#22: Brahma and Type-directed synthesis

Sankha Narayan Guria

EECS 700: Introduction to Program Synthesis

Brahma

- **Idea:** encode the space of loop-free (bit-vector) programs as an SMT constraint

Brahma encoding: take 1

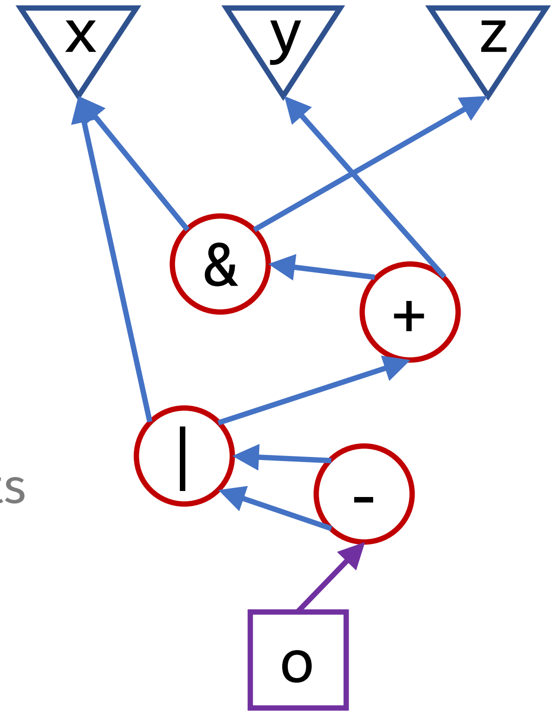
program = DAG

I

inputs

F

components

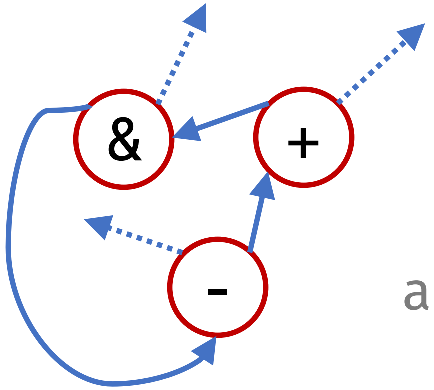


$V = I \cup F$ nodes

parameter space

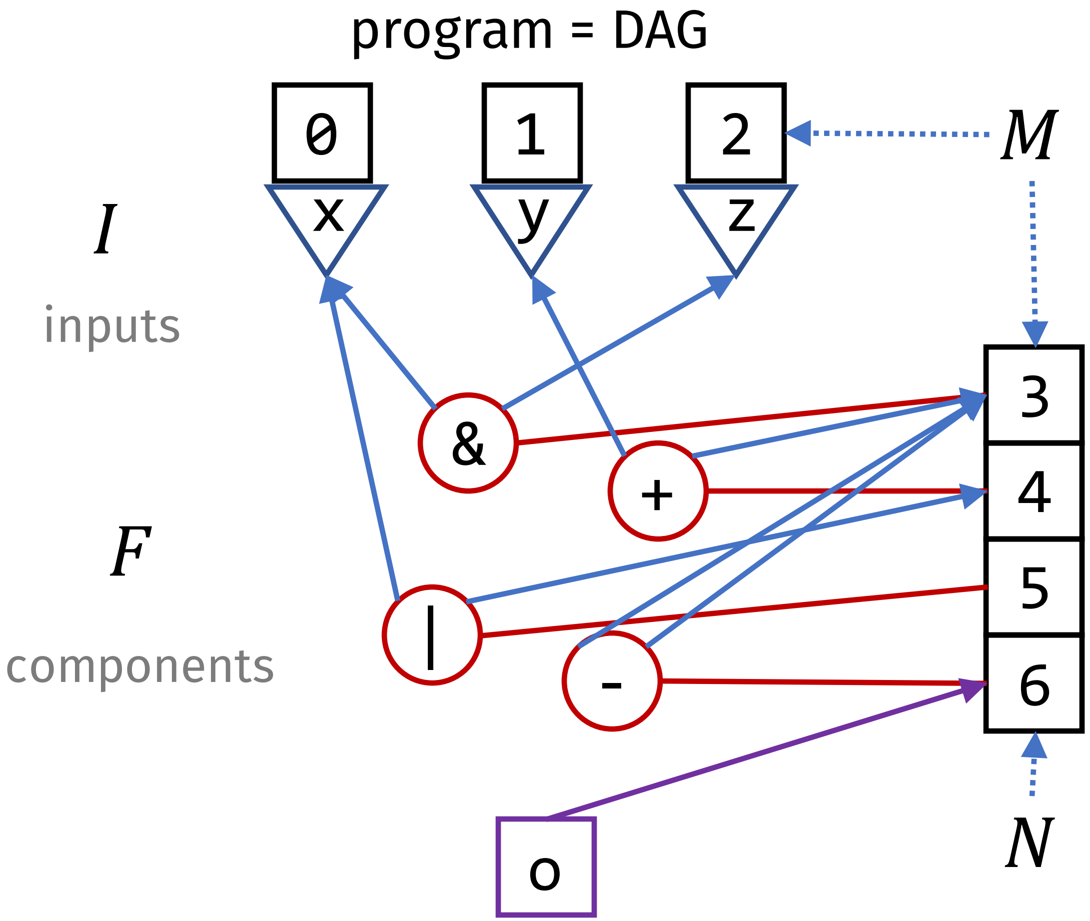
$$C = \{c_o : V\} \cup \bigcup_{f \in F} \{c_1^f, c_2^f : V\}$$

$\mathrm{wf}(C) \equiv ?$

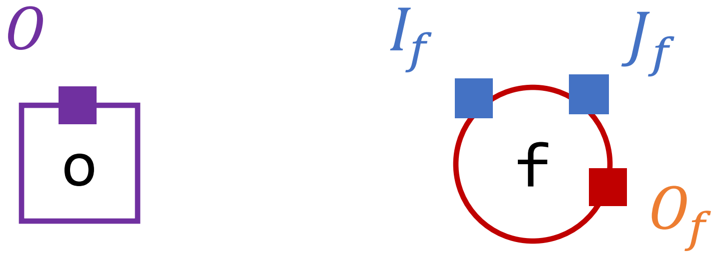


hard to encode acyclicity!

Brahma encoding: take 2



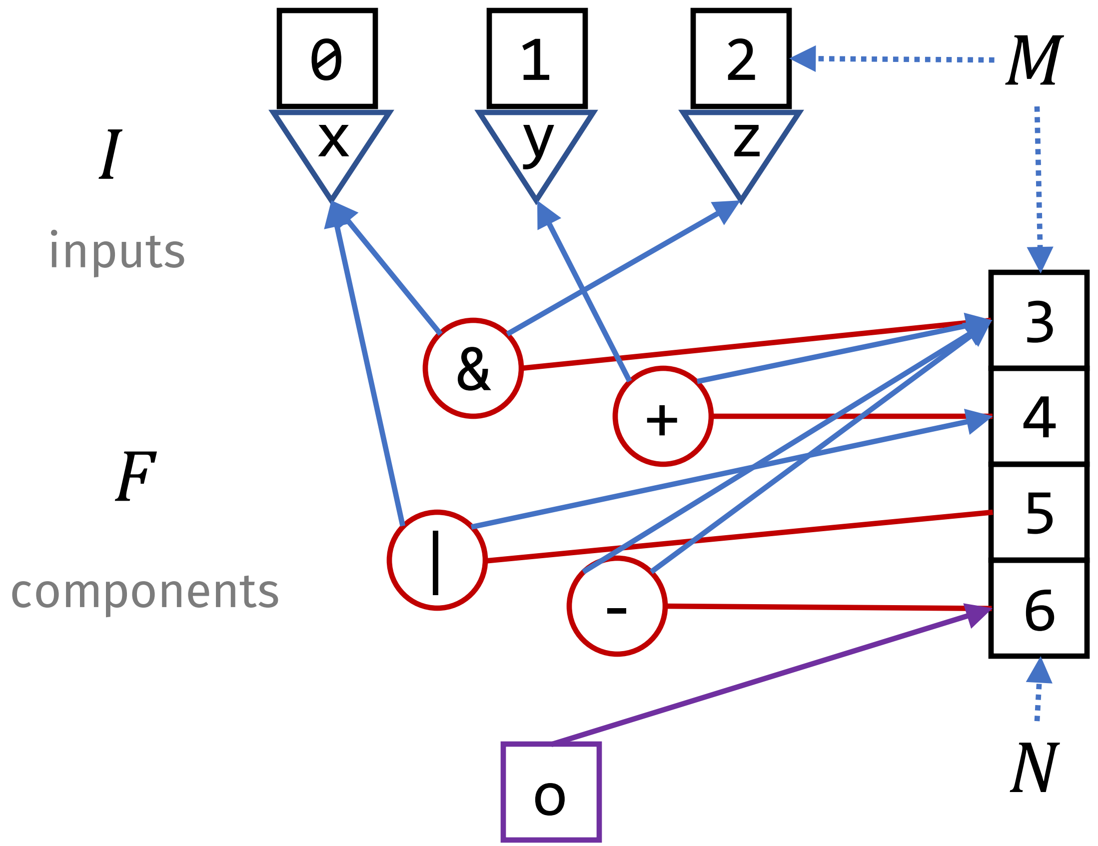
$$C = \{c_O : \mathrm{Int}\} \cup \overset{\text{parameter space}}{\bigcup_{f \in F}} \{c_{O_f}, c_{I_f}, c_{J_f} : \mathrm{Int}\}$$



$$\mathrm{wf}(C) \equiv c_O \in M \ \wedge \bigwedge_{f \in F} c_{O_f} \in N \wedge c_{I/J_f} \in M$$

Brahma encoding: take 2

program = DAG



parameter space

$$C = \{c_o : \mathrm{Int}\} \cup \bigcup_{f \in F} \{c_{O_f}, c_{I_f}, c_{J_f} : \mathrm{Int}\}$$

$$T = \bigcup_{f \in F} \{I_f, J_f. O_f\}$$

$$\varphi(C, I, O) \equiv \exists T. \bigwedge_{f \in F} O_f = F(I_f, J_f)$$

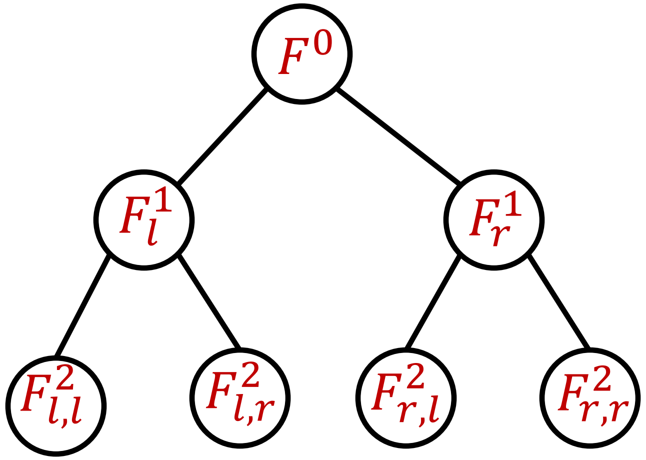
$$\wedge \bigwedge_{x,y \in T \cup I \cup \{O\}} c_x = c_y \Rightarrow x = y$$

Brahma: contributions

- SMT encoding of program space
  - sound? complete? solver-friendly?
  - more compact than alternatives*
- SMT solver can guess constants
  - e.g. 0x55555555 in P23

Alternative encodings

Tree encoding



Linear encoding

$$t_0 = F_0(t_{I0}, t_{J0})$$
$$t_1 = F_1(t_{I1}, t_{J1})$$
$$\ldots$$
$$t_N = F_N(t_{IN}, t_{JN})$$

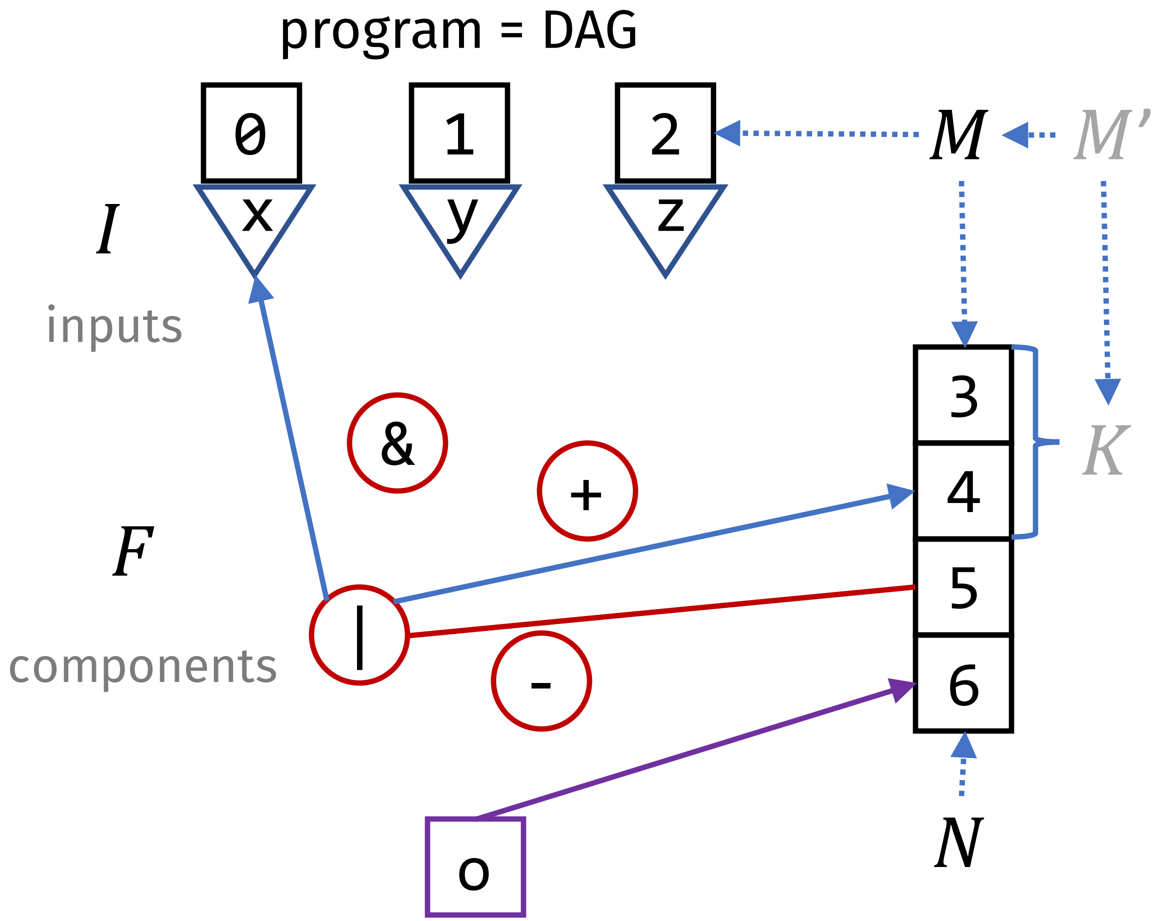
Brahma: limitations

- Requires component multiplicities
  - If we didn’t have multiplicities, where would their encoding break? How could we fix it?
  - What happens if user provides too many? too few?
  - What’s the alternative to including dead code?
- Requires *precise* SMT specs for components
  - What happens if we give an over-approximate spec?
- No loops, no types, no ranking

Brahma: questions

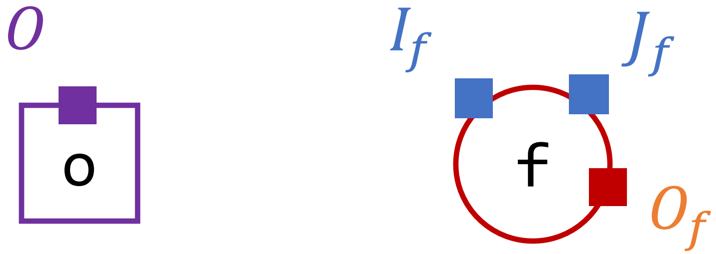
- Behavioral Constraints? Structural Constraints? Search Strategy?
  - First-order formula
  - A multiset of components + straight-line program
  - Constraint based + CEGIS
- Can we represent these structural constraints as a grammar?
  - Yes and no
  - No because grammars cannot encode multiplicities
    - also: you can have let-bindings in SyGuS but CFG cannot encode well-formedness
  - Yes, because the set is finite, so we can simply enumerate all possible programs
    - but this is not useful for synthesis

Limit #components to K?



parameter space

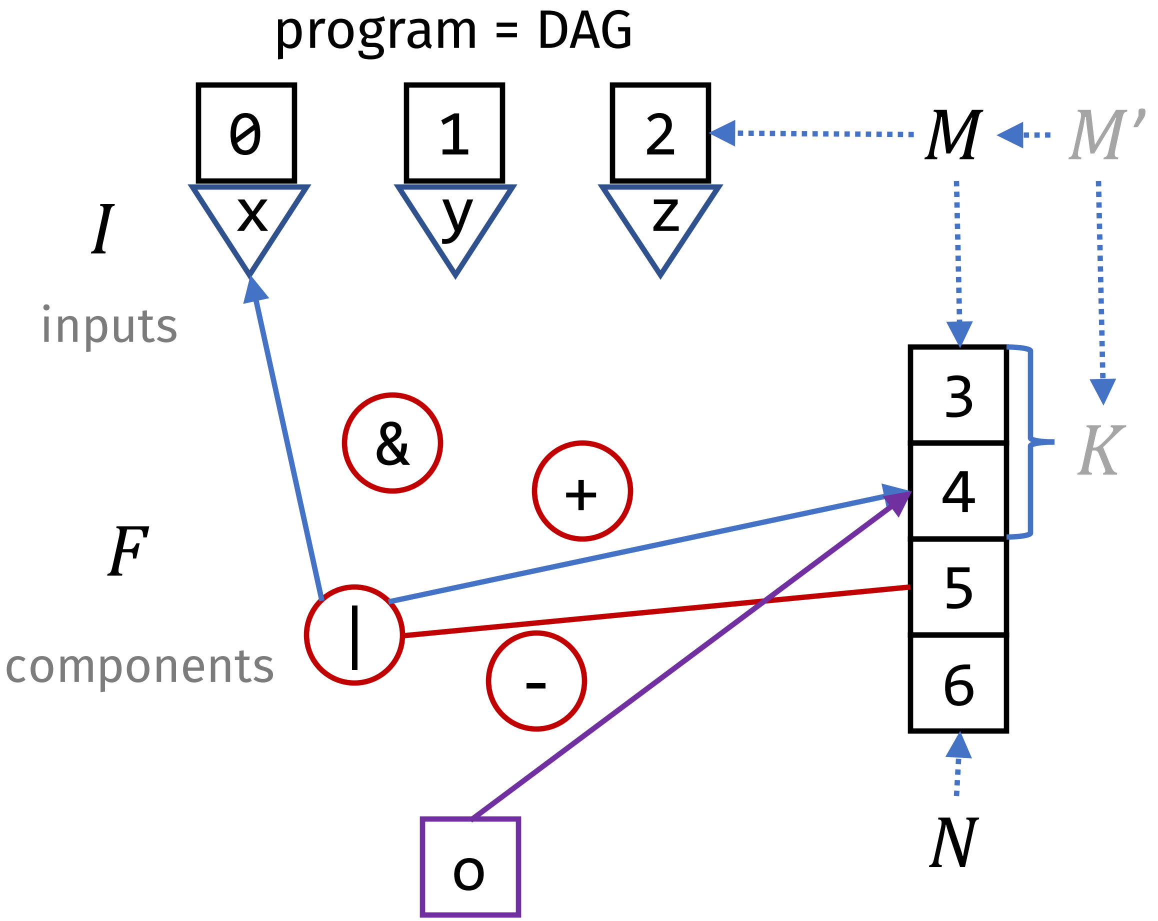
$$C = \{c_O : \text{Int}\} \cup \bigcup_{f \in F} \{c_{O_f}, c_{I_f}, c_{J_f} : \text{Int}\}$$



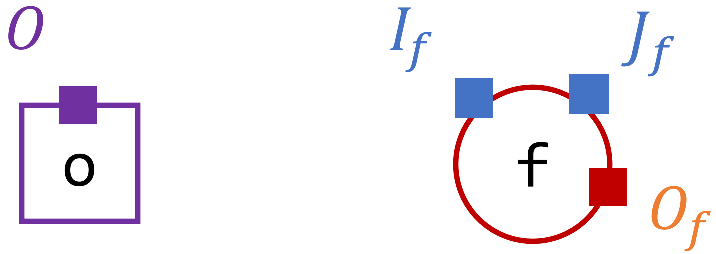
$$\text{wf}(C) \equiv c_O \in \overset{M'}{\cancel{M}} \wedge \bigwedge_{f \in F} c_{O_f} \in \overset{K}{\cancel{N}} \wedge c_{I/J_f} \in \overset{M'}{\cancel{M}}$$

$$\wedge \bigwedge_{f,g \in F, f \not\equiv g} c_{O_f} \neq c_{O_g} \wedge \bigwedge_{f \in F} c_{I/J_f} < c_{O_f}$$

Limit #components to K?



$$C = \{c_o : \mathrm{Int}\} \cup \overbrace{\bigcup_{f \in F} \{c_{O_f}, c_{I_f}, c_{J_f} : \mathrm{Int}\}}^{\text{parameter space}}$$



$$\mathrm{wf}(C) \equiv c_O \in \overset{M'}{\cancel{M}} \wedge \bigwedge_{f \in F} c_{O_f} \in N \wedge c_{I/J_f} \in M$$

$$\wedge \bigwedge_{f,g \in F, f \not\equiv g} c_{O_f} \neq c_{O_g} \wedge \bigwedge_{f \in F} c_{I/J_f} < c_{O_f}$$

On to today’s topic …

Typing rules

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

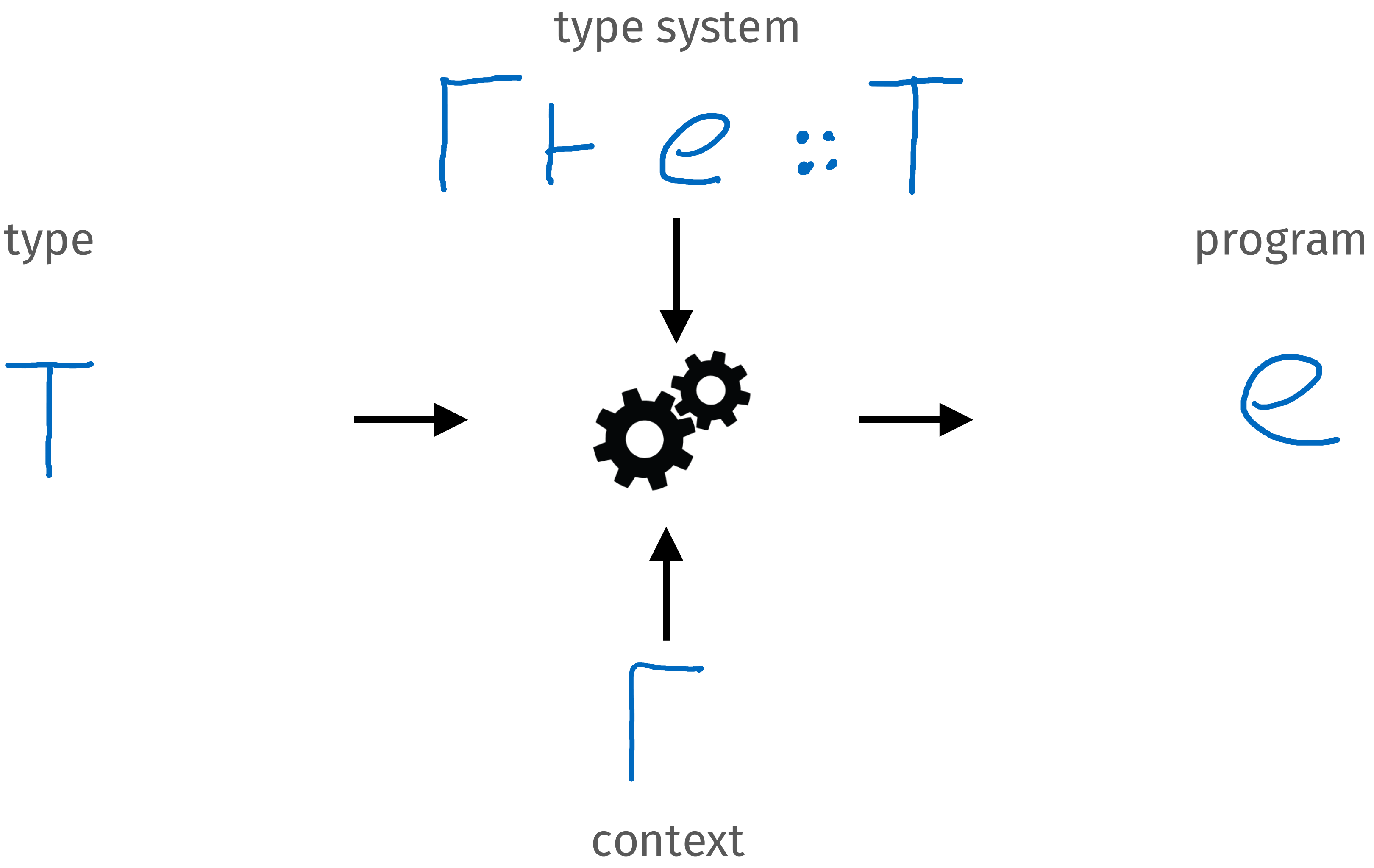
$$\text{t-cons}\ \frac{\Gamma \vdash e_1 :: \mathit{Int} \qquad \Gamma \vdash e_2 :: \mathit{List}}{\Gamma \vdash e_1 : e_2 :: \mathit{List}}$$

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

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

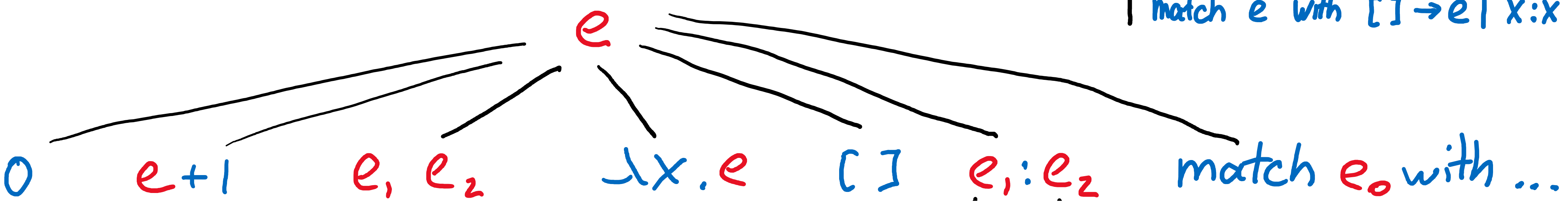
how should I enumerate all terms of type List $\to$ List?
(up to depth 2, in the empty context)

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

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



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 $\rightarrow$ 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

$\Gamma \vdash e :: T$ term known, type unknown: type inference

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

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

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

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

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