Harvard John A. Paulson
School of Engineering
and Applied Sciences

PL/HCI Seminar (252R/279R)

Type-and-Example-Directed Program Synthesis

Osera & Zdancewic, PLDI’15

Design Argument

- Person *?* in setting *Typed Functional Programming* wants to **(G)** synthesize program from examples but obstacles **(O1)** large search space get in the way. Any solution has to satisfy constraints **(X1)** types & **(X2)** examples, minimize costs **(Y1)** runtime & **(Y2)** complexity of solution, and avoid obstacles of **(Z1)** incorrect solution.

- **(A1)** “synthesis techniques have many potential applications” **(A2)** “synthesizing programs with structured data, recursion and higher-order functions in a typed programming language” is useful

- Approach MYTH has characteristics **(C1)** type-driven **(C2)** evaluate during enumeration **(C3)** prune by examples that help achieve goal **(G)** while avoiding obstacles **(O1)** & **(Z1)**.

Typed FP Program Synthesis

```
(∗ Type signature for natural numbers and lists ∗)

type nat = O | S of nat

type list = Nil | Cons of nat ∗ list

(∗ Goal type refined by input / output examples ∗)

let stutter : list -> list |>
  { [] => [] | [0] => [0;0] | [1;0] => [1;1;0;0] } = ?
```

Typed FP Program Synthesis

```
(* Type signature for natural numbers and lists *)

type nat = O | S of nat

type list = Nil | Cons of nat * list

(* Goal type refined by input / output examples *)

let stutter : list -> list |>
  { [] => [] | [0] => [0;0] | [1;0] => [1;1;0;0] } = ?
```

Algebraic
Data Types

Typed FP Program Synthesis

```
(* Type signature for natural numbers and lists *)

type nat = O | S of nat

type list = Nil | Cons of nat * list

(* Goal type refined by input / output examples *)

let stutter : list -> list |>
  { [] => [] | [0] => [0;0] | [1;0] => [1;1;0;0] } = ?
```

Example-
Driven

Typed FP Program Synthesis

```
(* Output: synthesized implementation of stutter *)
let stutter : list -> list =
  let rec f1 (l1:list) : list =
    match l1 with
    | Nil -> l1
    | Cons(n1, l2) -> Cons(n1, Cons(n1, f1 l2))
  in f1
```

Typed FP Program Synthesis

Recursive Function

```
(* Output: synthesized implementation of stutter *)
let stutter : list -> list =
  let rec f1 (l1:list) : list =
    match l1 with
    | Nil -> l1
    | Cons(n1, l2) -> Cons(n1, Cons(n1, f1 l2))
  in f1
```

Typed FP Program Synthesis

Recursive Function

```
(* Output: synthesized implementation of stutter *)
let stutter : list -> list =
  let rec f1 (l1:list) : list =
    match l1 with
    | Nil -> l1
    | Cons(n1, l2) -> Cons(n1, Cons(n1, f1 l2))
  in f1
```

Structurally Recursive

Approach Overview

- **Type Refinement**
  from type, we know input is type list and output is type list. Refine each example into a “world”.

- **Guessing**
  must agree on all examples
  must be structurally recursive

- **Match Refinement**
  case analysis on algebraic data type
  split examples accordingly

- **Recursive Functions**
  examples as approximation

Limitations

- **Type Refinement**
  from type, we know input is type list and output is type list. Refine each example into a “world”.

- **Guessing**
  must agree on all examples
  must be structurally recursive

- **Match Refinement**
  case analysis on algebraic data type
  split examples accordingly

- **Recursive Functions**
  examples as approximation

Key ideas

- Program synthesis as proof search
- Prune search space by input–output examples
- Refinement tree

“The rules make explicit when we are checking types (I–forms) versus generating types (E–forms), respectively. We can think of type information flowing into I–forms whereas type informations flows out of E–forms. This analogy of information flow extends to synthesis: when synthesizing I–forms, we push type–and–example information inward. In contrast, we are not able to push this information into E–forms.”

$\Sigma\,;\,\Gamma \vdash e : \tau$	e is well-typed.
$\Sigma\,;\,\Gamma \vdash E \Rightarrow \tau$	E produces type τ.
$\Sigma\,;\,\Gamma \vdash I \Leftarrow \tau$	I checks at type τ.
$\Sigma\,;\,\Gamma \vdash X : \tau$	X checks at type τ.

$\Sigma\,;\,\Gamma \vdash \tau \overset{E}{\rightsquigarrow} E$ (EGUESS): guess an E of type τ.

$\Sigma\,;\,\Gamma \vdash \tau \triangleright X \overset{I}{\rightsquigarrow} I$ (IREFINE): refine and synthesize an I of type τ that agrees with examples X.

$$\boxed{\Sigma\,;\ \Gamma \vdash \tau \overset{E}{\rightsquigarrow} E}$$

EGUESS_VAR

$$\frac{x : \tau \ \in\ \Gamma}{\Sigma\,;\ \Gamma \vdash \tau \overset{E}{\rightsquigarrow} x}$$

EGUESS_APP

$$\frac{\begin{array}{c}\Sigma\,;\ \Gamma \vdash \tau_1 \to \tau \overset{E}{\rightsquigarrow} E \\ \Sigma\,;\ \Gamma \vdash \tau_1 \rhd \cdot \overset{I}{\rightsquigarrow} I\end{array}}{\Sigma\,;\ \Gamma \vdash \tau \overset{E}{\rightsquigarrow} E\, I}$$

$$\boxed{\Sigma\,;\,\Gamma \vdash \tau \overset{E}{\rightsquigarrow} E}$$

Guess an *E* of type τ

EGUESS_VAR

$$\frac{x : \tau \in \Gamma}{\Sigma\,;\,\Gamma \vdash \tau \overset{E}{\rightsquigarrow} x}$$

APP

$$\frac{\Sigma\,;\,\Gamma \vdash \tau_1 \to \tau \overset{E}{\rightsquigarrow} E \qquad \Sigma\,;\,\Gamma \vdash \tau_1 \triangleright \cdot \overset{I}{\rightsquigarrow} I}{\Sigma\,;\,\Gamma \vdash \tau \overset{E}{\rightsquigarrow} E\, I}$$

$$\boxed{\Sigma \,;\, \Gamma \vdash \tau \overset{E}{\rightsquigarrow} E}$$

EGUESS_VAR

$$\frac{x : \tau \in \Gamma}{\Sigma \,;\, \Gamma \vdash \tau \overset{E}{\rightsquigarrow} x}$$

EGUESS_APP

$$\Sigma \,;\, \Gamma \vdash \tau_1 \to \tau \overset{E}{\rightsquigarrow} E$$

"Generating a variable requires no recursive generation—we simply choose any variable from the context of the appropriate type."

$$\boxed{\Sigma \, ; \, \Gamma \vdash \tau \overset{E}{\rightsquigarrow} E}$$

EGUESS_VAR

“Generating an application consists of generating a function that produces the desired goal type and then generating a compatible argument.”

EGUESS_APP

$$\frac{\begin{array}{c} \Sigma \, ; \, \Gamma \vdash \tau_1 \rightarrow \tau \overset{E}{\rightsquigarrow} E \\ \Sigma \, ; \, \Gamma \vdash \tau_1 \rhd \cdot \overset{I}{\rightsquigarrow} I \end{array}}{\Sigma \, ; \, \Gamma \vdash \tau \overset{E}{\rightsquigarrow} E \, I}$$

$$\boxed{\Sigma\;;\;\Gamma \vdash \tau \triangleright X \overset{I}{\rightsquigarrow} I}$$

IREFINE_GUESS

$$\frac{\Sigma\;;\;\Gamma \vdash \tau \overset{E}{\rightsquigarrow} E \qquad E \models X}{\Sigma\;;\;\Gamma \vdash \tau \triangleright X \overset{I}{\rightsquigarrow} E}$$

… other cases for constructor, fix, match …

$$\boxed{\Sigma ; \Gamma \vdash \tau \rhd X \overset{I}{\rightsquigarrow} I}$$

IREFINE_GUESS

$$\frac{\Sigma ; \Gamma \vdash \tau \overset{E}{\rightsquigarrow} E \qquad E \models X}{\Sigma ; \Gamma \vdash \tau \rhd X \overset{I}{\rightsquigarrow} E}$$

… other cases for constructor, fix, match …

“Because Es are also syntactically considered Is, generate an E by using the [previous] judgment.”

$$\boxed{\Sigma ; \Gamma \vdash \tau \triangleright X \overset{I}{\rightsquigarrow} I}$$

IREFINE_GUESS

$$\frac{\Sigma ; \Gamma \vdash \tau \overset{E}{\rightsquigarrow} E \qquad E \models X}{\Sigma ; \Gamma \vdash \tau \triangleright X \overset{I}{\rightsquigarrow} E}$$

… other cases for constructor, fix, match …

“generate a constructor value by recursively generating arguments to that constructor”

$$\boxed{\Sigma \,;\, \Gamma \vdash \tau \triangleright X \overset{I}{\rightsquigarrow} I}$$

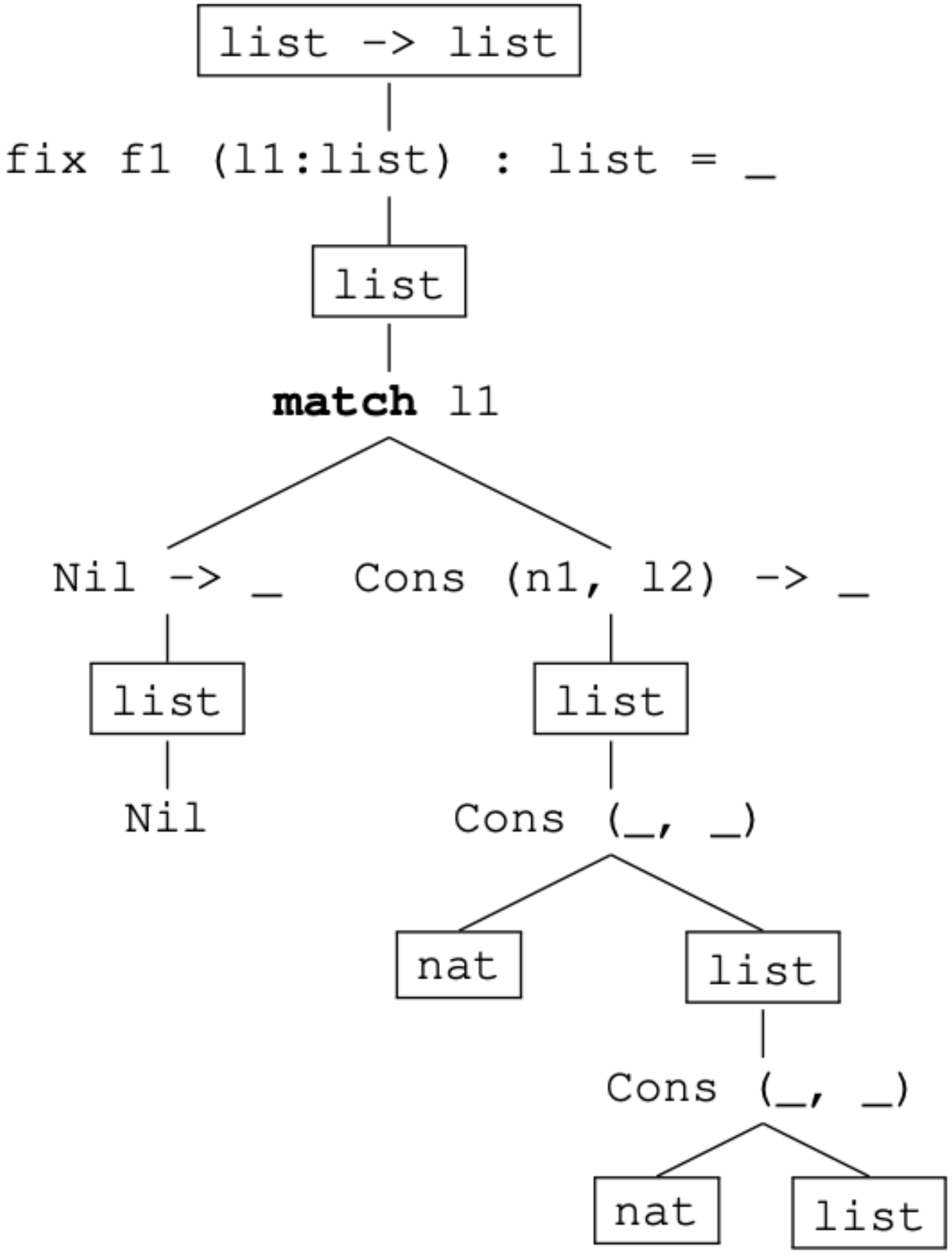
IREFINE_GUESS

$$\frac{\Sigma \,;\, \Gamma \vdash \tau \overset{E}{\rightsquigarrow} E \qquad E \vDash X}{\Sigma \,;\, \Gamma \vdash \tau \triangleright X \overset{I}{\rightsquigarrow} E}$$

… other cases for constructor, fix, match …

“Our new insight is that it is possible to modify the typing rules so that they “push” examples towards the leaves of the typing derivation trees that serve as the scaffolding for the generated program terms. Doing so permits the algorithm to evaluate candidate terms early in the search process, thereby potentially pruning the search space dramatically. Rather than following the naïve strategy of “enumerate and then evaluate,” our algorithm follows the more nuanced approach of “evaluate during enumeration.””

“A refinement tree is a data structure that describes all the possible shapes (using I–forms) that our synthesized program can take, as dictated by the given examples. Alternatively, it represents the partial evaluation of the synthesis search procedure against the examples. In [the stutter synthesis], match has been specialized to case on l1, the only informative scrutinee justified by the examples.”


list -> list
fix f1 (l1:list) : list = _
list
match l1
Nil -> _
Cons (n1, l2) -> _
list
Nil
list
Cons (_, _)
nat
list
Cons (_, _)
nat
list

Limitations

Trace–Complete (No)

```
let list_stutter : list -> list |>
  { [] => []
  | [0] => [0;0]
  | [1;0] => [1;1;0;0]
  (*| [1;1;0] => [1;1;1;1;0;0]*)
  | [1;1;1;0] => [1;1;1;1;1;1;0;0]
  } = ?
```

Trace–Complete (OK)

```
let list_stutter : list -> list |>
  { [] => []
  | [0] => [0;0]
  | [1;0] => [1;1;0;0]
  (*| [1;1;0] => [1;1;1;1;0;0]*)
  (*| [1;1;1;0] => [1;1;1;1;1;1;0;0]*)
  } = ?
```

Trace–Complete (OK)

```
let list_stutter : list -> list |>

  { [] => []

  | [0] => [0;0]

  | [1;0] => [1;1;0;0]

  | [1;1;0] => [1;1;1;1;0;0]

  | [1;1;1;0] => [1;1;1;1;1;1;0;0]

  } = ?
```

“However, in the presence of recursive functions, doing so is unsound. Consider synthesizing the Cons branch of the stutter function from Section 2 but with the example set {[] => [], [1;0] => [1;1;0;0]}. If we synthesized the term f1 l2 rather than Cons(n1, Cons(n1, f1 l2)), then we will encounter a NoMatch exception because l2 = [0]. This is because our example set for f1 contains no example for [0]. If we simply accepted f1 l2, then we would admit a term that contradicted our examples since f1 l2 actually evaluates to [] once plugged into the overall recursive function.”

Other Limitations

- No higher-order functions in input/output.
- Helper functions have to be provided in context.

Arith Example

```
let arith : exp -> nat |>
{ Const (0) => 0 | Const (1) => 1 | Const (2) => 2
| Sum (Const(2), Const(2)) => 4
| Sum (Const(2), Const(1)) => 3
| Sum (Const(0), Const(2)) => 2
| Prod (Const(0), Const(2)) => 0
| Prod (Const(2), Const(1)) => 2
| Prod (Const(2), Const(2)) => 4
| Prod (Prod(Const(2), Const(2)), Const(2)) => 8
| Prod (Sum(Const(2), Const(1)), Const(2)) => 6
(* … *)
} = ?
```

Arith Example

```
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)
      (* … *)
  in f1
```

Artifact

- https://github.com/silky/myth

Discussion

- Good fit in real–world use cases? User studies?
- Interactively validating/rejecting examples?
  (A: in practice, you iteratively refine the examples if the synthesizer doesn’t do what you want.)
- Applicability?
- Inside–out recursion?
- Comparisons with other tools?

Design Argument

- Person *?* in setting *Typed Functional Programming* wants to **(G)** synthesize program from examples but obstacles **(O1)** large search space get in the way. Any solution has to satisfy constraints **(X1)** types & **(X2)** examples, minimize costs **(Y1)** runtime & **(Y2)** complexity of solution, and avoid obstacles of **(Z1)** incorrect solution.

- **(A1)** “synthesis techniques have many potential applications” **(A2)** “synthesizing programs with structured data, recursion and higher-order functions in a typed programming language” is useful

- Approach MYTH has characteristics **(C1)** type-driven **(C2)** evaluate during enumeration **(C3)** prune by examples that help achieve goal **(G)** while avoiding obstacles **(O1)** & **(Z1)**.