

#22: Type-directed Synthesis - II

Sankha Narayan Guria

EECS 700: Introduction to Program Synthesis



This week

intro to type systems

enumerating well-typed terms

bidirectional type systems

synthesis with types and examples

polymorphic types

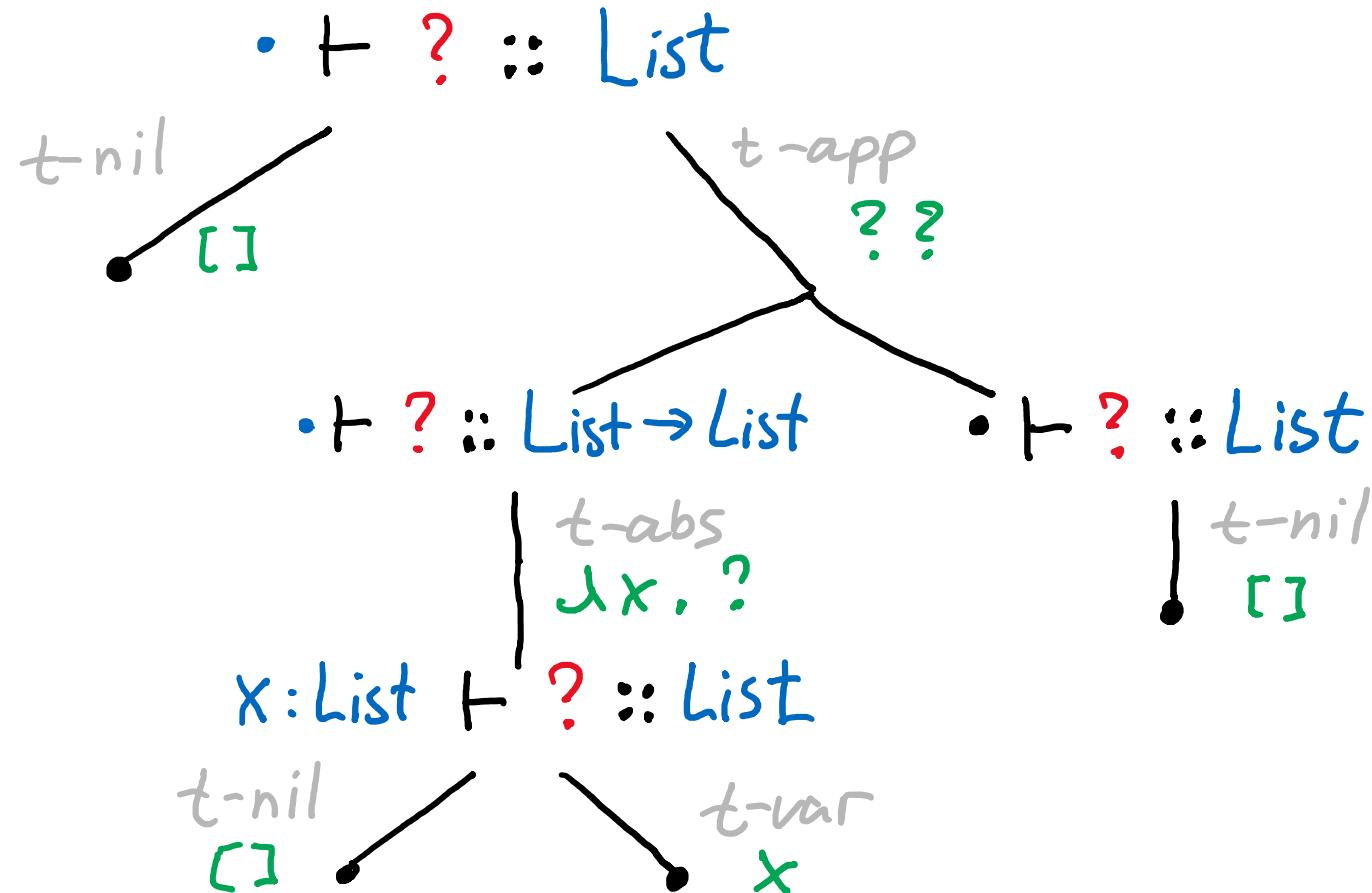
refinement types

synthesis with refinement types

Bidirectional type system

- Makes top-down propagation of types explicit
- Helps with equivalence reduction

What's wrong with this search?



Enumerated 3 programs:

1. nil
2. $(\lambda x. x) \text{ nil}$
3. $(\lambda x. \text{nil}) \text{ nil}$

They are all equivalent!

Redundant programs

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

$$\begin{array}{l} \text{match } [] \text{ with} \\ \quad [] \rightarrow e_1 \\ \quad y:ys \rightarrow e_2 \end{array} \xrightarrow{\text{match-nil}} e_1$$

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

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

$e ::= x \mid ei$

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

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

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

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

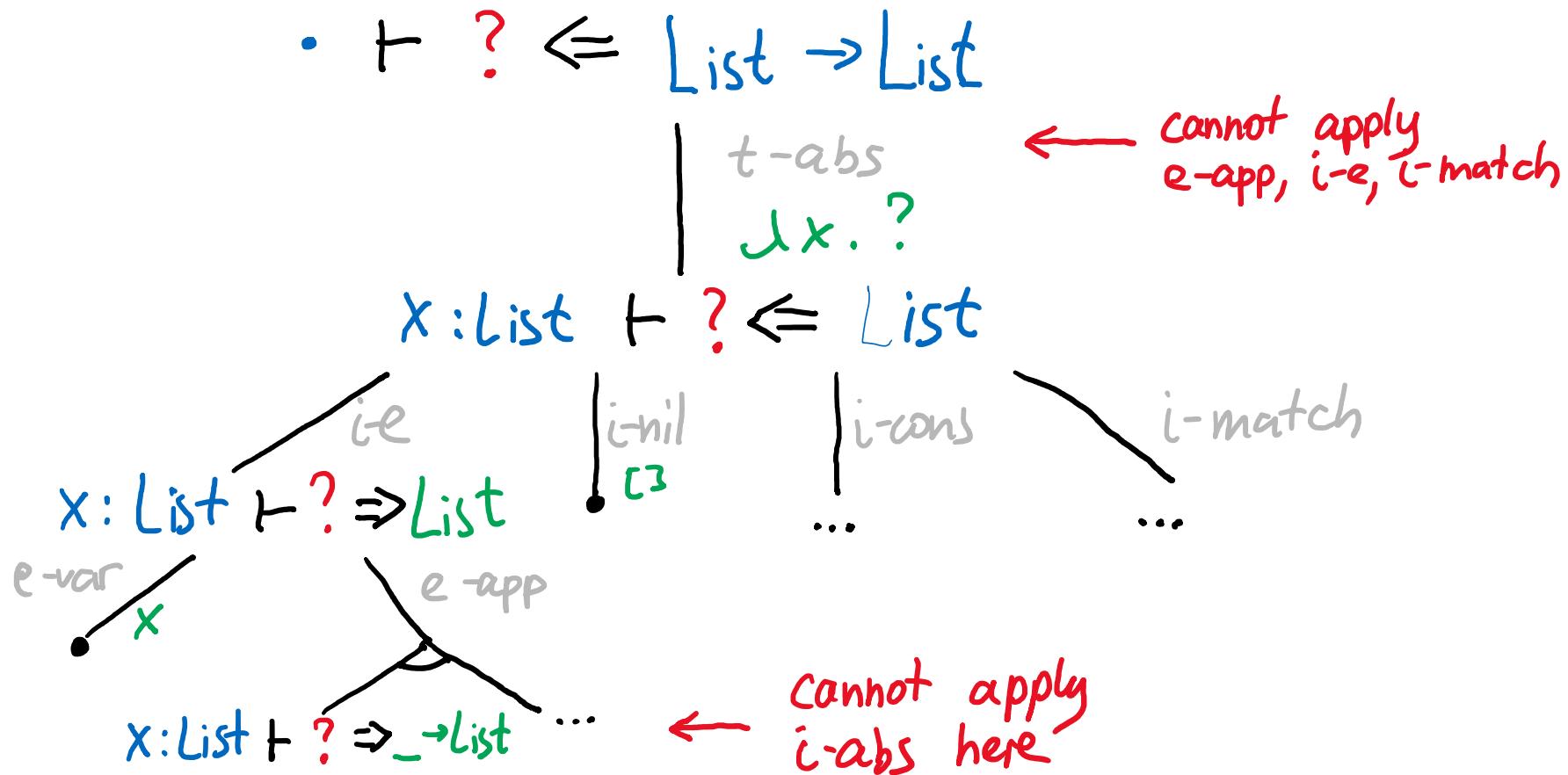
$$\text{i-nil} \frac{}{\Gamma \vdash [] \Leftarrow \text{List}}$$

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

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

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

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

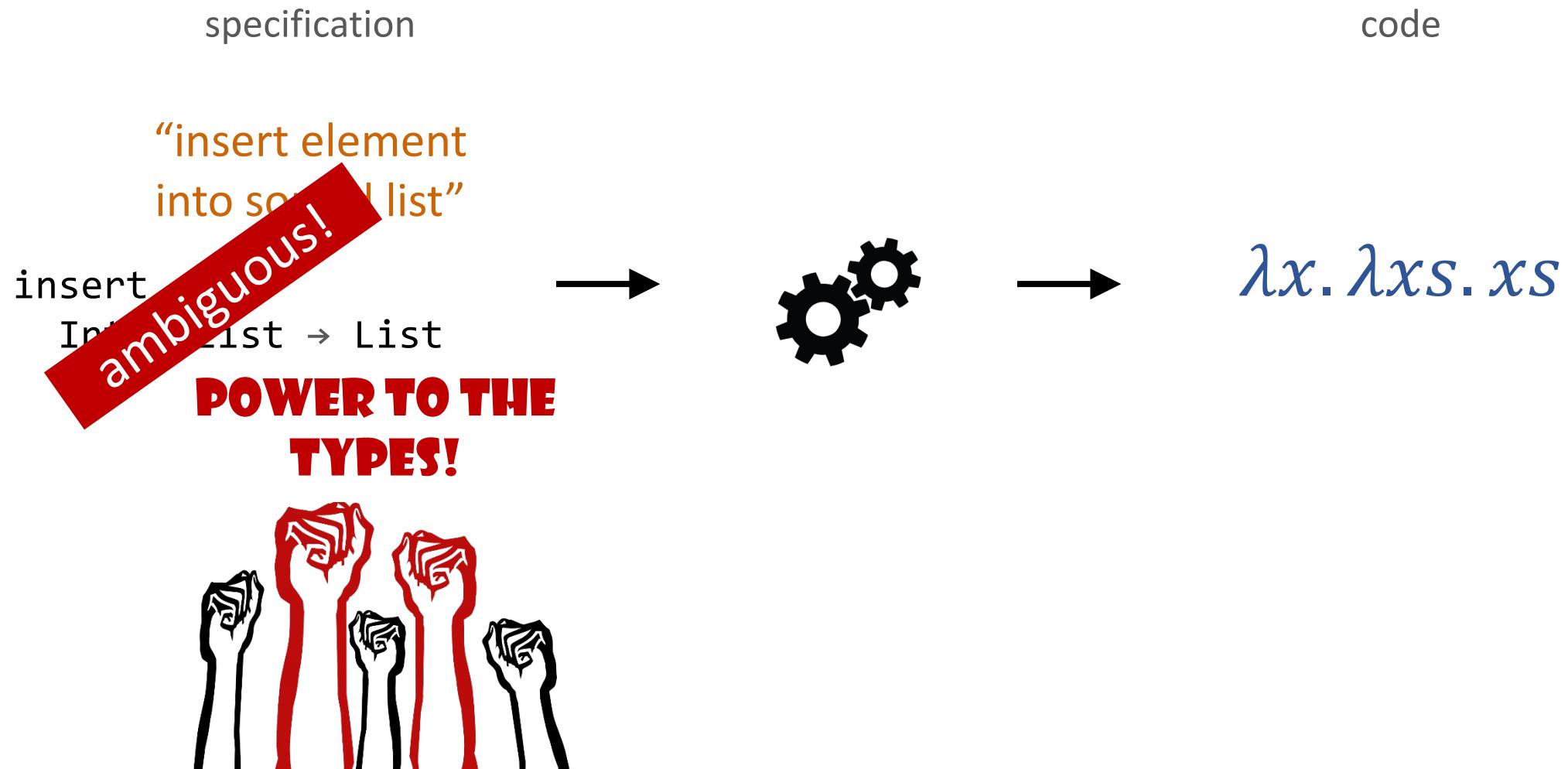
“duplicate every
element in a list”

stutter :: List \rightarrow List \longrightarrow



$\lambda xs. xs$

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

bidirectional type systems

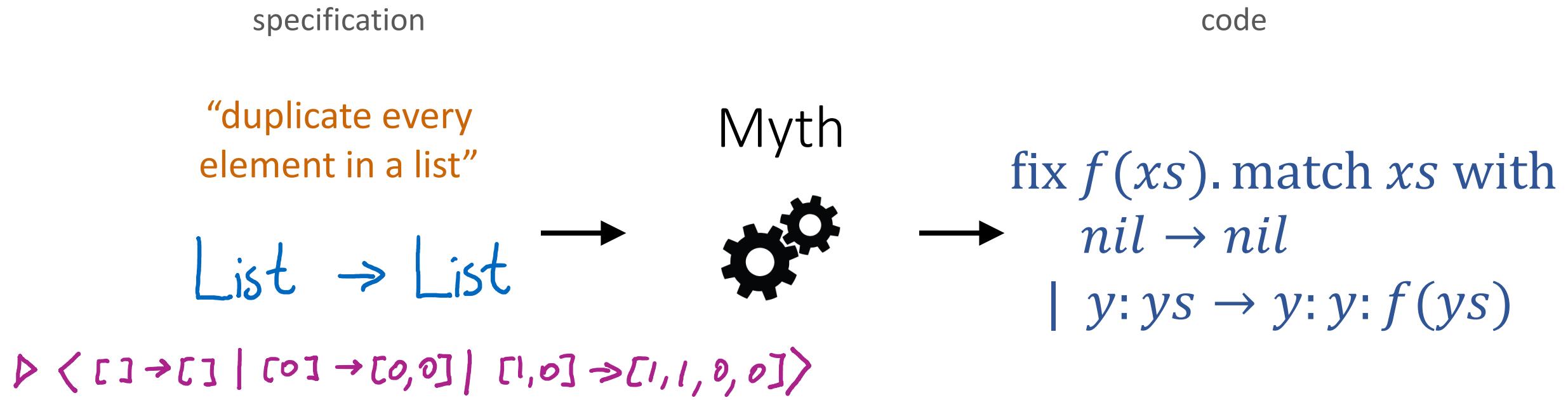
synthesis with types and examples

polymorphic types

refinement types

synthesis with refinement types

Type + examples

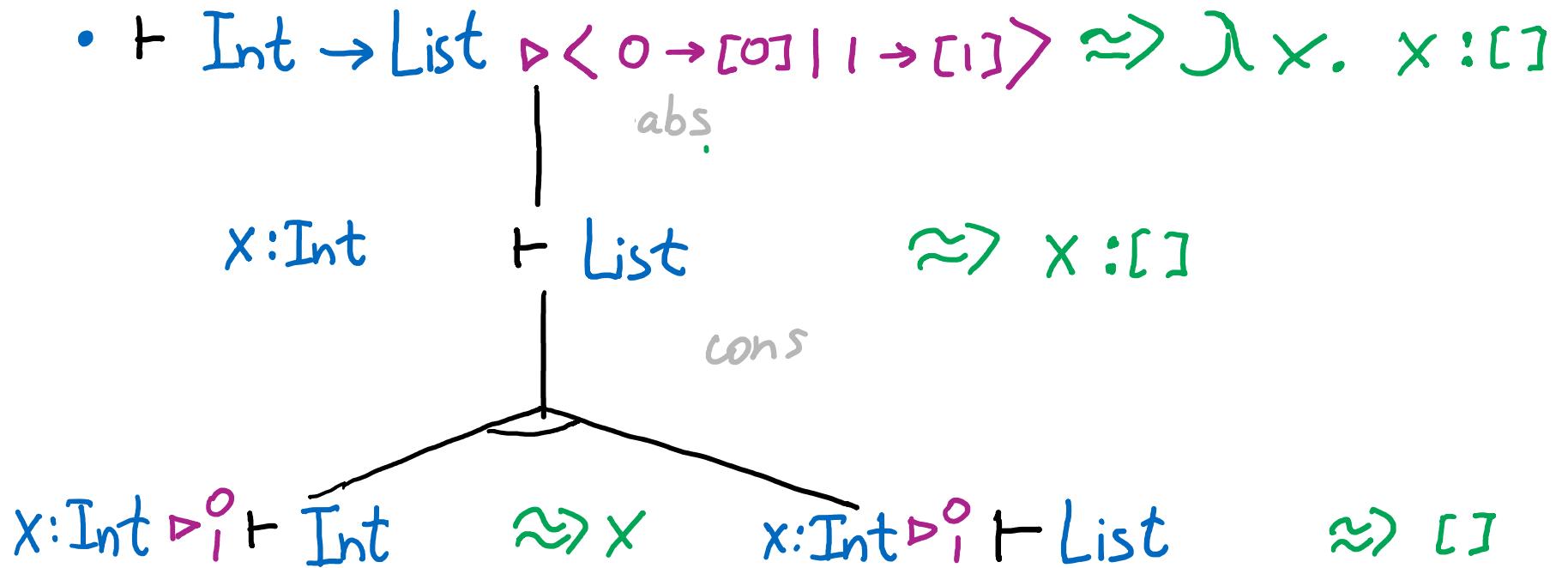


[Osera, Zdancewic , Type-and-Example-Directed Program Synthesis. 2015]

Types + examples: syntax

$V ::= 0 \mid v+ \mid [] \mid V:V \mid \overline{V \rightarrow V}$	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

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

Polymorphic types for synthesis

- $\vdash ? :: \text{Int} \rightarrow \text{List Int}$
- $\vdash ? :: \forall \alpha. \alpha \rightarrow \text{List } \alpha$

1. $\lambda x. \text{nil}$
2. $\lambda x. [0], \lambda x. [1], \dots$
3. $\lambda x. [x]$
4. $\lambda x. [\text{double } 0], \lambda x. [\text{dec } 0]$
5. $\lambda x. [0,0], \lambda x. [0,1], \dots$
6. $\lambda x. [x, x]$

which of these programs
match the polymorphic type?

Polymorphic types for synthesis

- $\vdash ? :: \text{Int} \rightarrow \text{List Int}$
 - $\vdash ? :: \forall \alpha. \alpha \rightarrow \text{List } \alpha$
-
1. $\lambda x. \text{nil}$
 2. $\lambda x. [0], \lambda x. [1], \dots$
 3. $\lambda x. [x]$
 4. $\lambda x. [\text{double 0}], \lambda x. [\text{dec 0}]$
 5. $\lambda x. [0,0], \lambda x. [0,1], \dots$
 6. $\lambda x. [x, x]$
-
- eliminate ambiguity!
- prune the search!

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)

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

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

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

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

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

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

This week

intro to type systems

enumerating well-typed terms

bidirectional type systems

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

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 ≤ v }
                           → SList α
```

example: insert an element into a sorted list

```
insert :: ??
```

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

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

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

sub-base
$$\frac{\llbracket \Gamma \rrbracket \wedge \phi' \Rightarrow \phi}{\Gamma \vdash \{\nu: B \mid \phi'\} <: \{\nu: B \mid \phi\}}$$

Pos <: Nat 

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

- Int → Int <: Int → Nat 
- Int → Int <: Nat → Int 
- x:Int → {Int | ν = x + 1} <: Nat → Nat 

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

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

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

ie

$$\frac{i\text{-abs}}{\Gamma \vdash \lambda x. \text{inc } x \Leftarrow \text{Nat} \rightarrow \text{Nat}}$$

subtyping constraints

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

implications

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

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

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

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

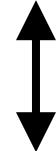
```
match xs with  
  Nil → Cons x Nil  
  Cons h t →  
    if x ≤ h  
      then Cons x xs  
      else Cons h (insert x t)
```

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}}
```

context:
x: a
xs: SList a



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

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}}
```

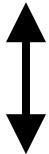


```
insert x xs =
  match xs with
    Nil → ???
    Cons h t → ???
```

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!

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

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

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

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

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

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


```
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≤v}

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

SMT solver: INVALID!

```
insert :: x: $\tau$  →  
      xs:SList  $\tau$  →  
      SList  $\tau$ 
```

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

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