Type-and-Example-Directed Program Synthesis

Steve Zdancewic
University *of* Pennsylvania

TFP 2016

ExCAPE
Expeditions in Computer Augmented Program Engineering

Joint With

Peter-Michael Osera

GRINNELL COLLEGE

Jonathan Frankle

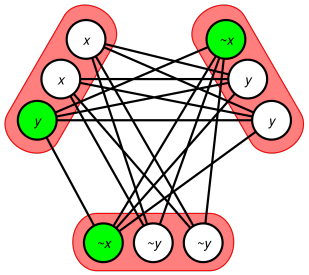
Georgetown University

David Walker

PRINCETON UNIVERSITY

Program Synthesis

Program Search + *Specification*

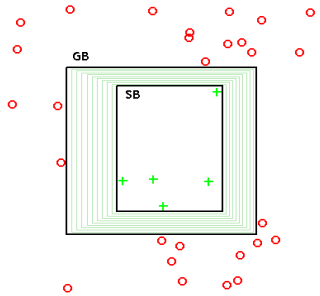


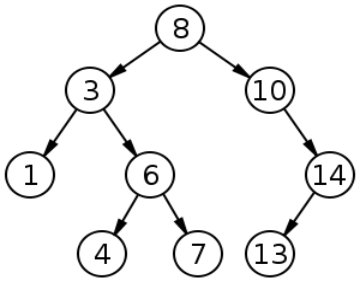
Formal methods: SMT solvers, model checking, CEGIS

[Solar-Lezama; Kneuss; Bodik, Torlak; Gulwani; ...]

Machine Learning: inductive learning, genetic prog.

[Lau; Weimer; Seshia; ...]





AI/Logic: syntactic enumerative (SyGus) / proof theory

[Green (1969), Summers (1976), Waldinger & Manna (1980) … Kitzelmann; Albarghouthi, Kincaid; Kuncak, Piscak, ...]

This talk

Type-and-Example-Directed Program synthesis

[PLDI 2015: Osera & Zdancewic]

Example-Directed Synthesis: A Type-Theoretic Interpretation

[POPL 2016: Frankle, Osera, Walker, & Zdancewic]

Types:
What are they good for?

Program Design

"My program writes itself!"
(a.k.a. type-directed programming)

```
t1 -> t2  ~~>  let f (x:t1) : t2 = ■
```

```
C  ~~>  (g . f) ■
(f : A -> B, g : B -> C)
```

How can we mechanize this reasoning?

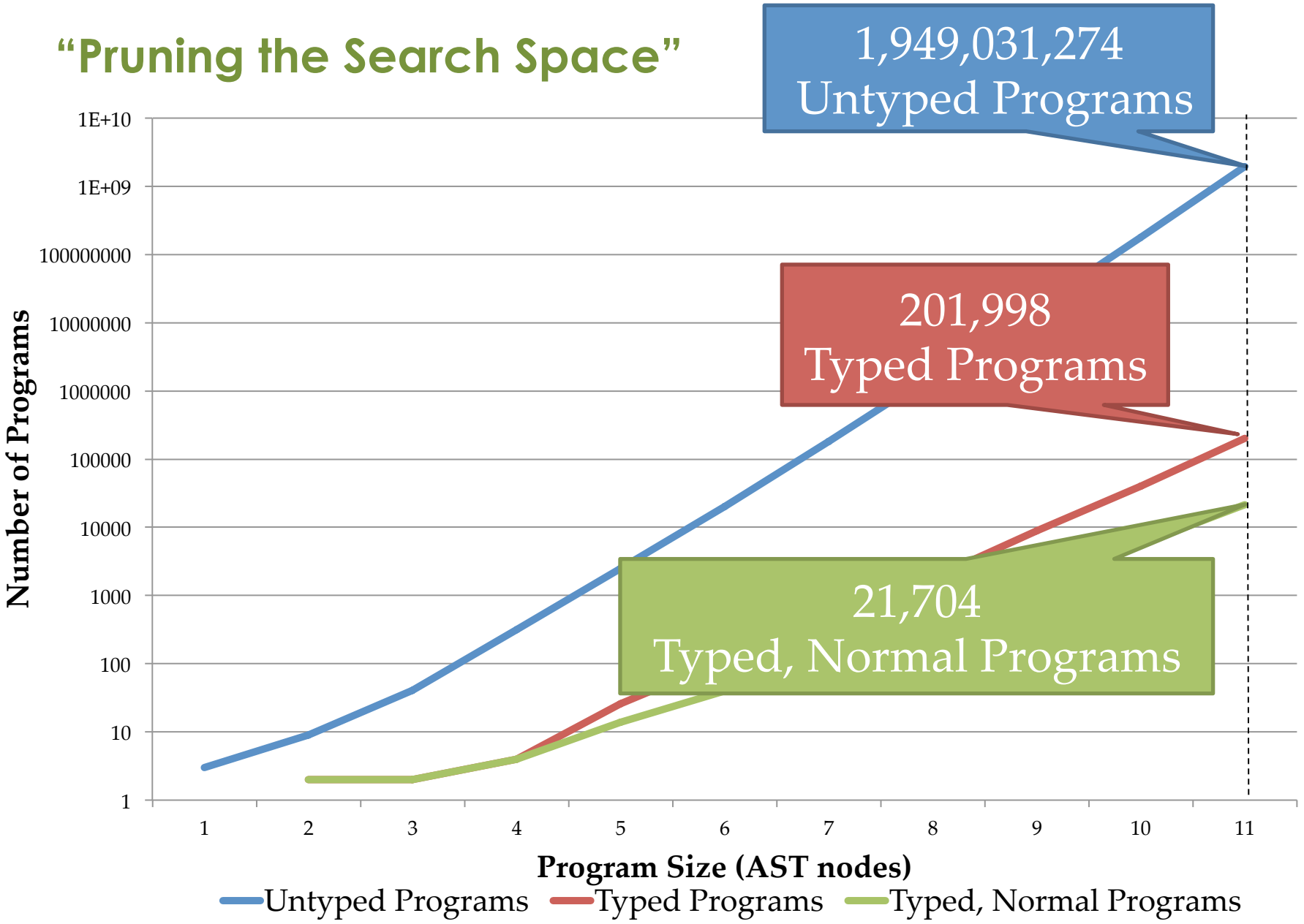
λ **Type theoretic** foundations

1. Synthesize programs with *higher-order functions*, *recursion*, and *algebraic datatypes*.

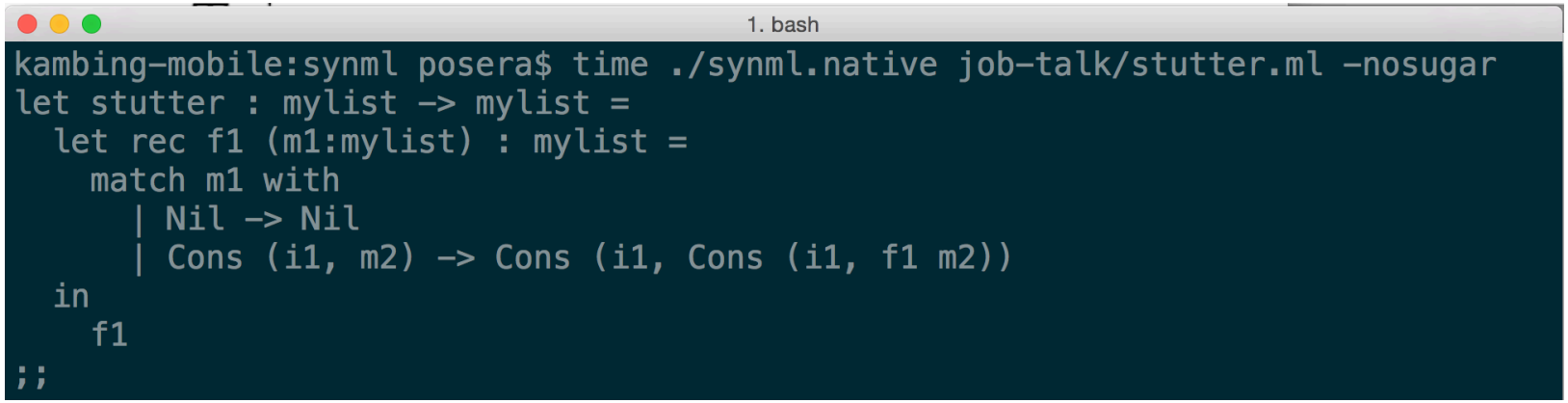
2. Type structure prunes the search space.

3. Take advantage of techniques from proof theory literature

"Pruning the Search Space"



Myth, a program synthesizer for typed, functional programs (OCaml).



(**MYTH** DEMO)

(One type per expression)

Inputs | Output*

$$\Gamma \vdash e : \tau$$

⇒ Typechecking specification

Simply-typed Lambda Calculus

$$
\begin{array}{lcll}
\tau & ::= & \tau_1 \to \tau_2 \mid T & \text{Types} \\
e & ::= & x \mid e_1\, e_2 \mid \lambda x{:}\tau.\, e \mid c & \text{Terms}
\end{array}
$$

T-VAR

$$\frac{x \in \Gamma}{\Gamma \vdash x : \tau}$$

T-LAM

$$\frac{x{:}\tau_1, \Gamma \vdash e : \tau_2}{\Gamma \vdash \lambda x{:}\tau_1.\, e : \tau_1 \to \tau_2}$$

T-APP

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\, e_2 : \tau_2}$$

T-BASE

$$\frac{}{\Gamma \vdash c : T}$$

Type Checking / Inference

$$\dfrac{\dfrac{g : B \to C \in \Gamma}{\Gamma \vdash g : B \to C} \qquad \dfrac{\dfrac{f : A \to B \in \Gamma}{\Gamma \vdash f : A \to B} \qquad \dfrac{x : A \in \Gamma}{\Gamma \vdash x : A}}{\Gamma \vdash f\ x : B}}{\overbrace{f{:}A \to B\ ,\ g{:}B \to C,\ x{:}A}^{\Gamma} \vdash g\ (f\ x) : C}$$

$$\dfrac{\Gamma \vdash g\ (f\ x) : C}{f{:}A \to B\ ,\ g{:}B \to C \vdash \lambda x{:}A.\, g\ (f\ x) : A \to C}$$

$$\dfrac{f{:}A \to B\ ,\ g{:}B \to C \vdash \lambda x{:}A.\, g\ (f\ x) : A \to C}{f{:}A \to B \vdash \lambda g{:}B \to C.\ \lambda x{:}A.\, g\ (f\ x) : (B \to C) \to A \to C}$$

$$\dfrac{f{:}A \to B \vdash \lambda g{:}B \to C.\ \lambda x{:}A.\, g\ (f\ x) : (B \to C) \to A \to C}{\vdash \lambda f{:}A \to B.\ \lambda g{:}B \to C.\ \lambda x{:}A.\, g\ (f\ x) : (A \to B) \to (B \to C) \to A \to C}$$

(Many expressions per type)

Inputs

Output

$$\Gamma \vdash \tau \rightsquigarrow e$$

⇒ Well-typed term *search* procedure

Derivation → Enumerated Program

Eliminating Redundancy: Normal Form Terms

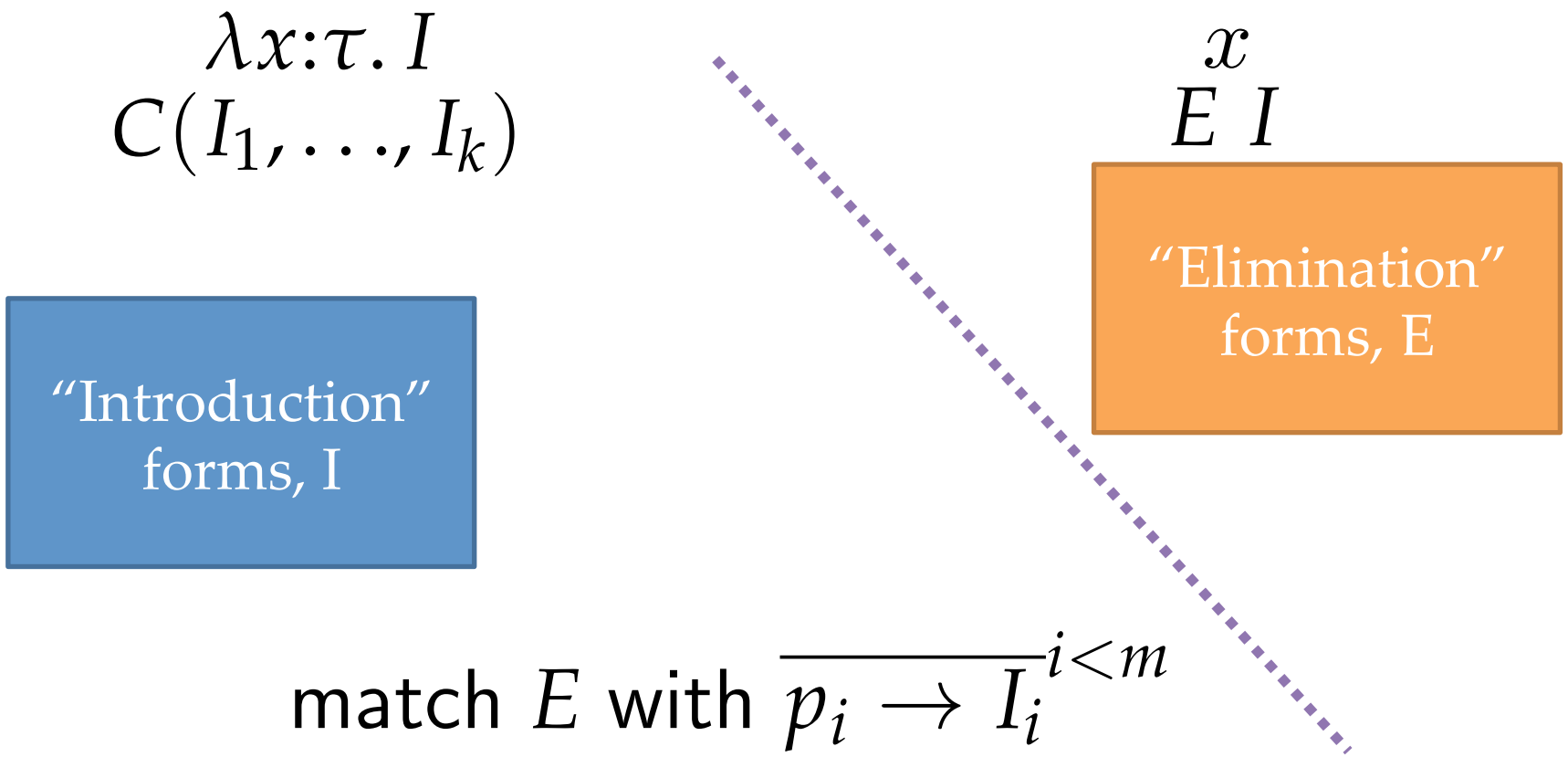
```
(fun x : nat -> x + 1)  3        ≡        4
```

```
match Nil with
  | Nil -> 4                     ≡        4
  | Cons (x,l) -> e
```

⇒ Avoid enumerating non-normal terms

$$\lambda x{:}\tau.\, e \qquad\qquad x$$
$$C(e_1, \ldots, e_k) \qquad\qquad e_1\, e_2$$

$$\text{match } e \text{ with } \overline{p_i \to e_i}^{\,i<m}$$


$\lambda x{:}\tau.\, I$
$C(I_1, \ldots, I_k)$
x
$E\ I$
"Introduction" forms, I
"Elimination" forms, E
match E with $\overline{p_i \rightarrow I_i}^{\,i<m}$

$$\lambda x{:}\tau.\, I$$
$$C(I_1, \ldots, I_k)$$

$$x$$
$$E\ I$$

(1) Type-directed **Refinement**

Type → Term Shape → Example Refinement.

$$\text{match } E \text{ with } \overline{p_i \rightarrow I_i}^{\,i<m}$$

$$\lambda x{:}\tau.\, I$$
$$C(I_1, \ldots, I_k)$$

$$\begin{array}{c} x \\ E\ I \end{array}$$

(1) Type-directed **Refinement** (2) **Guess**-and-check

Normalize-and-compare strategy.

match E with $\overline{\qquad\qquad}^{i<m}$ $p_i \rightarrow I_i$

$$\lambda x{:}\tau.\, I \qquad\qquad x$$
$$C(I_1, \ldots, I_k) \qquad\qquad E\ I$$

(1) Type-directed **Refinement** (2) **Guess**-and-check

Decompose data to learn more information.

match E with $\overline{p_i \rightarrow I_i}^{\,i<m}$

(3) **Learning** via pattern matching

$$\lambda x{:}\tau.\, I \qquad\qquad x$$
$$C(I_1, \ldots, I_k) \qquad\qquad E\ I$$

(1) Type-directed **Refinement** (2) **Guess**-and-check

Breadth-first search for valid derivations.

⇒

Synthesize the *smallest* satisfying program.

$$\text{match } E \text{ with } \overline{p_i \rightarrow I_i}^{\,i<m}$$

(3) **Learning** via pattern matching

Example values:
- Constructors
- Input/output pairs

$$\chi ::= C(\chi_1, \ldots, \chi_k) \mid \overline{v_i \Rightarrow \chi_i}^{\,i<n}$$

Inputs | Output

$$\Gamma \vdash \tau \triangleright X \rightsquigarrow e$$

⇒ Type-and-example *refinement* procedure

Derivation → Satisfying Program

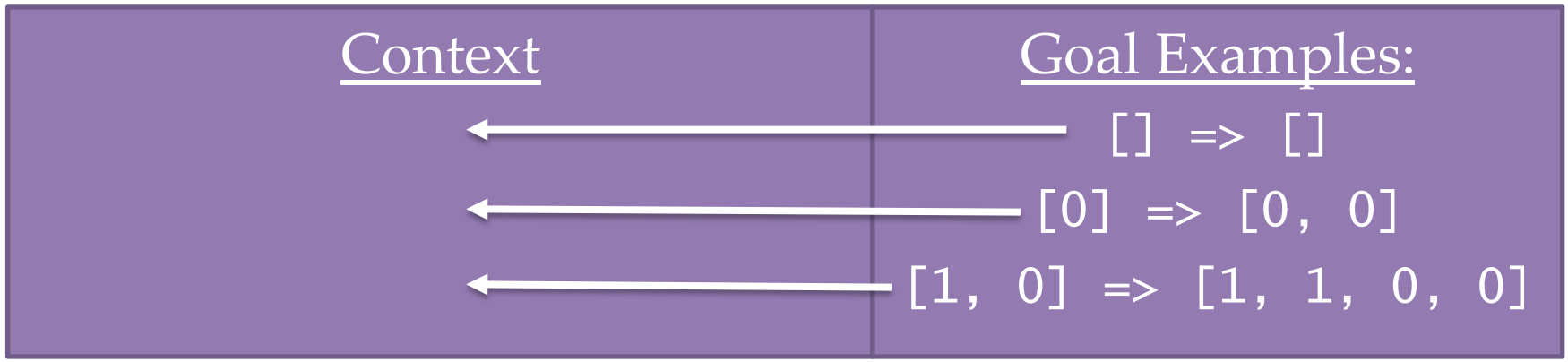
```
stutter : list -> list
```

- ■ : `list -> list`

Context	Goal Examples:
	`[] => []`
	`[0] => [0, 0]`
	`[1, 0] => [1, 1, 0, 0]`

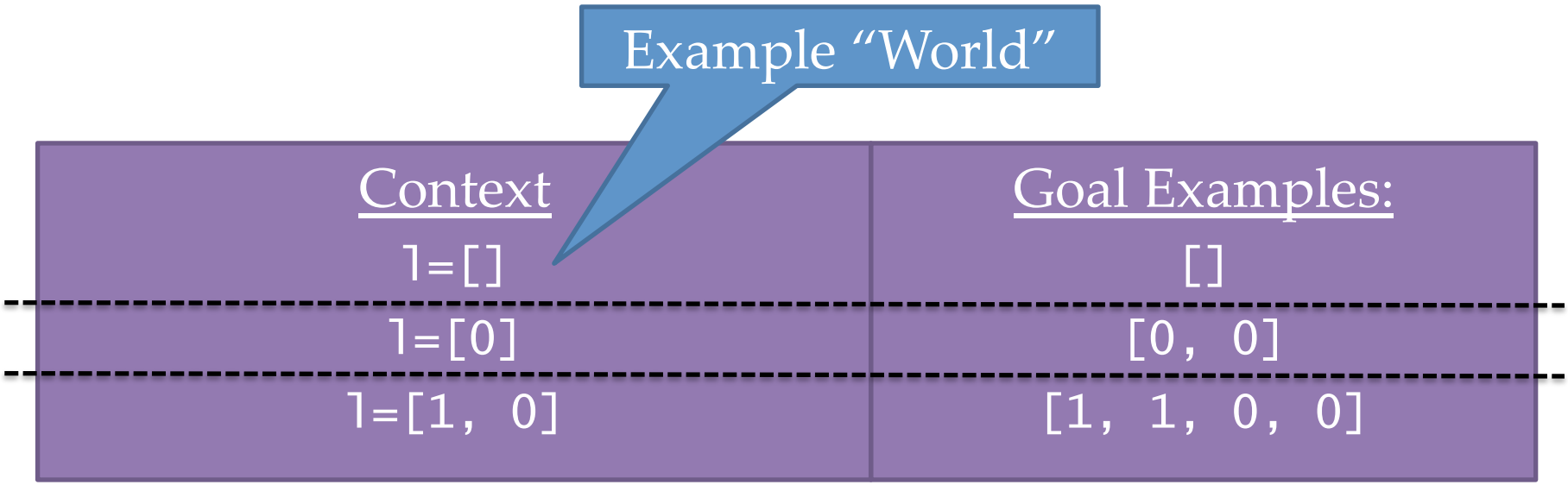
```
stutter : list -> list
```

```
let rec stutter (l:list) : list =
  ■ : list
```



```
stutter : list -> list
```

```
let rec stutter (l:list) : list =
  ■ : list
```



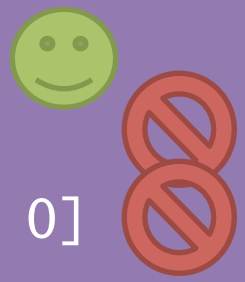
```
stutter : list -> list
```

```
let rec stutter (l:list) : list =
  l
```

1. Generate an expression*
2. Evaluate and check against each example world

*We don't generate `stutter l` – syntactic restriction on recursive calls.

Context	Goal Examples:
`l=[]`	`[]`
`l=[0]`	`[0, 0]`
`l=[1, 0]`	`[1, 1, 0, 0]`



```
stutter : list -> list
```

```
let rec stutter (l:list) : list =
 match l with
   | Nil -> ■ : list
   | Cons (x, l') -> ■ : list
```

```
stutter : list -> list
```

```
let rec stutter (l:list) : list =
 match l with
   | Nil -> ■ : list
   | Cons (x, l') -> ■ : list
```

Context	Goal Examples:
l=[]	[]

```
stutter : list -> list
```

```
let rec stutter (l:list) : list =
 match l with
   | Nil -> Nil
   | Cons (x, l') -> ■ : list
```

Context	Goal Examples:
l=[]	[] 

```
stutter : list -> list
```

```
let rec stutter (l:list) : list =
 match l with
   | Nil -> Nil
   | Cons (x, l') -> ■ : list
```

Context	Goal Examples:
l=[0], ...	[0, 0]
l=[1, 0], ...	[1, 1, 0, 0]

```
stutter : list -> list
```

```
let rec stutter (l:list) : list =
 match l with
   | Nil -> Nil
   | Cons (x, l') -> ■ : list
```

Context	Goal Examples:
l=[0], x=0, l'=[]	[0, 0]
l=[1, 0], x=1, l'=[0]	[1, 1, 0, 0]

```
stutter : list -> list
```

```
let rec stutter (l:list) : list =
 match l with
   | Nil -> Nil
   | Cons (x, l') ->
       Cons (x, Cons (x, ■ : list))
```

want: `stutter l'`

try ..., ...

Context	Goal Examples:
`l=[0], x=0, l'=[]`	`[]`
`l=[1, 0], x=1, l'=[0]`	`[0, 0]`

```
stutter : list -> list
```

```
let rec stutter (l:list) : list =
 match l with
   | Nil -> Nil
   | Cons (x, l') ->
       Cons (x, Cons (x, stutter l'))
```

```
stutter=( [] => [] | [0] => [0, 0]
        | [1, 0] => [1, 1, 0, 0]  )
```

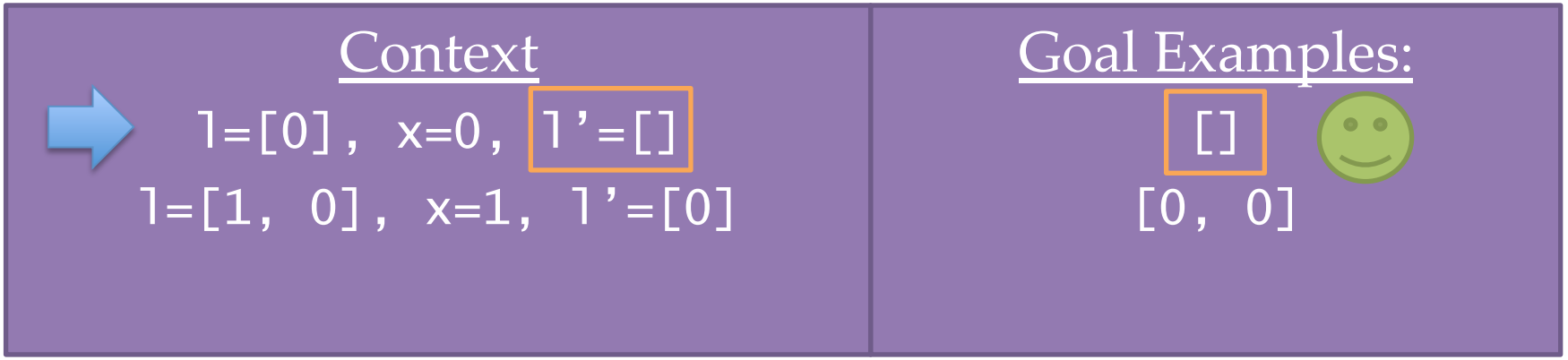
Context	Goal Examples:
l=[0], x=0, l'=[]	[]
l=[1, 0], x=1, l'=[0]	[0, 0]

```
stutter : list -> list
```

```
let rec stutter (l:list) : list =
 match l with
   | Nil -> Nil
   | Cons (x, l') ->
       Cons (x, Cons (x, stutter l'))
```

stutter [] = []

```
stutter=( [] => [] | [0] => [0, 0]
        | [1, 0] => [1, 1, 0, 0]  )
```



```
stutter : list -> list
```

```
let rec stutter (l:list) : list =
 match l with
   | Nil -> Nil
   | Cons (x, l') ->
       Cons (x, Cons (x, stutter l'))
```

stutter [0] = [0, 0]

```
stutter=( [] => [] | [0] => [0, 0]
        | [1, 0] => [1, 1, 0, 0]  )
```

Context	Goal Examples:
l=[0], x=0, l'=[]	[]
l=[1, 0], x=1, l'=[0]	[0, 0]

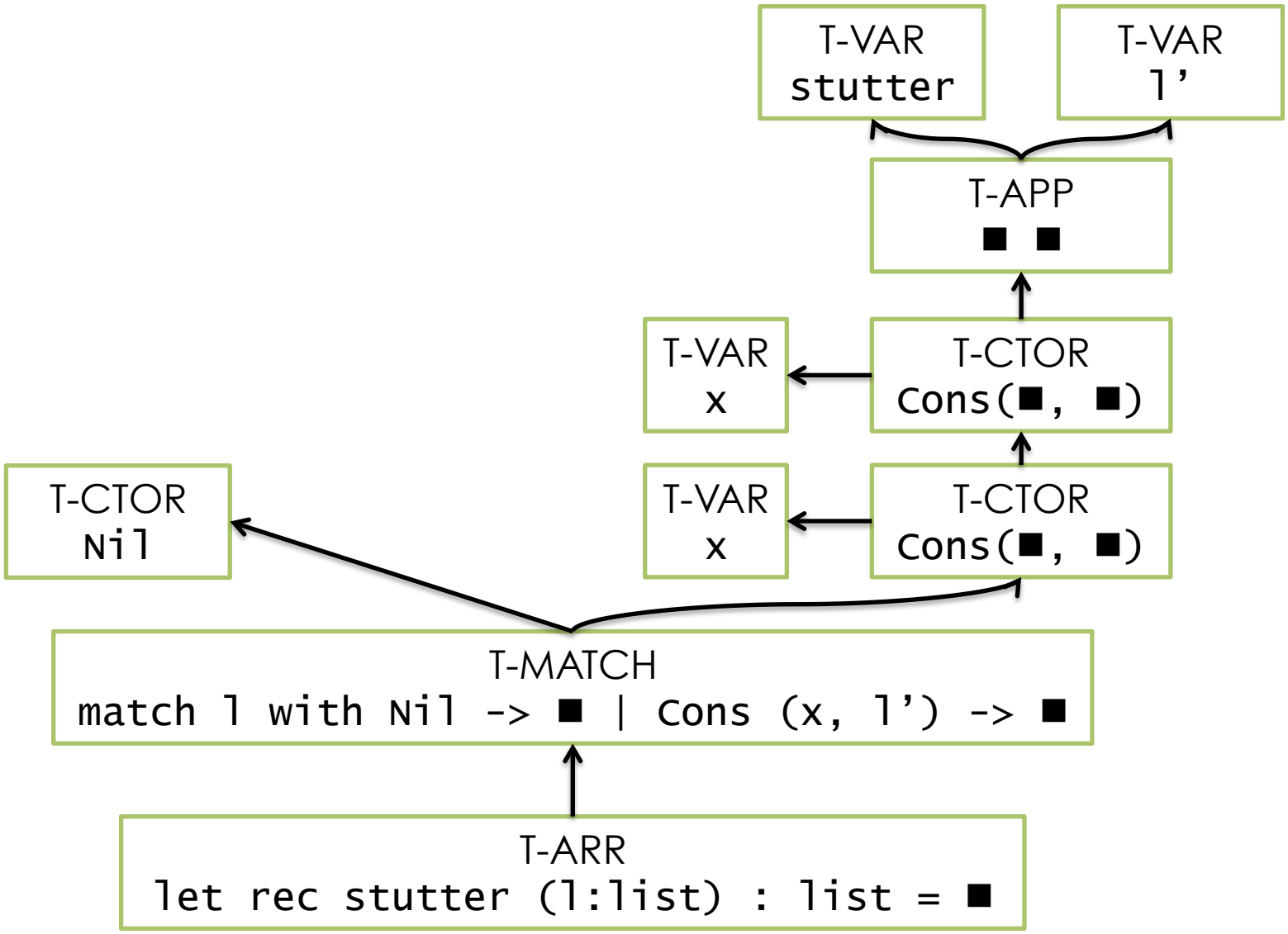
```
stutter : list -> list
```

```
let rec stutter (l:list) : list =
 match l with
   | Nil -> Nil
   | Cons (x, l') ->
       Cons (x, Cons (x, stutter l'))
```

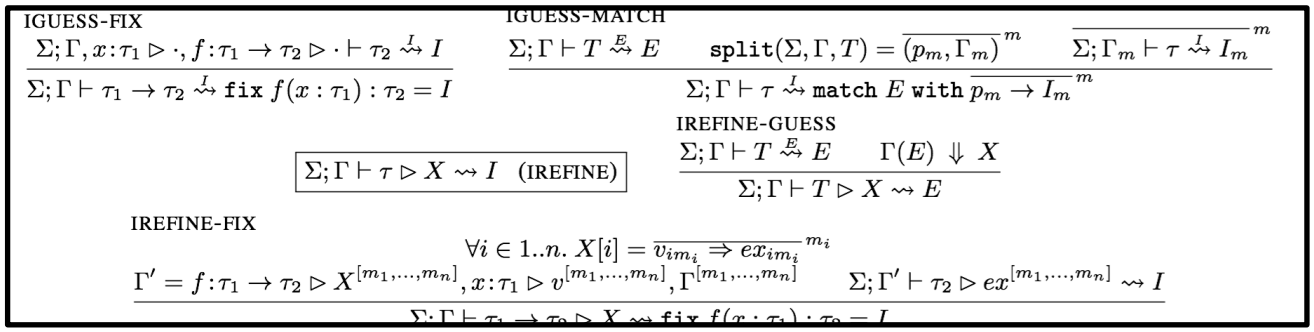
```
stutter=( [] => [] | [0] => [0, 0]
        | [1, 0] => [1, 1, 0, 0]  )
```

"Trace Completeness"

e.g., [1, 0] ⟶ [0] ⟶ []


T-VAR
stutter
T-VAR
l'
T-APP
■ ■
T-VAR
x
T-CTOR
Cons(■, ■)
T-CTOR
Nil
T-VAR
x
T-CTOR
Cons(■, ■)
T-MATCH
match l with Nil -> ■ | Cons (x, l') -> ■
T-ARR
let rec stutter (l:list) : list = ■

λ_{syn}, a logical foundation for program synthesis.



IGUESS-FIX

$$\frac{\Sigma; \Gamma, x{:}\tau_1 \triangleright \cdot, f{:}\tau_1 \to \tau_2 \triangleright \cdot \vdash \tau_2 \overset{I}{\rightsquigarrow} I}{\Sigma; \Gamma \vdash \tau_1 \to \tau_2 \overset{I}{\rightsquigarrow} \texttt{fix}\ f(x:\tau_1) : \tau_2 = I}$$

IGUESS-MATCH

$$\frac{\Sigma; \Gamma \vdash T \overset{E}{\rightsquigarrow} E \qquad \texttt{split}(\Sigma, \Gamma, T) = \overline{(p_m, \Gamma_m)}^{\,m} \qquad \overline{\Sigma; \Gamma_m \vdash \tau \overset{I}{\rightsquigarrow} I_m}^{\,m}}{\Sigma; \Gamma \vdash \tau \overset{I}{\rightsquigarrow} \texttt{match}\ E\ \texttt{with}\ \overline{p_m \to I_m}^{\,m}}$$

$$\boxed{\Sigma; \Gamma \vdash \tau \triangleright X \rightsquigarrow I \quad \text{(IREFINE)}}$$

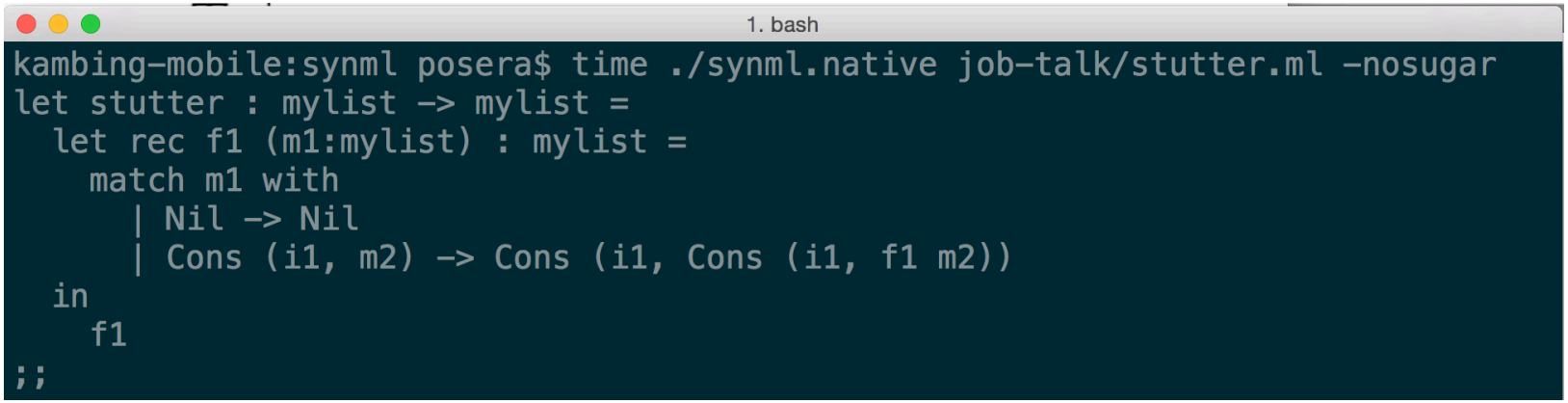
IREFINE-GUESS

$$\frac{\Sigma; \Gamma \vdash T \overset{E}{\rightsquigarrow} E \qquad \Gamma(E) \Downarrow X}{\Sigma; \Gamma \vdash T \triangleright X \rightsquigarrow E}$$

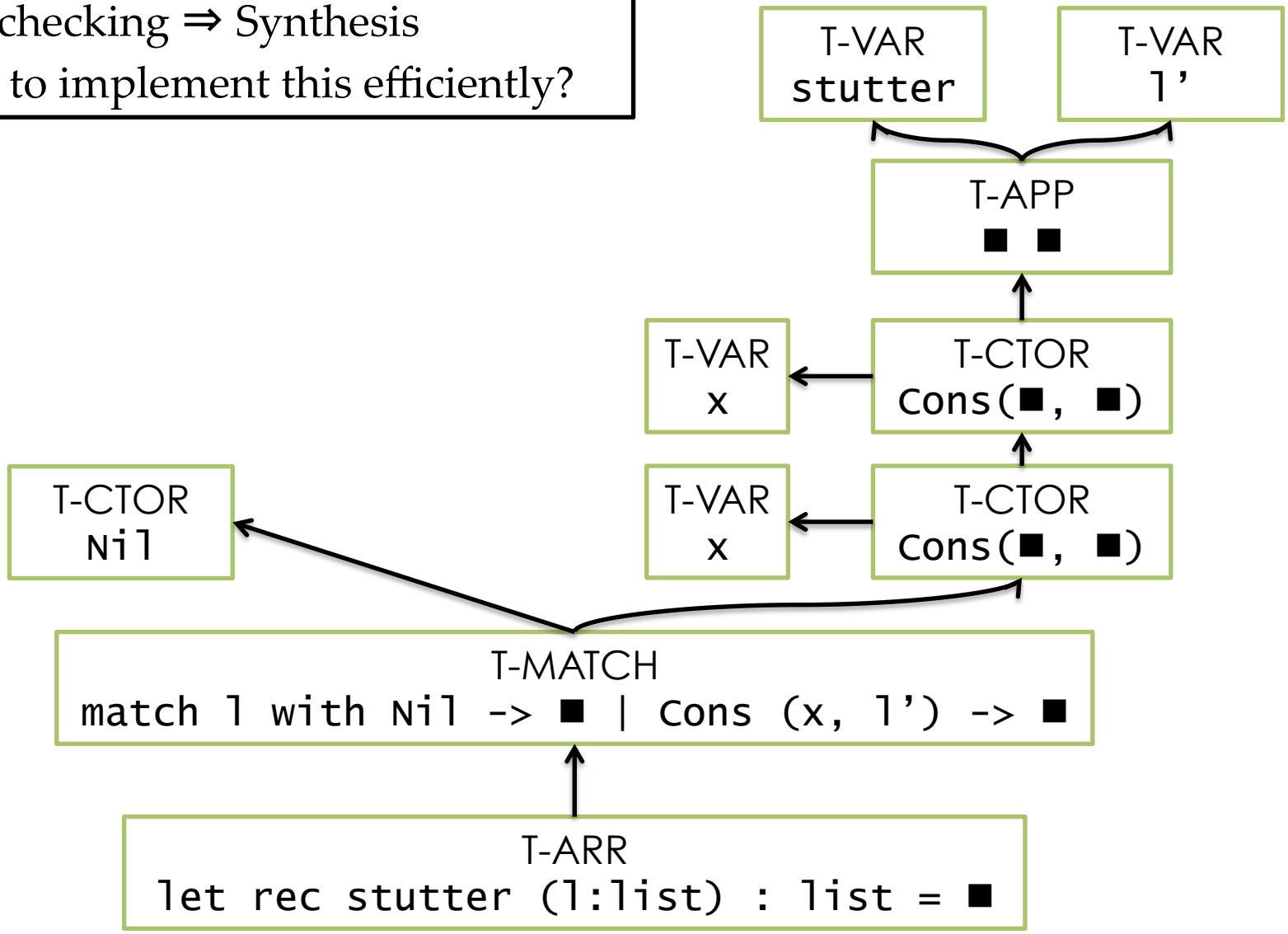
IREFINE-FIX

$$\frac{\forall i \in 1..n.\ X[i] = \overline{v_{im_i} \Rightarrow ex_{im_i}}^{\,m_i} \qquad \Gamma' = f{:}\tau_1 \to \tau_2 \triangleright X^{[m_1,\ldots,m_n]}, x{:}\tau_1 \triangleright v^{[m_1,\ldots,m_n]}, \Gamma^{[m_1,\ldots,m_n]} \qquad \Sigma; \Gamma' \vdash \tau_2 \triangleright ex^{[m_1,\ldots,m_n]} \rightsquigarrow I}{\Sigma; \Gamma \vdash \tau_1 \to \tau_2 \triangleright X \rightsquigarrow \texttt{fix}\ f(x:\tau_1) : \tau_2 = I}$$

Myth, a program synthesizer for typed, functional programs (OCaml).



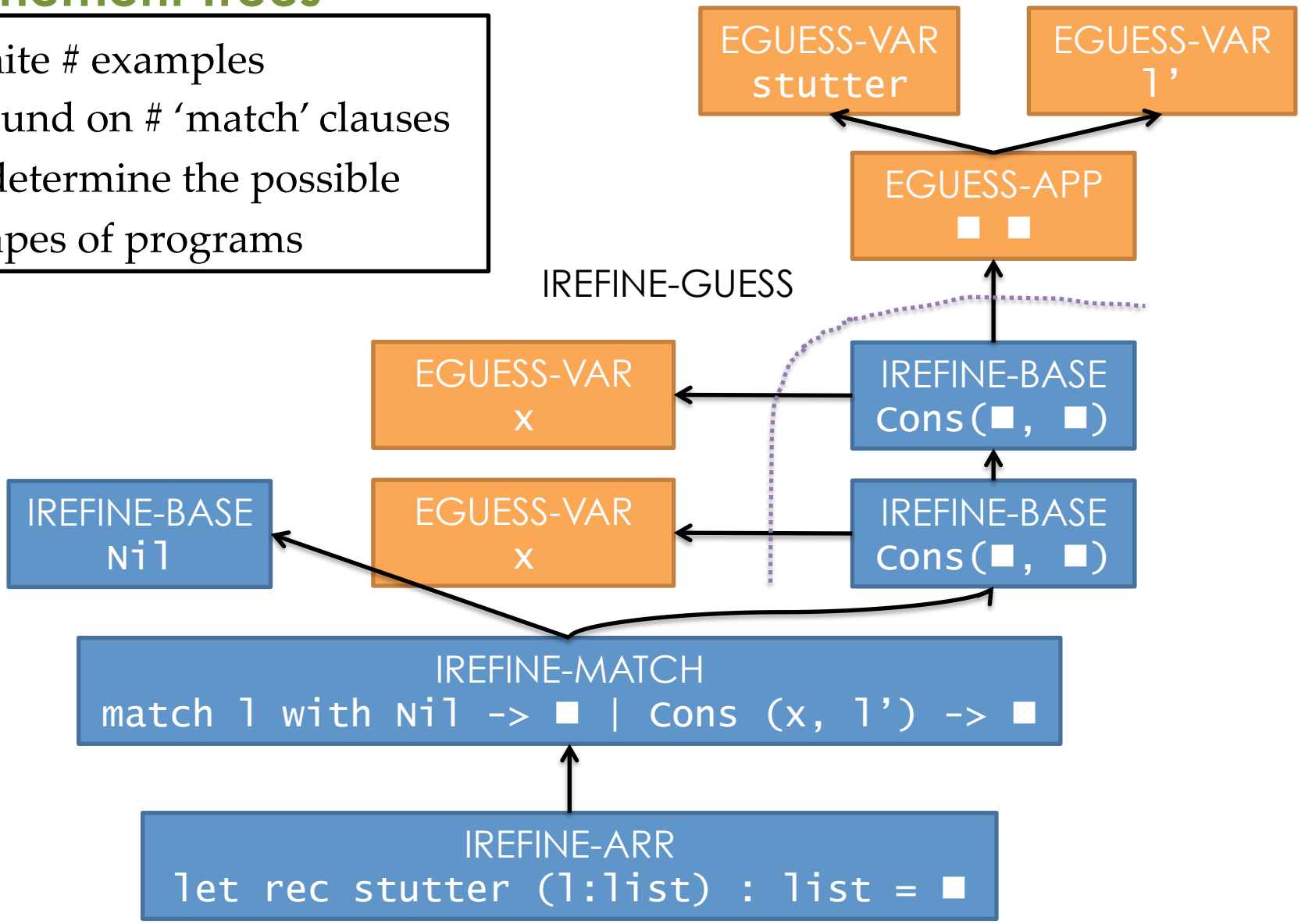
Typechecking ⇒ Synthesis
How to implement this efficiently?



Refinement Trees

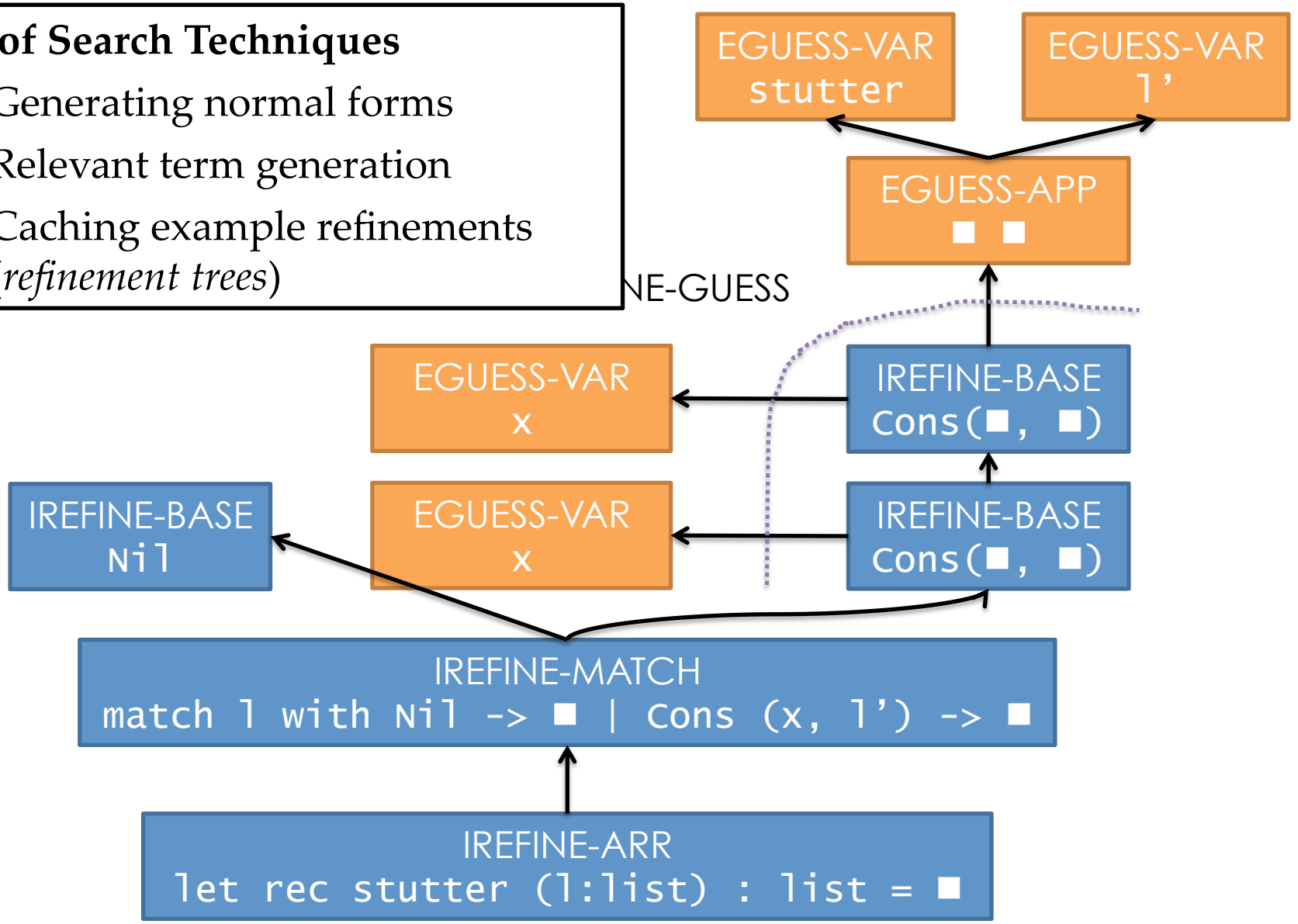
- finite # examples
- bound on # 'match' clauses

⇒ determine the possible shapes of programs



Proof Search Techniques

- Generating normal forms
- Relevant term generation
- Caching example refinements (*refinement trees*)



Algorithmic Insight: Relevance Logic ⇒ Caching

$$\mathbf{gen}_E(\Sigma;\Gamma;\tau;n)$$

"Generate type τ E-forms in context Γ with size n"

$$\mathbf{gen}_E(\Sigma;\cdot;\tau;n) = \{\}$$
$$\mathbf{gen}_E(\Sigma;\cdot;\tau;0) = \{\}$$
$$\mathbf{gen}_E(\Sigma;x{:}\tau_1,\Gamma;\tau;n) = \mathbf{gen}_E^{x:\tau_1}(\Sigma;\Gamma;\tau;n) \cup \mathbf{gen}_E(\Sigma;\Gamma;\tau;n)$$

x relevant (*must* be used)

x irrelevant (*not* used)

cacheable

Implementing Relevance

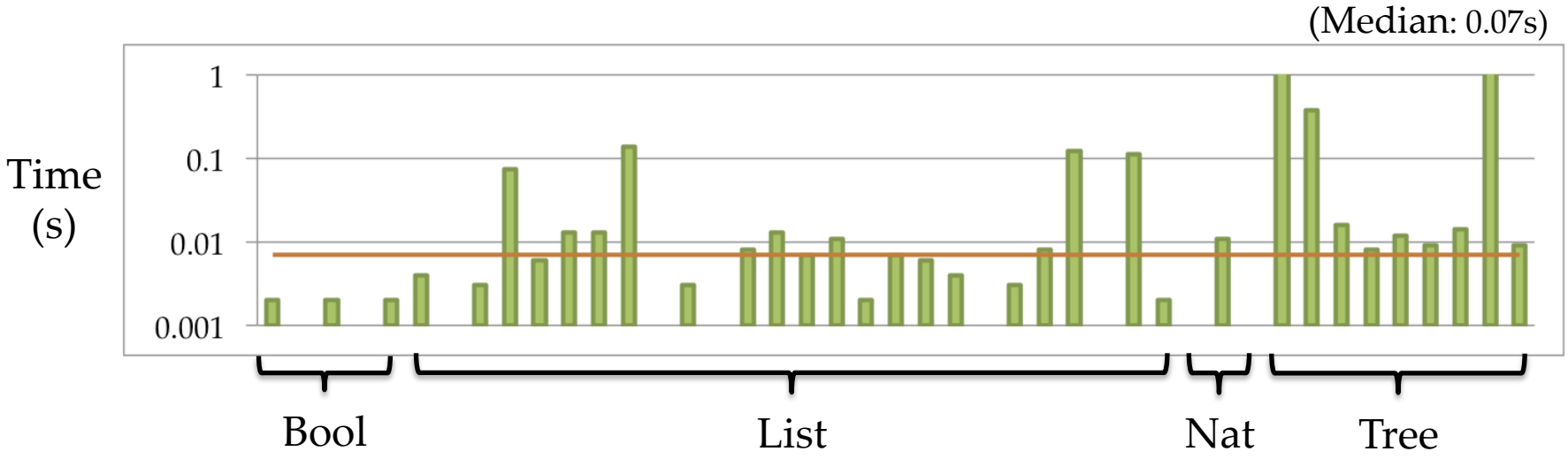
$$\mathbf{gen}_E^{x:\tau_1}(\Sigma;\Gamma;\tau;n)$$

"Generate type τ E-forms *that definitely mention* x in context Γ with size n"

$$\mathbf{gen}_E^{x:\tau_1}(\Sigma;\Gamma;\tau;0) = \{\}$$
$$\mathbf{gen}_E^{x:\tau}(\Sigma;\Gamma;\tau;1) = \{x\}$$
$$\mathbf{gen}_E^{x:\tau_1}(\Sigma;\Gamma;\tau;1) = \{\} \qquad (\tau \neq \tau_1)$$

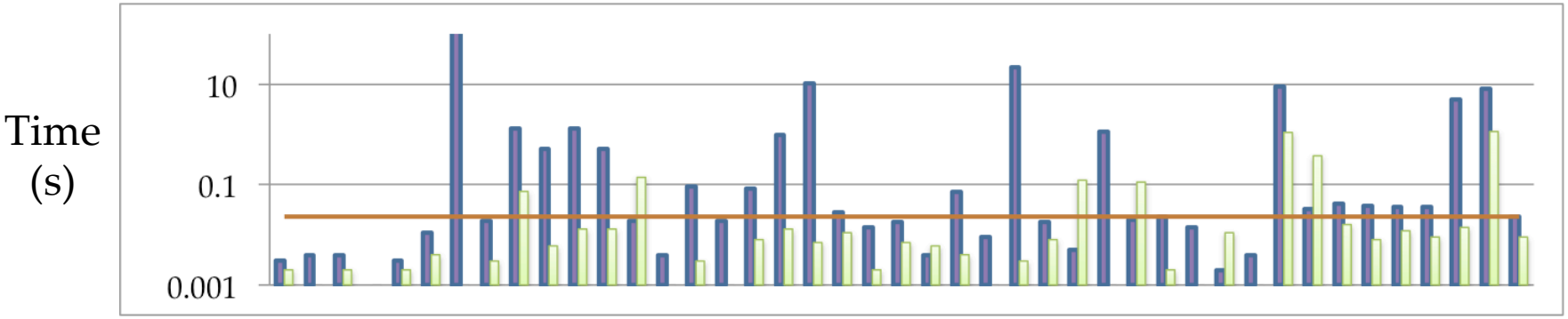
$$\mathbf{gen}_E^{x:\tau_1}(\Sigma;\Gamma;\tau;n) = \bigcup_{\tau_2\to\tau\in\Gamma} \bigcup_{k=1}^{n-1}$$

$$\begin{array}{lll} & (\mathbf{gen}_E^{x:\tau_1}(\Sigma;\Gamma;\tau_2\to\tau;k) & \underset{app}{\otimes} \quad \mathbf{gen}_I(\Sigma;\Gamma;\tau_2;n-k)) \\ \cup & (\mathbf{gen}_E(\Sigma;\Gamma;\tau_2\to\tau;k) & \underset{app}{\otimes} \quad \mathbf{gen}_I^{x:\tau_1}(\Sigma;\Gamma;\tau_2;n-k)) \\ \cup & (\mathbf{gen}_E^{x:\tau_1}(\Sigma;\Gamma;\tau_2\to\tau;k) & \underset{app}{\otimes} \quad \mathbf{gen}_I^{x:\tau_1}(\Sigma;\Gamma;\tau_2;n-k)) \end{array}$$



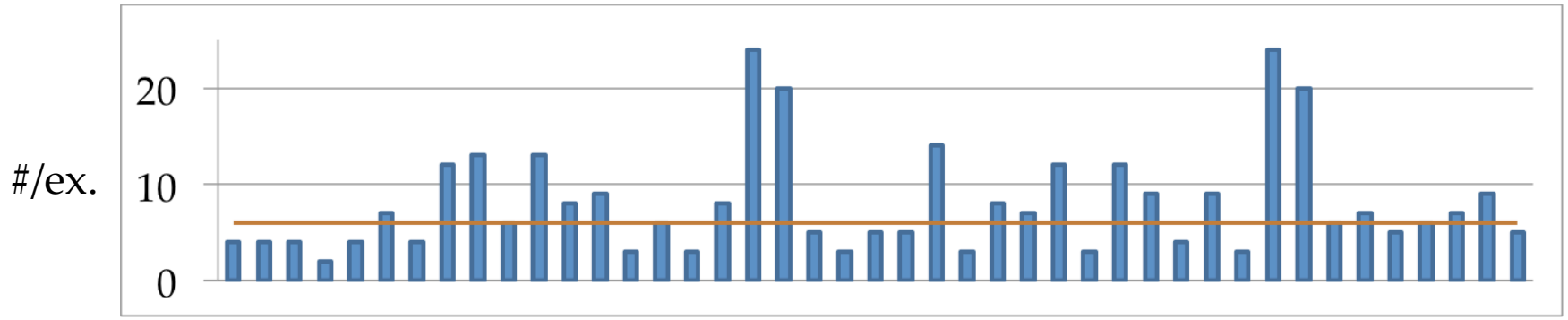
- Evaluation: 43 benchmarks tests.
  - "Intro FP programs": bools, lists, nats, and trees.
  - *Purpose of evaluation—exploration:*
    - What is the performance?
    - How many examples are necessary to generate good results?
  - **Median runtime: 0.07s.**
  - **Average #/examples: 6.**
  - **Average program size: 13.**

(Median: 0.023s)



- Synthesis in larger contexts is challenging!
  - Outs: equivalences, richer types, *e.g.*, polymorphism.

(Median: 6 examples)



- Reigning in #/examples requires additional info.
  - Ex. taking advantage of the program being synthesized...

Calculator interpreter. (22 examples, size 47, 11s)

```
let arith : exp -> nat =
  let rec f1 (e1:exp) : nat =
    match e1 with
      | Const (n1) -> n1
      | Sum (e2, e3) -> sum (f1 e2) (f1 e3)
      | Prod (e2, e3) -> mult (f1 e2) (f1 e3)
      | Pred (e2) -> (match f1 e2 with
                        | O -> 0
                        | S (n1) -> n1)
      | Max (e2, e3) -> (match compare (f1 e2) (f1 e3) with
                           | LT -> f1 e3
                           | EQ -> f1 e3
                           | GT -> f1 e2)
  in
    f1
;;
```

Free variable collector for a lambda calculus.

(31 examples, **size 75**, 3.9s)

```
let fvs_large : exp -> list =
  let rec f1 (e1:exp) : list =
    match e1 with
      | Unit -> []
      | Bvar (n1) -> []
      | FVar (n1) -> [n1]
      | Lam (n1, e2) -> f1 e2
      | App (e2, e3) -> append (f1 e2) (f1 e3)
      | Pair (e2, e3) -> append (f1 e2) (f1 e3)
      | Fst (e2) -> f1 e2
      | Snd (e2) -> f1 e2
      | Inl (e2) -> f1 e2
      | Inr (e2) -> f1 e2
      | Match (e2, n1, e3, n2, e4) ->
        (match f1 e2 with
          | Nil -> append (f1 e4) (f1 e3)
          | Cons (n3, l1) ->
            Cons (n3, append (f1 e3) (f1 e4)))
      | Const (n1) -> []
      | Binop (e2, b1, e3) -> append (f1 e3) (f1 e2)
  in
    f1
;;
```

λ_{syn}, a logical foundation for program synthesis.

IGUESS-FIX

$$\frac{\Sigma; \Gamma, x{:}\tau_1 \triangleright \cdot, f{:}\tau_1 \to \tau_2 \triangleright \cdot \vdash \tau_2 \overset{I}{\rightsquigarrow} I}{\Sigma; \Gamma \vdash \tau_1 \to \tau_2 \overset{I}{\rightsquigarrow} \texttt{fix}\ f(x : \tau_1) : \tau_2 = I}$$

IGUESS-MATCH

$$\frac{\Sigma; \Gamma \vdash T \overset{E}{\rightsquigarrow} E \qquad \texttt{split}(\Sigma, \Gamma, T) = \overline{(p_m, \Gamma_m)}^{\,m} \qquad \overline{\Sigma; \Gamma_m \vdash \tau \overset{I}{\rightsquigarrow} I_m}^{\,m}}{\Sigma; \Gamma \vdash \tau \overset{I}{\rightsquigarrow} \texttt{match}\ E\ \texttt{with}\ \overline{p_m \to I_m}^{\,m}}$$

$$\boxed{\Sigma; \Gamma \vdash \tau \triangleright X \rightsquigarrow I \quad \text{(IREFINE)}}$$

IREFINE-GUESS

$$\frac{\Sigma; \Gamma \vdash T \overset{E}{\rightsquigarrow} E \qquad \Gamma(E) \Downarrow X}{\Sigma; \Gamma \vdash T \triangleright X \rightsquigarrow E}$$

IREFINE-FIX

$$\frac{\forall i \in 1..n.\ X[i] = \overline{v_{im_i} \Rightarrow ex_{im_i}}^{\,m_i} \qquad \Gamma' = f{:}\tau_1 \to \tau_2 \triangleright X^{[m_1,\ldots,m_n]}, x{:}\tau_1 \triangleright v^{[m_1,\ldots,m_n]}, \Gamma^{[m_1,\ldots,m_n]} \qquad \Sigma; \Gamma' \vdash \tau_2 \triangleright ex^{[m_1,\ldots,m_n]} \rightsquigarrow I}{\Sigma; \Gamma \vdash \tau_1 \to \tau_2 \triangleright X \rightsquigarrow \texttt{fix}\ f(x : \tau_1) : \tau_2 = I}$$

Soundness

$$\Gamma \vdash \tau \triangleright X \rightsquigarrow e$$

$$\Gamma \vdash e : \tau$$

(Type Soundness)

$$e \vDash X$$

(Example Soundness)

λ_{syn}, a logical foundation for program synthesis.

IGUESS-FIX

$$\frac{\Sigma; \Gamma, x{:}\tau_1 \triangleright \cdot, f{:}\tau_1 \to \tau_2 \triangleright \cdot \vdash \tau_2 \overset{I}{\rightsquigarrow} I}{\Sigma; \Gamma \vdash \tau_1 \to \tau_2 \overset{I}{\rightsquigarrow} \texttt{fix}\ f(x:\tau_1) : \tau_2 = I}$$

IGUESS-MATCH

$$\frac{\Sigma; \Gamma \vdash T \overset{E}{\rightsquigarrow} E \qquad \texttt{split}(\Sigma, \Gamma, T) = \overline{(p_m, \Gamma_m)}^{\,m} \qquad \overline{\Sigma; \Gamma_m \vdash \tau \overset{I}{\rightsquigarrow} I_m}^{\,m}}{\Sigma; \Gamma \vdash \tau \overset{I}{\rightsquigarrow} \texttt{match}\ E\ \texttt{with}\ \overline{p_m \to I_m}^{\,m}}$$

$$\boxed{\Sigma; \Gamma \vdash \tau \triangleright X \rightsquigarrow I \quad \text{(IREFINE)}}$$

IREFINE-GUESS

$$\frac{\Sigma; \Gamma \vdash T \overset{E}{\rightsquigarrow} E \qquad \Gamma(E) \Downarrow X}{\Sigma; \Gamma \vdash T \triangleright X \rightsquigarrow E}$$

IREFINE-FIX

$$\frac{\forall i \in 1..n.\ X[i] = \overline{v_{im_i} \Rightarrow ex_{im_i}}^{\,m_i} \qquad \Gamma' = f{:}\tau_1 \to \tau_2 \triangleright X^{[m_1,\ldots,m_n]}, x{:}\tau_1 \triangleright v^{[m_1,\ldots,m_n]}, \Gamma^{[m_1,\ldots,m_n]} \qquad \Sigma; \Gamma' \vdash \tau_2 \triangleright ex^{[m_1,\ldots,m_n]} \rightsquigarrow I}{\Sigma; \Gamma \vdash \tau_1 \to \tau_2 \triangleright X \rightsquigarrow \texttt{fix}\ f(x:\tau_1) : \tau_2 = I}$$

Completeness

$$\Gamma \vdash e : \tau$$
$$e \vDash X \quad \Longrightarrow \quad \Gamma \vdash \tau \triangleright X \rightsquigarrow e$$

(Due to deciding equality between recursive functions.)

Example-Directed Synthesis: A Type-Theoretic Interpretation

[POPL 2016: Frankle, Osera, Walker, & Zdancewic]

$$\Gamma \vdash \tau \rhd X \rightsquigarrow e$$

What are "examples"?

Examples *are* Refinement Types

Singleton types (constructors): `[], 0, cons, …`

Function types: `t1 -> t2`

Intersection types: `t1 /\ t2`

```
stutter : list -> list /\
          [] -> []     /\
          [1] -> [1,1] /\
          [2,1] -> [2,2,1,1]
```

More Expressive Types

Union types: `t1 \/ t2`

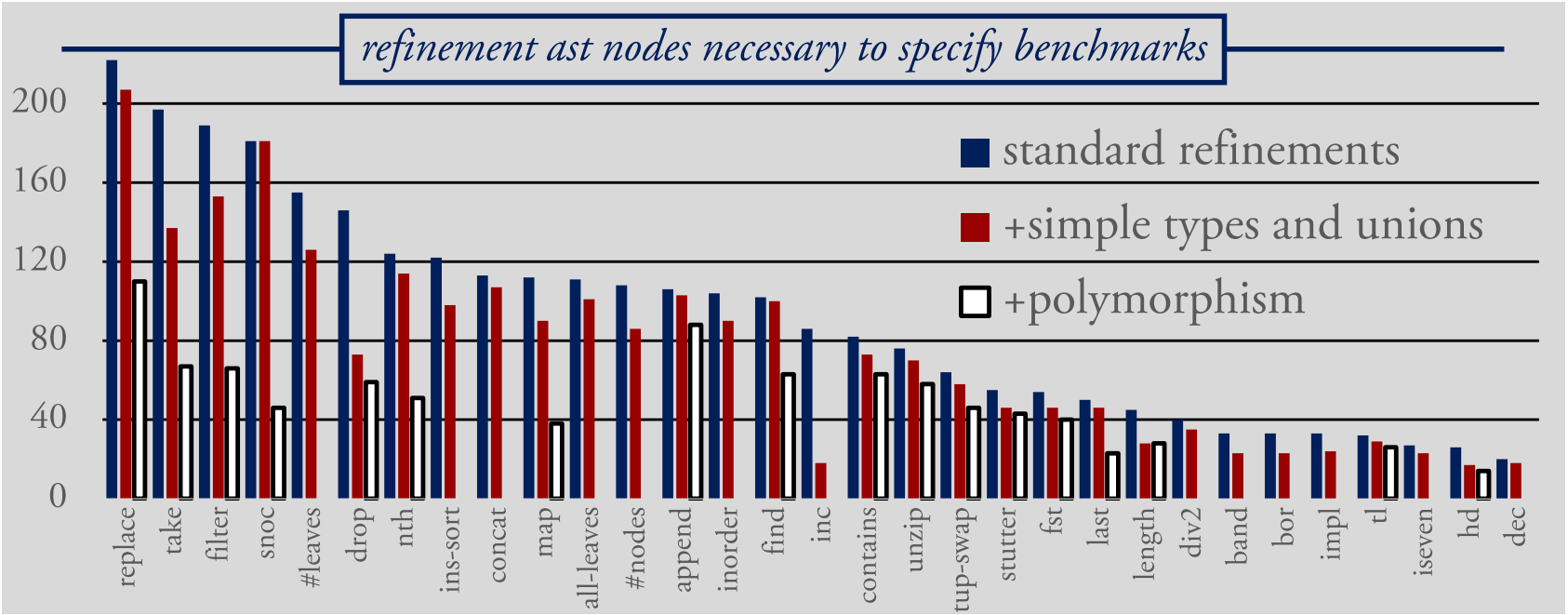
(Limited) Negations: `~t1`

Allows for counter-example guided user interactions.

Polymorphic Types: `∀a. t`

Expresses symmetries

Specification Size



Significantly more succinct examples
(~20% shorter) & simpler specifications.

Tradeoffs in Search

```
decrement : nat -> nat /\
            0 -> 0     /\
            1 -> 0     /\
            2 -> 1
```

(in the context)

```
decrement(■) : goal
```

0 -> 0	1 -> 0	2 -> 1	goal
			0
			0
			1

Argument "Sudoku"

```
decrement : nat -> nat /\
            0 -> 0     /\
            1 -> 0     /\
            2 -> 1
```

(in the context)

```
decrement(■) : goal
```

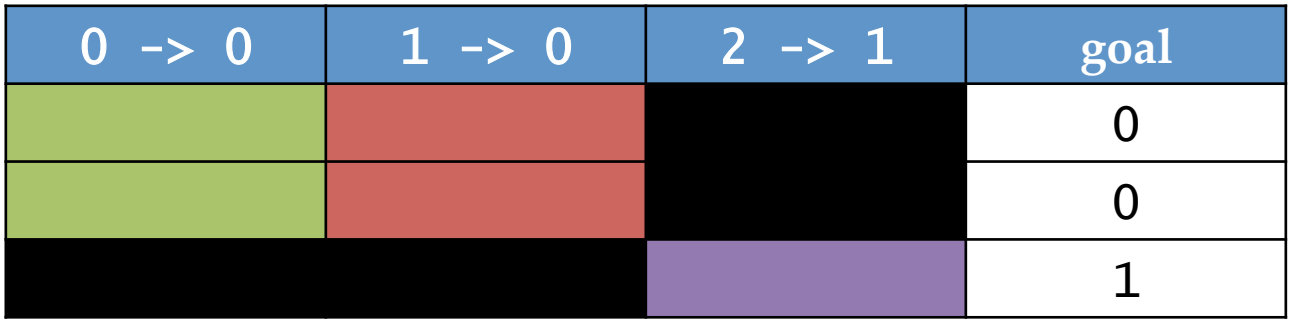
0 -> 0	1 -> 0	2 -> 1	goal
		■	0
		■	0
■	■		1

Argument "Sudoku"

```
decrement : nat -> nat /\
            0 -> 0     /\
            1 -> 0     /\
            2 -> 1
```

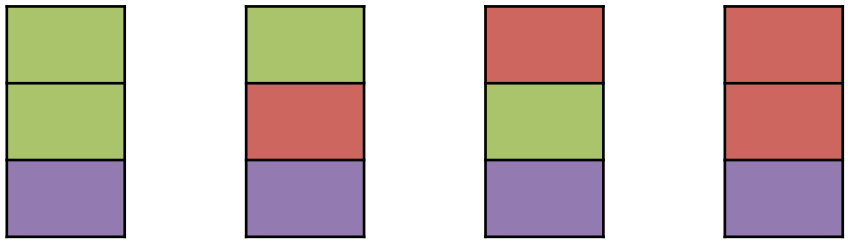
(in the context)

```
decrement(■) : goal
```



Argument "Sudoku"

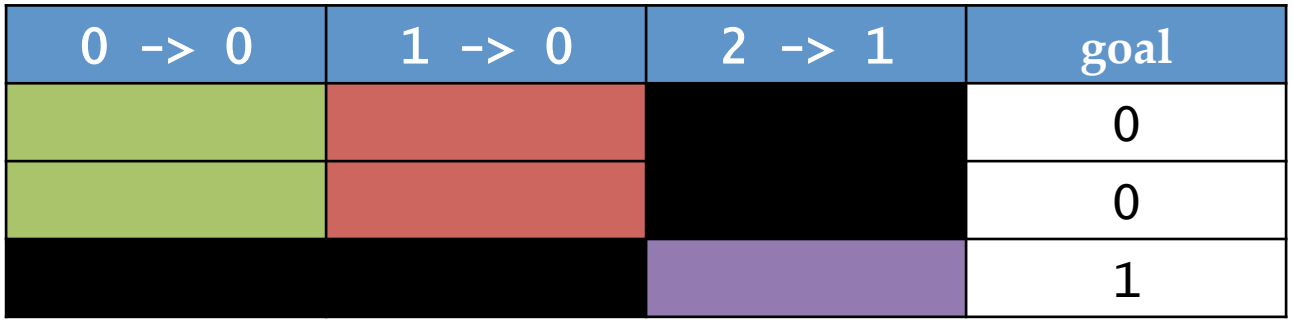
0 -> 0	1 -> 0	2 -> 1	goal
			0
			0
			1

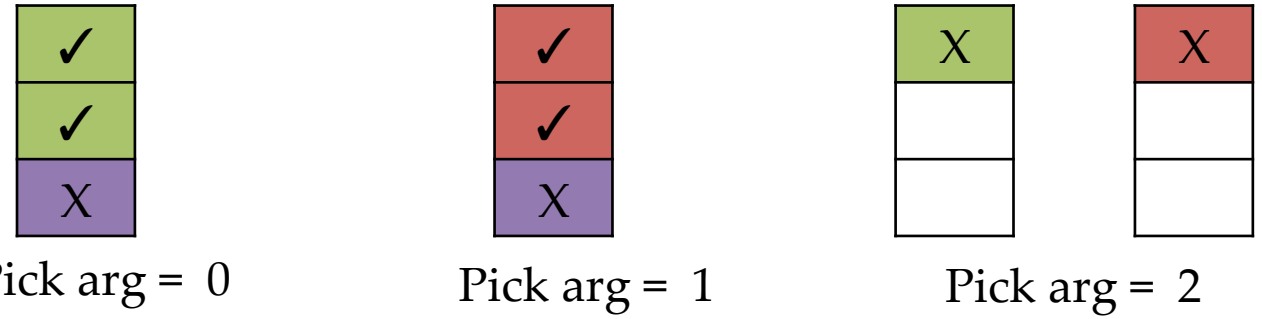


"pick boxes then find candidates"

⇒ 12 search problems (all of which fail for constant terms)

Argument "Sudoku"





In practice, we found raw enumeration to be better for performance than "smarter" search approaches.

Moral: The "smoothness" of the search space can be important

Limitations / Challenges

- Enumerative search
  - fundamentally combinatorial
- Trace completeness
  - use the definition of the partially defined function?
- I/O examples only
- Not full verification
  - other validation needed for correctness
- Normal forms
  - cannot synthesize “helper” functions
  - minimal program size can be exponentially bigger

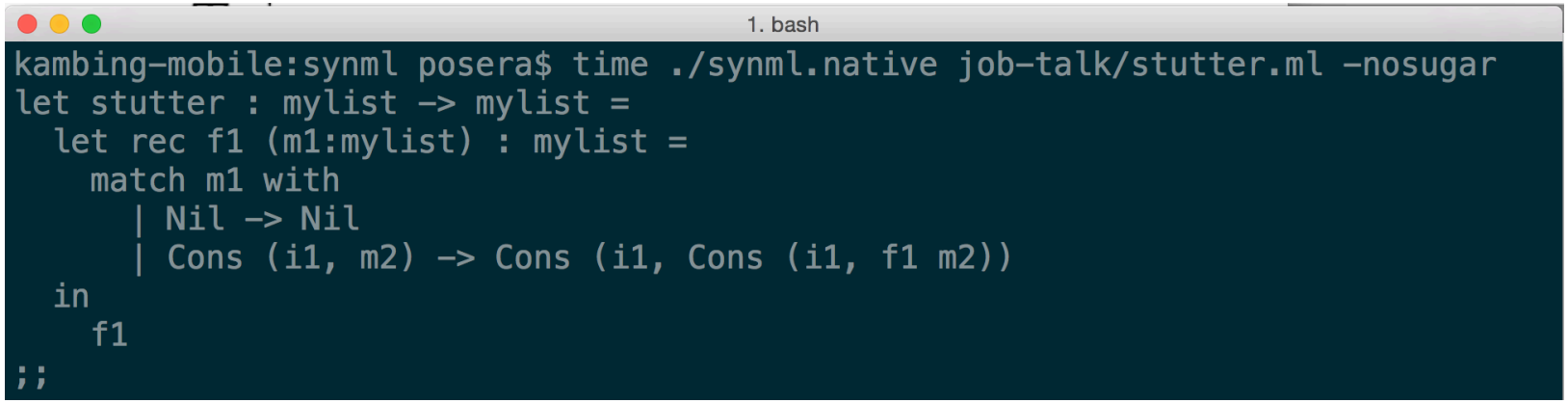
Benefits

- Type structure largely determines program structure

  ⇒ very good for “wide” & “shallow” search

- Suggests principled ways to extend to richer language features

A Surprising Example



(**MYTH** DEMO)

Research Directions / Questions

- Richer specification languages
  - *Beyond I/O: fewer examples, more general constraints,*
- Other search optimizations
  - *(Higher-order) unification*
- Combination with other techniques
  - *CEGIS, constraint solving with SAT or SMT*
- Combine with domain-specific languages
  - *Application-specific types / combinators*

Related Work

- Proof Search:
  - *Automated Theorem Proving (lecture notes)* [Pfenning]
- Recent Papers (very partial! list):
  - *Program Synthesis from Polymorphic Refinement Types*
    [PLDI 2016: Polikarpova, Kuraj, Solar-Lezama]
  - *Example-Directed Synthesis: A Type-Theoretic Interpretation*
    [POPL 2016: Frankle, Osera, Walker, & Zdancewic]
  - *Type-and-Example-Driven Program Synthesis*
    [PLDI 2015: Osera, Zdancewic]
  - *Synthesizing Data Structure Transformations from Input-Output Examples*
    [PLDI 2015: Feser, Chaudhuri, Dillig]
  - *Test-driven Synthesis*
    [Perelmen, et al. 2014]
  - *Recursive Program Synthesis*
    [Albarghouthi, Gulwani, Kincaid 2013]
  - *Complete Completion Using Types and Weights*
    [Gvero, et al. 2013]

Type and Example Directed Program Synthesis

1. Synthesize programs with *higher-order functions*, *recursion*, and *algebraic datatypes*.
2. Type structure prunes the search space.
3. Take advantage of techniques from proof theory literature

ExCAPE
Expeditions in Computer Augmented Program Engineering

```
let list_pairwise_swap : list -> list =
  let rec f1 (l1:list) : list =
    match l1 with
      | Nil -> []
      | Cons (n1, l2) ->
        (match f1 l2 with
          | Nil ->
            (match l2 with
              | Nil -> []
              | Cons (n2, l3) ->
                  Cons (n2, Cons (n1, f1 l3)))
          | Cons (n2, l3) -> [])
  in
    f1
;;
```

??

f1 l2 ≡
"Does l2 have even length?"

Ex. [0, 1, 0, 1] ⇒ [1, 0, 1, 0]
[0, 1, 0] ⇒ []