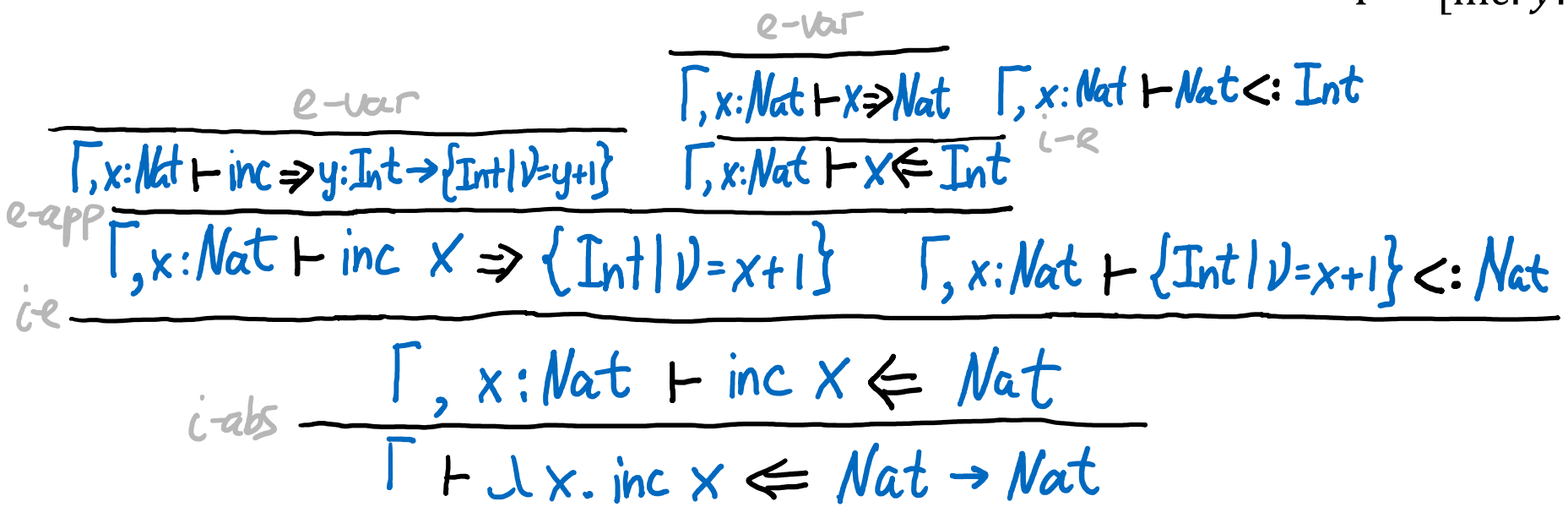
Typing rules

$$\text{i-e}\ \frac{\Gamma \vdash e \Rightarrow T \qquad \Gamma \vdash T <: \{B \mid \varphi\}}{\Gamma \vdash e \Leftarrow \{B \mid \varphi\}}$$

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

Example: increment

$\Gamma = [\text{inc}: y: \text{Int} \to \{v: \text{Int} \mid v = y + 1\}]$



subtyping constraints

$$\text{Nat} <: \text{Int}$$

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

implications

$$v \geq 0 \Rightarrow true$$

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

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

[Rondon, Kawaguchi, Jhala. PLDI'2008]

This week

intro to type systems

enumerating well-typed terms

bidirectional type systems

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)
```

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 for insert

```
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

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

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

context:
x: a
xs: SList a

Type-directed enumeration

$$\{\nu : \texttt{SList a} \mid \texttt{elems } \nu = \texttt{elems xs} \cup \{\texttt{x}\}\}$$

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

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

Type-directed enumeration

$$\{\nu : \texttt{SList a} \mid \texttt{elems } \nu = \texttt{elems xs} \cup \{\texttt{x}\}\}$$

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

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

Constraints:
∀x: {} = {} ∪ {x}

SMT solver: INVALID!

The hard part: application

```
x:a  →  xs:SList a  →
  {ν:SList a | elems ν = 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!

Round-trip type-checking (RTTC)

$\Gamma \vdash i \Leftarrow S$

type checking:

"under context Gamma, i checks against schema S"

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

type strengthening:

"under context Gamma, e checks against type T *and* generates a stronger type T'"

RTTC rules

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

$$\text{e-var}\ \frac{x : T' \in \Gamma \qquad \Gamma \vdash T' <: T}{\Gamma \vdash x \Leftarrow T \Rightarrow T'}$$

$$\text{e-app}\ \frac{\Gamma \vdash e \Leftarrow \bot \to T \Rightarrow y : T_1 \to T_2 \qquad \Gamma \vdash i \Leftarrow T_1}{\Gamma \vdash e\ i \Leftarrow T \Rightarrow [y \mapsto i]\, T_2}$$

The hard part: application

elems will depend on the missing part…

but sortedness we can already check!

```
{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 ??)
```

The hard part: application

$\{\nu$:a | h ≤ $\nu\}$

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

context:
x: a
xs: SList a
h: a
t: SList {a|h≤ν}

Constraints:
∀x, h: h ≤ x

SMT solver: INVALID!

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

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)