#23: 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

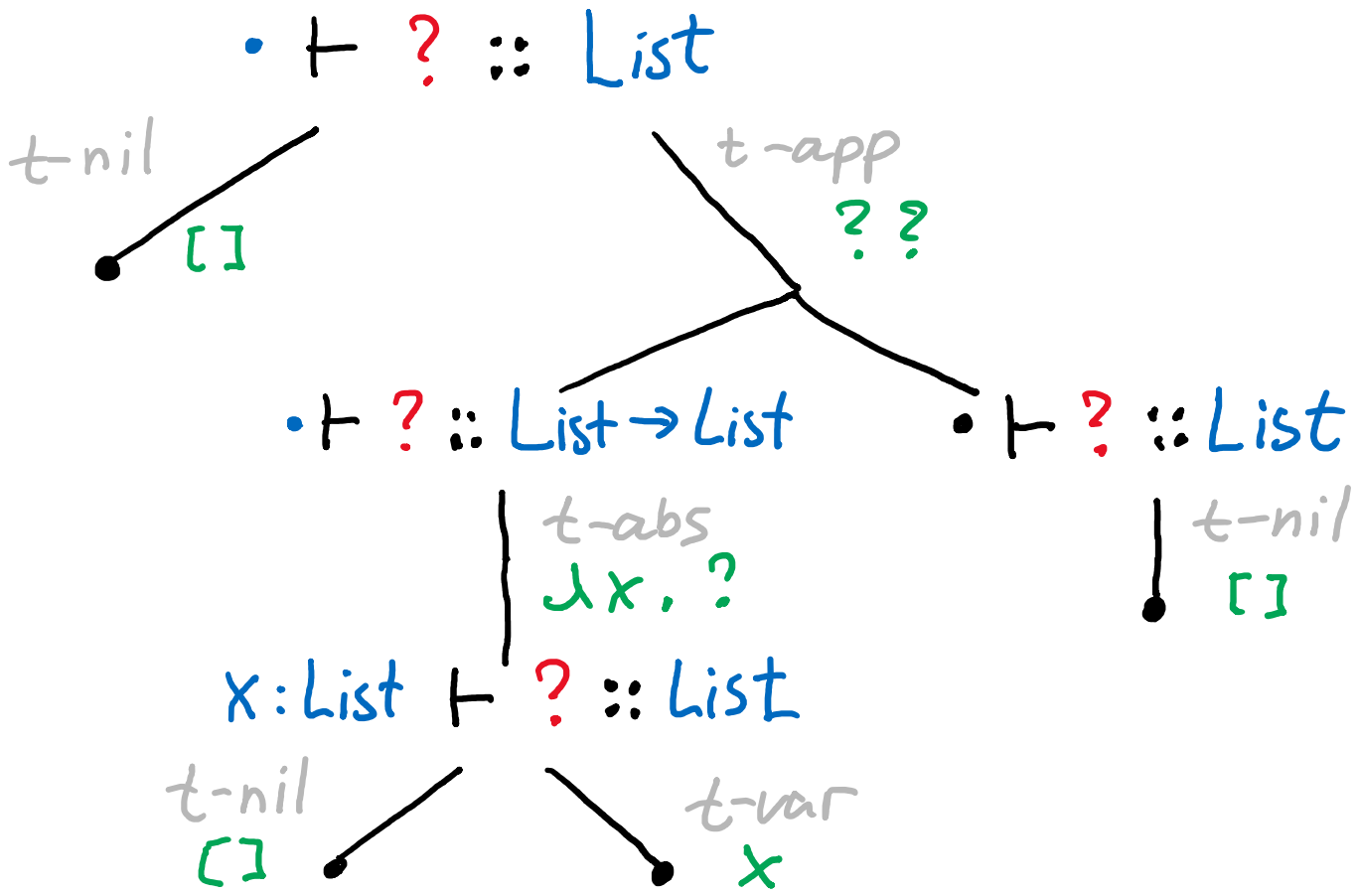
refinement types

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

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

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

introduction forms

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

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]

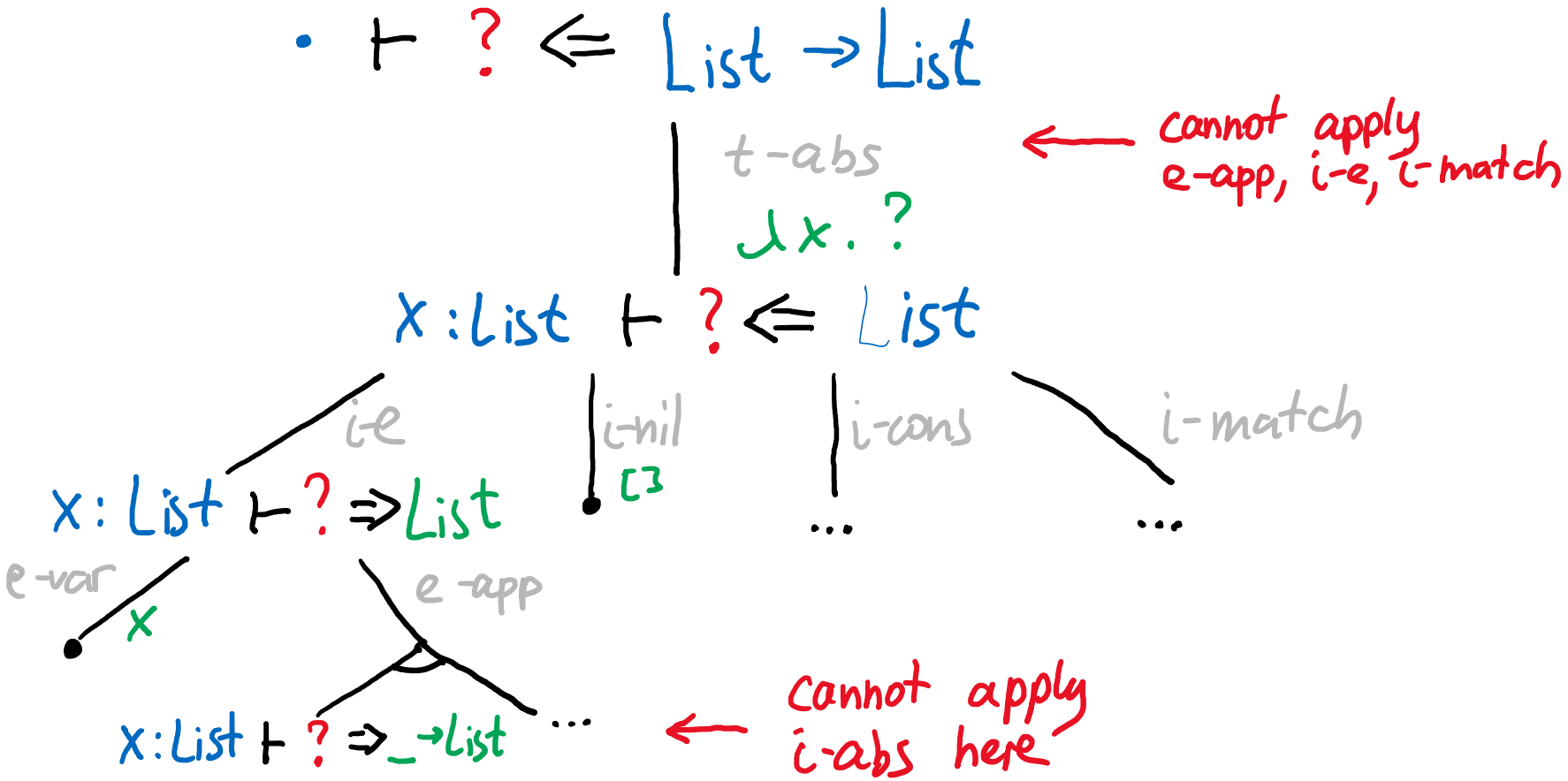
Bidirectional typing rules

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

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

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

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

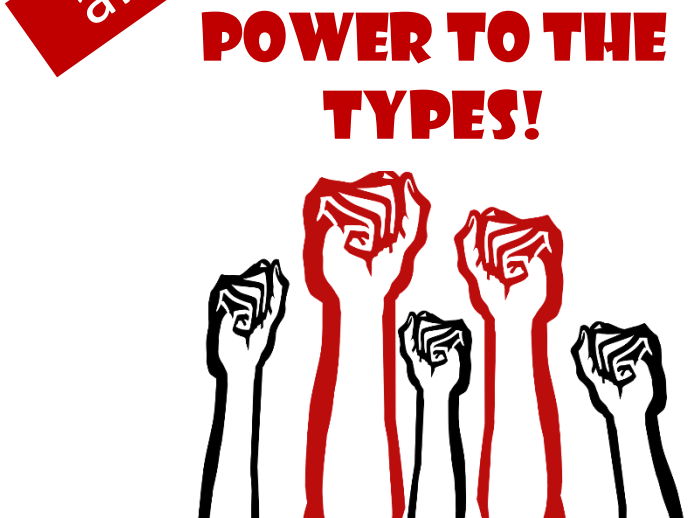
specification

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

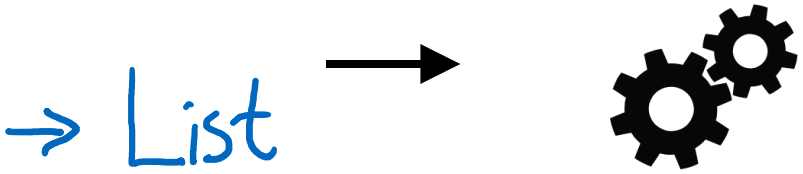
Type + examples

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

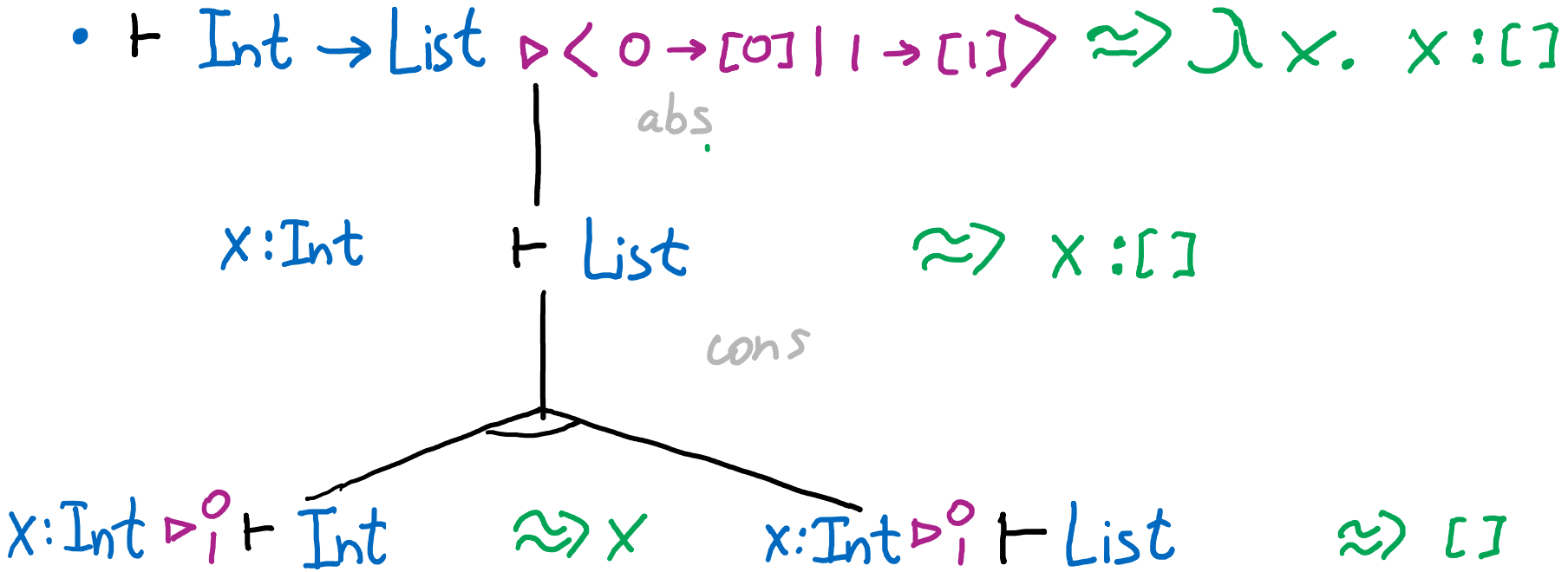
$$R ::= T \triangleright X$$

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

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

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

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

which of these programs
match the polymorphic type?

Polymorphic types for synthesis

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

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

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

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 ν = 0 }
    Cons  ::  x: α  → xs: List α
                    → { List α | len ν = 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 $$\frac{[\![\Gamma]\!] \wedge \phi' \Rightarrow \phi}{\Gamma \vdash \{\nu: B \mid \phi'\} <: \{\nu: B \mid \phi\}}$$

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

$\text{Pos} <: \text{Nat}$ ✓

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

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

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

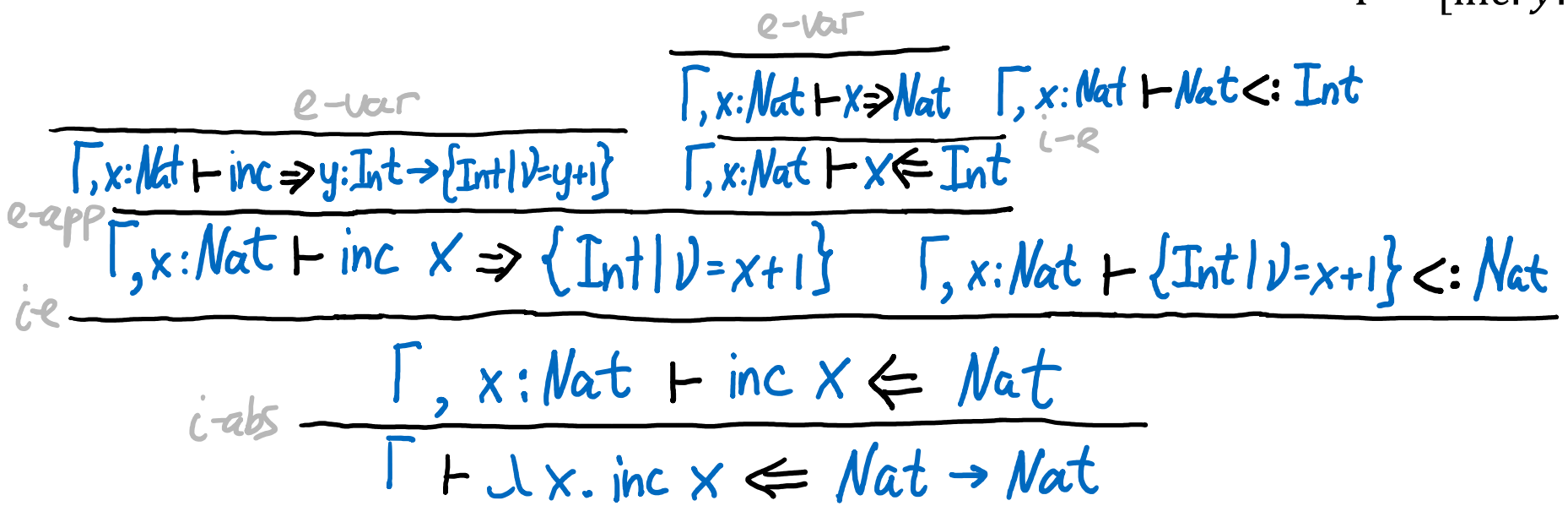
Typing rules

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

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

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 →
  {ν:List a | len ν =
    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

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

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

context:
x: a
xs: SList a

Type-directed enumeration

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

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

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

Type-directed enumeration

```
{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:
∀x: {} = {} ∪ {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)

$\Gamma \vdash i \Leftarrow S$

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

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

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

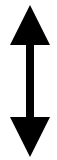
$$\text{e-app}\ \frac{\Gamma \vdash e \Leftarrow \bot \to T \Rightarrow y : T_1 \to T_2 \qquad \Gamma \vdash i \Leftarrow T_1}{\Gamma \vdash e\ i \Leftarrow T \Rightarrow [y \mapsto i]\, T_2}$$

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

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

```
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
```