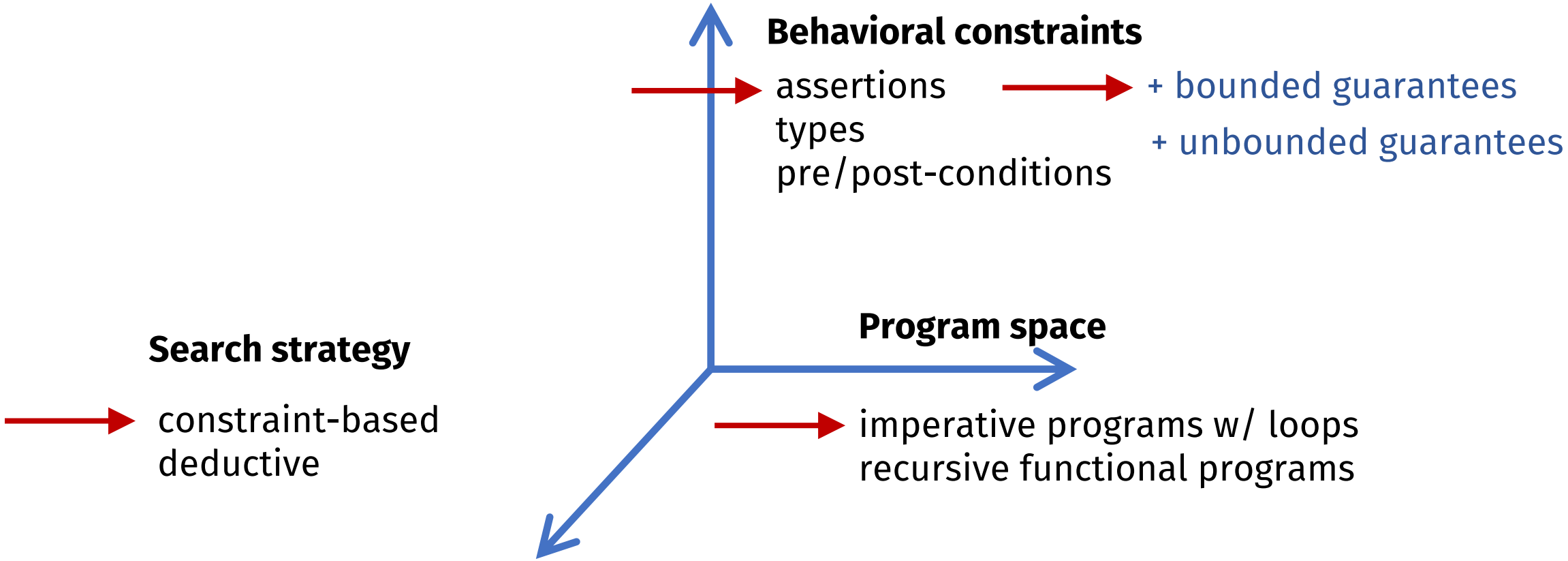
#21: Type-directed Synthesis

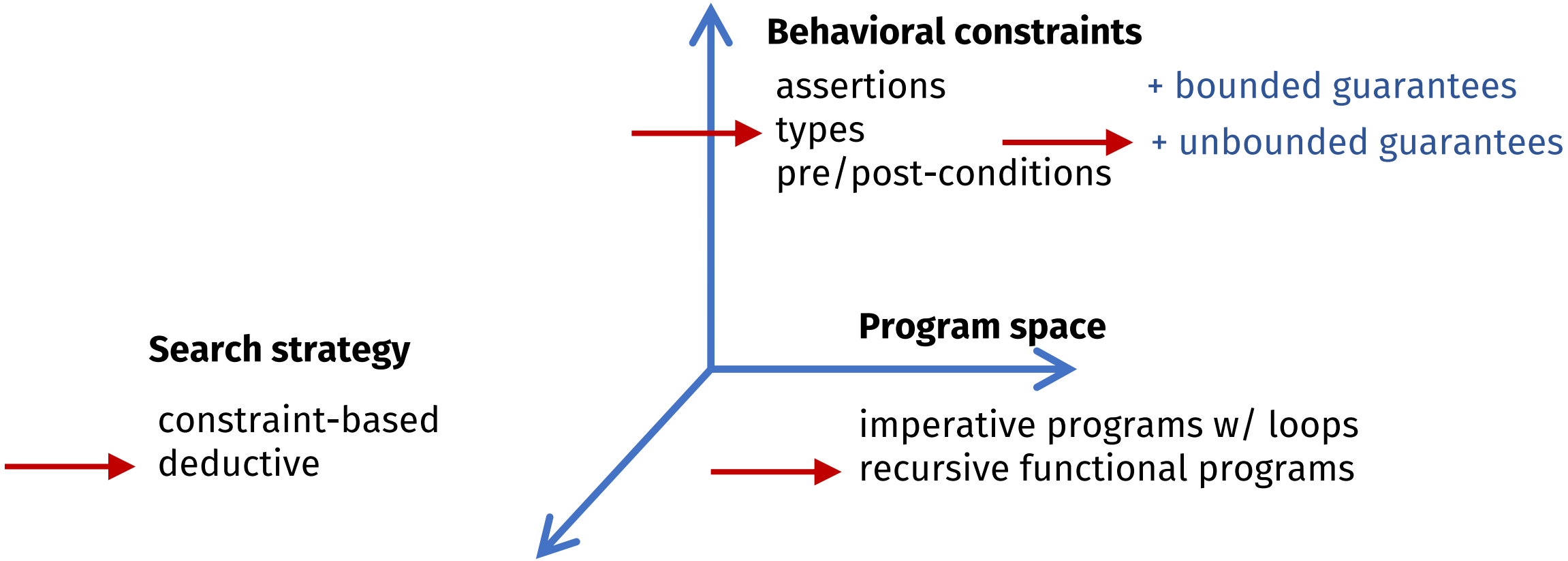
Sankha Narayan Guria

EECS 700: Introduction to Program Synthesis

Last week


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

This week

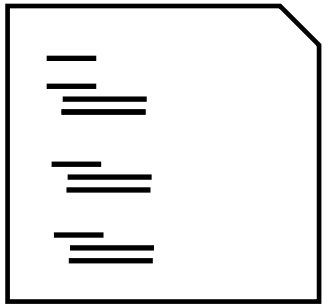

Behavioral constraints
assertions
types
pre/post-conditions
+ bounded guarantees
+ unbounded guarantees
Program space
imperative programs w/ loops
recursive functional programs
Search strategy
constraint-based
deductive

Type-driven program synthesis

specification

search

program



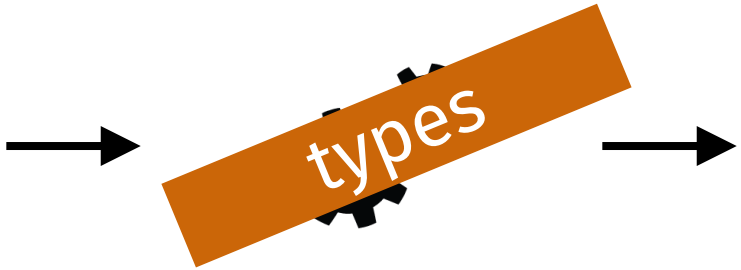
programmer-friendly
informative

Type-driven program synthesis

specification

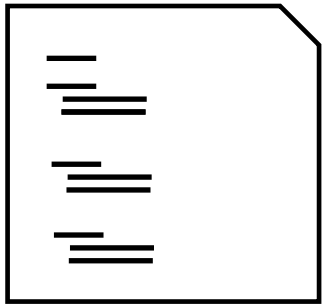
programmer-friendly
informative

search



help prune
the search space

program



Which program do I have in mind?

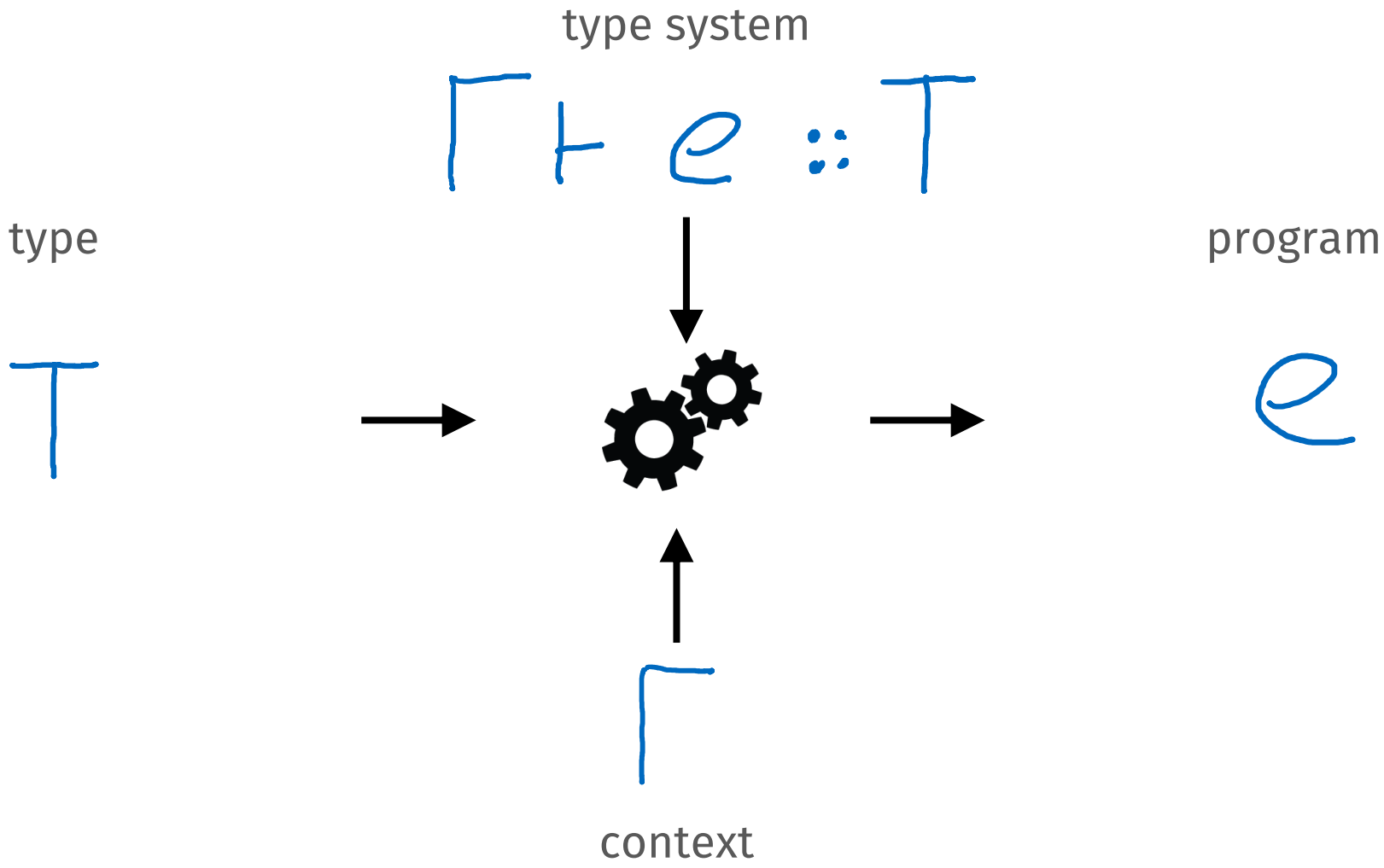
```
Char -> String -> [String]
```

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

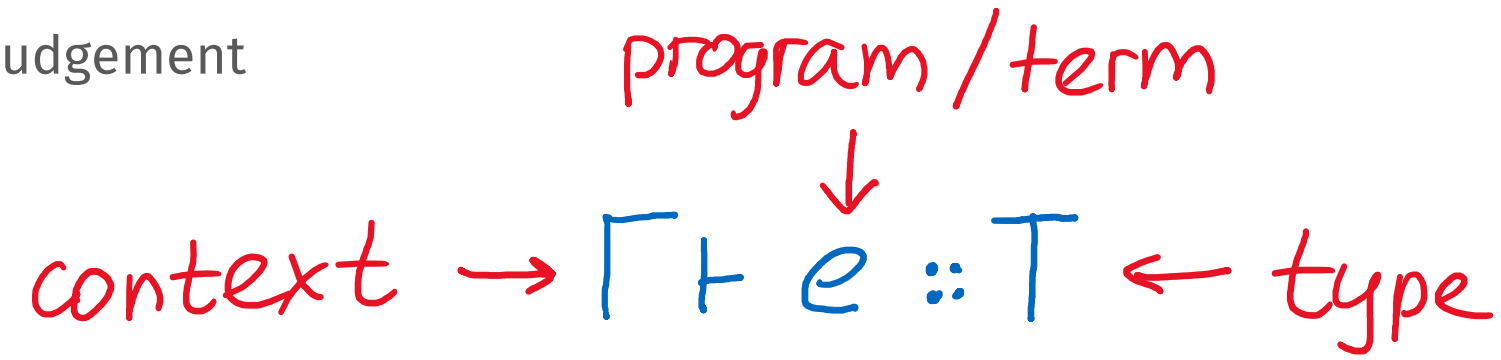
synthesis with refinement types

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

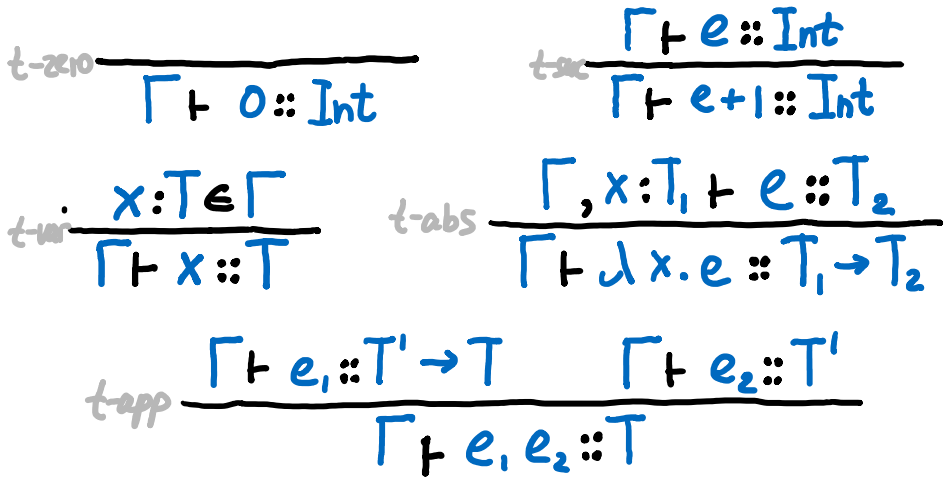
Typing derivations

let's build a derivation of

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

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

Typing derivations



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

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!

$e ::= \ldots \mid [\,] \mid e : e \mid$ match e with $[\,] \to e \mid x : x \to e$

$T ::= \text{Int} \mid \text{List} \mid T \to T$

Example program: head with default

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

Typing rules

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

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

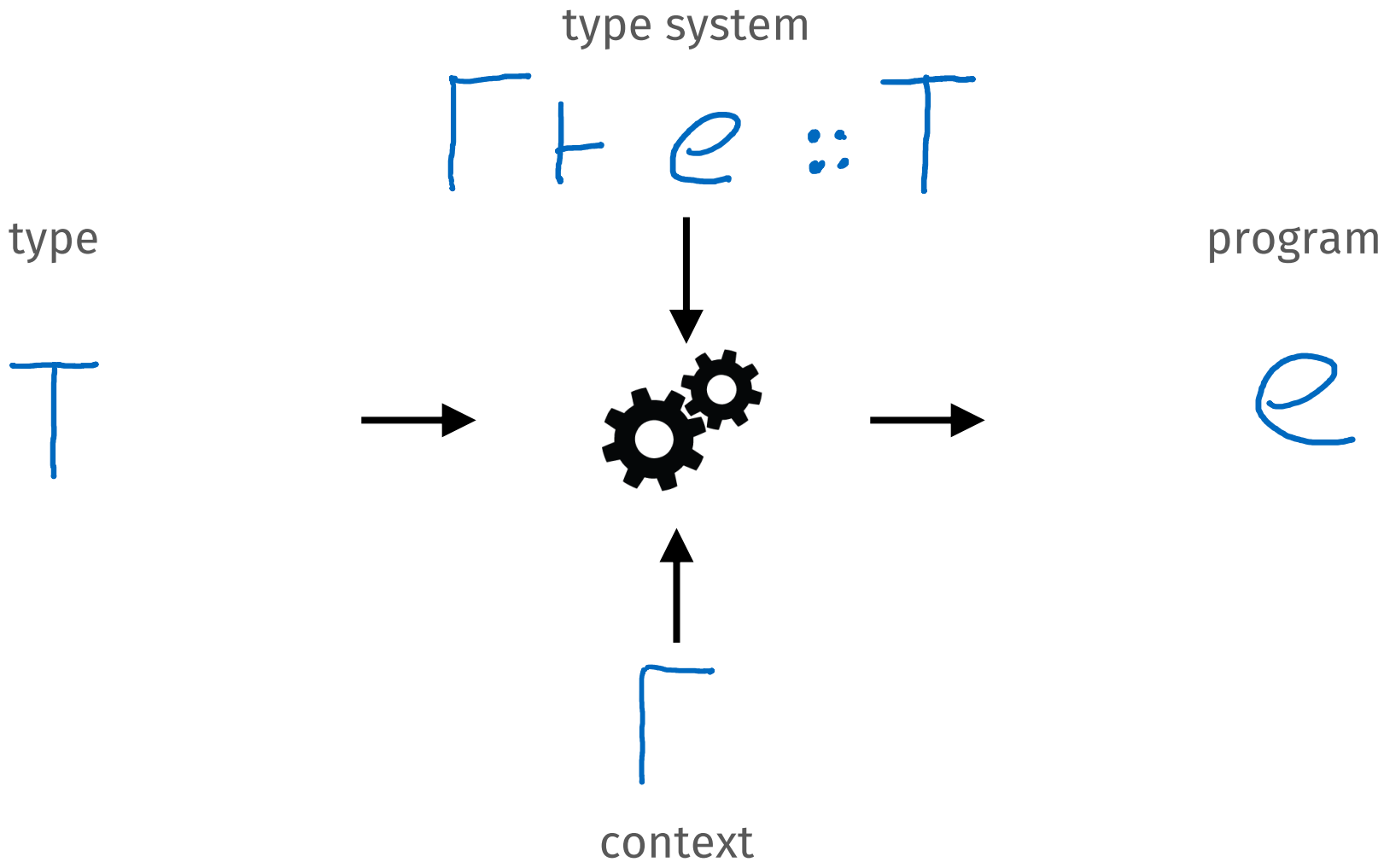
$$\text{t-nil}\ \frac{}{\Gamma \vdash [] :: \text{List}} \qquad \text{t-cons}\ \frac{\Gamma \vdash e_1 :: \text{Int} \quad \Gamma \vdash e_2 :: \text{List}}{\Gamma \vdash e_1 : e_2 :: \text{List}}$$

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

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



Enumerating well-typed terms

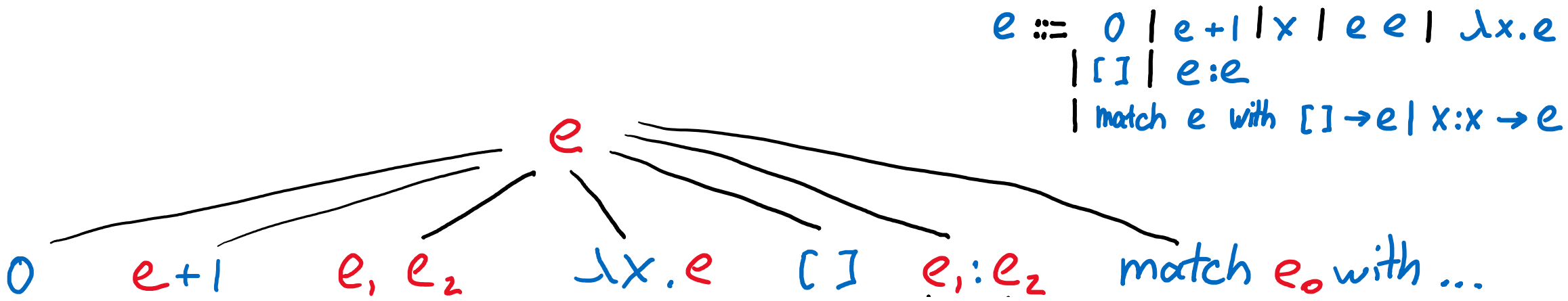
how should I enumerate all terms of type List → List?

(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 List → 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

term and type are known: type checking

term known, type unknown: type inference

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: 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

only 2 programs fully constructed!

all other programs *rejected early*

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

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

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

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