Type-and-Example-Directed Program Synthesis

Steve Zdancewic

Joint work with
Peter-Michael Osera
Jonathan Frankle
David Walker

ExCAPE webinar 4/5/2016
Program Synthesis

Program Search + Specification

Formal methods: SMT solvers, model checking, CEGIS
[Solar-Lezama; Kneuss; Bodik, Torlak; Gulwani; ...]

[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?

- Verification
- Optimization

Program Design
“My program writes itself!”
(a.k.a. type-directed programming)

\[ t_1 \rightarrow t_2 \quad \leadsto \quad \text{let } f \ (x : t_1) : t_2 = \_ \]

\[ C \quad \leadsto \quad (g \ . \ f) \_ \]
\[ (f : A \rightarrow B, \ g : B \rightarrow C) \]

How can we mechanize this reasoning?
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”

- **Untyped Programs**: 1,949,031,274
- **Typed Programs**: 201,998
- **Typed, Normal Programs**: 21,704

Program Size (AST nodes)

- Untyped Programs
- Typed Programs
- Typed, Normal Programs
**Myth**, a program synthesizer for typed, functional programs (OCaml).

```
kabling-mobile:ynml posera$ time ./synml.native job-talk/stutter.ml --nosugar
let stutter : mylist -> mylist =
  let rec f1 (m1:mylist) : mylist =
    match m1 with
    | Nil -> Nil
    | Cons (i1, m2) -> Cons (i1, Cons (i1, f1 m2))
  in
  f1
;;
```
(One type per expression)

Input: \( \Gamma \)

Output: \( e : \tau \)

\[ \Gamma \vdash e : \tau \]

\( \Rightarrow \) Typechecking specification
Simply-typed Lambda Calculus

\[
\begin{align*}
\tau &::= \tau_1 \rightarrow \tau_2 \mid T \\
e &::= x \mid e_1 e_2 \mid \lambda x:\tau. e \mid c
\end{align*}
\]

**Types**

**Terms**

\[
\begin{align*}
\text{T-VAR} & \quad x \in \Gamma \quad \quad \Rightarrow \quad \Gamma \vdash x : \tau \\
\text{T-LAM} & \quad x:\tau_1, \Gamma \vdash e : \tau_2 \quad \quad \Rightarrow \quad \Gamma \vdash \lambda x:\tau_1.e : \tau_1 \rightarrow \tau_2 \\
\text{T-APP} & \quad \Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \quad \Gamma \vdash e_2 : \tau_1 \quad \quad \Rightarrow \quad \Gamma \vdash e_1 e_2 : \tau_2 \\
\text{T-BASE} & \quad \Gamma \vdash c : T
\end{align*}
\]
Type Checking / Inference

\[ \Gamma \vdash g : B \rightarrow C \]

\[ \Gamma \vdash f : A \rightarrow B \]

\[ \Gamma \vdash x : A \]

\[ \Gamma \vdash f \ x : B \]

\[ \Gamma \vdash f : A \rightarrow B, \ g : B \rightarrow C, \ x : A \vdash g \ (f \ x) : C \]

\[ \Gamma \vdash \lambda x : A. g \ (f \ x) : A \rightarrow C \]

\[ \Gamma \vdash \lambda g : B \rightarrow C. \ \lambda x : A. g \ (f \ x) : (B \rightarrow C) \rightarrow A \rightarrow C \]

\[ \vdash \lambda f : A \rightarrow B. \ \lambda g : B \rightarrow C. \ \lambda x : A. g \ (f \ x) : (A \rightarrow B) \rightarrow (B \rightarrow C) \rightarrow A \rightarrow C \]
(Many expressions per type)

\[ \Gamma \vdash \tau \leadsto e \]

⇒ Well-typed term *search* procedure

Derivation → Enumerated Program
Eliminating Redundancy: Normal Form Terms

\[(\text{fun } x : \text{nat} \rightarrow x + 1) \ 3 \equiv 4\]

\[\text{match Nil with}
| \text{Nil} \rightarrow 4 \equiv 4
| \text{Cons} (x, l) \rightarrow e\]

⇒ Avoid enumerating non-normal terms
\( \lambda x: \tau. \ e \)

\( C(e_1, \ldots, e_k) \)

\( x \)

\( e_1 \ e_2 \)

match \( e \) with \( p_i \to e_i^{i<m} \)
We are able to provide a value binding for each argument that we record in problems.

Recursive Functions does not affect synthesis in any significant way as we have demonstrated that our recursive functions are fully saturated, but stands in opposition to Haskell which treats a constructor constrain patterns to be of the form

Recall from Section 3.3.4 that we synthesized a (non-recursive) function by

Note that this particular formulation of constructors implies that a constructor must be fully saturated, that is, it is always provided all its arguments.

Figure 3.5:

Figure 6.2: ML

match $E$ with $\rho_i \rightarrow I_i$
(1) Type-directed **Refinement**

\[
\lambda x: \tau. \; I \\
C(I_1, \ldots, I_k)
\]

\[
x \\
E \; I
\]

**Type → Term Shape → Example Refinement.**

match $E$ with $\rho_i \rightarrow I_i^{i<^m}$
\[ \lambda x: \tau. \, I \]

\[ C(I_1, \ldots, I_k) \]

(1) Type-directed **Refinement**

(2) **Guess**-and-check

Normalize-and-compare strategy.

match \( E \) with \( \rho_i \rightarrow I_i^{i < m} \)
we are able to provide a value binding for each argument that we record in recursive functions where synthesis approach handles both tuples and partially applied functions without constrain patterns to be of the form

\[ \lambda x: \tau. I \]

\[ C(I_1, \ldots, I_k) \]

If this particular formulation of constructors implies that a constructor does not affect synthesis in any significant way as we have demonstrated that our choice has implications for the implementation of data types in these languages, it stands in opposition to Haskell which treats a constructor as a function in order to simplify our presentation without loss of expressiveness.

\[ \text{match } E \text{ with } p_i \rightarrow I_i^{i<m} \]

(1) Type-directed **Refinement**  
(2) **Guess**-and-check  
(3) **Learning** via pattern matching

Decompose data to learn more information.
\[ \lambda x: \tau. \, I \qquad C(I_1, \ldots, I_k) \qquad x \quad E \quad I \]

(1) Type-directed **Refinement**  \hspace{1cm} (2) **Guess**-and-check

Breadth-first search for valid derivations.

\[ \Rightarrow \]

Synthesize the *smallest* satisfying program.

match \( E \) with \( p_i \rightarrow I_i \) \( i \leq m \)

(3) **Learning** via pattern matching
Example values:
- Constructors
- Input/output pairs

\[
\chi ::= C(\chi_1, \ldots, \chi_k) \\
| \quad v_i \Rightarrow \chi_i^{i<n}
\]

\[\Gamma \vdash \tau \triangleright \ X \rightsquigarrow e\]

\[\Rightarrow \text{Type-and-example refinement procedure}\]

Derivation \rightarrow \text{Satisfying Program}
stutter : list -> list

\[ : list -> list \]

<table>
<thead>
<tr>
<th>Context</th>
<th>Goal Examples:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>[] =&gt; []</td>
</tr>
<tr>
<td></td>
<td>[0] =&gt; [0, 0]</td>
</tr>
<tr>
<td></td>
<td>[1, 0] =&gt; [1, 1, 0, 0]</td>
</tr>
</tbody>
</table>
let rec stutter (l:list) : list =
  □ : list

Goal Examples:
[] => []
[0] => [0, 0]
[1, 0] => [1, 1, 0, 0]
let rec stutter (l:list) : list =
    ■ : list

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

Goal Examples:
[0, 0]
[1, 1, 0, 0]

Context
l=[0], ...
l=[1, 0], ...

Goal Examples:
[0, 0]
[1, 1, 0, 0]
stutter : list -> list

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

Context
| l=[0], x=0, l'=[]
| l=[1, 0], x=1, l'=[0]

Goal Examples:
| [0, 0]
| [1, 1, 0, 0]
let rec stutter (l:list) : list =
  match l with
  | Nil -> Nil
  | Cons (x, l') ->
    Cons (x, Cons (x, try: list))

Goal Examples:

[]
[0, 0]
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]  )
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
l=[0], x=0, l'=[]
l=[1, 0], x=1, l'=[0]

Goal Examples:
[ [], [0, 0] ]
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:
[ [] => [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] → []
let rec stutter (l:list) : list =
match l with Nil ->
| Cons (x, l') ->
T-CTOR Nil
T-CTOR Cons(x, l')
T-VAR x
T-APP (stutter, l')
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.

Myth, a program synthesizer for typed, functional programs (OCaml).
Let's define a function `stutter` to efficiently implement this:

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

To implement this efficiently, we can use tail recursion to avoid stack overflow. How to achieve this?
Refinement Trees

- finite # examples
- bound on # ‘match’ clauses
⇒ determine the possible shapes of programs

```
let rec stutter (l:list) : list =
  match l with
  Nil ->
  | Cons (x, l') ->
    stutter
```
Proof Search Techniques
- Generating normal forms
- Relevant term generation
- Caching example refinements (refinement trees)
Algorithmic Insight: Relevance Logic $\Rightarrow$ Caching

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

"Generate type $\tau$ E-forms in context $\Gamma$ with size $n$"

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

- $x$ relevant (must be used)
- $x$ irrelevant (not used)
- cacheable
Implementing Relevance

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

“Generate type $\tau$ E-forms that definitely mention $x$ in context $\Gamma$ with size $n$”

$$\begin{align*}
\text{gen}_E^{x:\tau_1} (\Sigma; \Gamma; \tau; 0) &= \{\}\hspace{1cm} \\
\text{gen}_E^{x:\tau} (\Sigma; \Gamma; \tau; 1) &= \{x\}\hspace{1cm} \\
\text{gen}_E^{x:\tau_1} (\Sigma; \Gamma; \tau; 1) &= \{} \quad (\tau \neq \tau_1) \\
\text{gen}_E^{x:\tau_1} (\Sigma; \Gamma; \tau; n) &= \bigcup_{\tau_2 \rightarrow \tau \in \Gamma} \bigcup_{k=1}^{n-1} \\
&\quad \left( \text{gen}_E^{x:\tau_1} (\Sigma; \Gamma; \tau_2 \rightarrow \tau; k) \times_{\text{app}} \text{gen}_I (\Sigma; \Gamma; \tau_2; n - k) \right) \\
&\quad \bigcup \left( \text{gen}_E (\Sigma; \Gamma; \tau_2 \rightarrow \tau; k) \times_{\text{app}} \text{gen}_I^{x:\tau_1} (\Sigma; \Gamma; \tau_2; n - k) \right) \\
&\quad \bigcup \left( \text{gen}_E^{x:\tau_1} (\Sigma; \Gamma; \tau_2 \rightarrow \tau; k) \times_{\text{app}} \text{gen}_I^{x:\tau_1} (\Sigma; \Gamma; \tau_2; n - k) \right)
\end{align*}$$
• **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.
• Synthesis in larger contexts is challenging!
  – Outs: equivalences, richer types, e.g., polymorphism.

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

Free variable collector for a lambda calculus.
(31 examples, size 75, 3.9s)
\( \lambda_{syn} \), a logical foundation for program synthesis.

\[ \text{Soundness} \]

\[ \Gamma \vdash e : \tau \quad \text{(Type Soundness)} \]

\[ e \models X \quad \text{(Example Soundness)} \]
\( \lambda_{\text{syn}} \), a logical foundation for program synthesis.

**Completeness**

\[
\Gamma \vdash e : \tau \\
\Gamma \vdash X \Rightarrow \quad \Gamma \vdash \tau \triangleright X \rightsquigarrow e
\]

(Due to deciding equality between recursive functions.)
Example-Directed Synthesis: A Type-Theoretic Interpretation
[POPL 2016: Frankle, Osera, Walker, & Zdancewic]
What are "examples"?
Examples are Refinement Types

Singleton types (constructors): \([\[]\), 0, cons, ...

Function types: \(t_1 \rightarrow t_2\)

Intersection types: \(t_1 \land t_2\)

stutter : list \(\rightarrow\) list \(\land\)
\([\[] \rightarrow [\[] \land\)
\([1] \rightarrow [1,1] \land\)
\([2,1] \rightarrow [2,2,1,1]\)
More Expressive Types

Union types: 
\( t_1 \lor t_2 \)

(Limited) Negations: 
\( \sim t_1 \)

Allows for counter-example guided user interactions.

Polymorphic Types: 
\( \forall a. \ t \)

Expresses symmetries
Significantly more succinct examples (~20% shorter) & simpler specifications.
Tradeoffs in Search

\[
\text{decrement} : \text{nat} \to \text{nat} \land \\
0 \to 0 \land \\
1 \to 0 \land \\
2 \to 1
\] (in the context)

decrement(■) : goal

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th>goal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>
Argument "Sudoku"

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

(in the context)

decrement(■) : goal

<table>
<thead>
<tr>
<th>0 -&gt; 0</th>
<th>1 -&gt; 0</th>
<th>2 -&gt; 1</th>
<th>goal</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>
Argument "Sudoku"

decrement : nat -> nat \( \lor \)
\[
\begin{array}{c}
0 \rightarrow 0 \\
1 \rightarrow 0 \\
2 \rightarrow 1
\end{array}
\]
(in the context)

decrement(\(\square\)) : goal
"pick boxes then find candidates"
⇒ 12 search problems (all of which fail for constant terms)
## Argument "Sudoku"

<table>
<thead>
<tr>
<th>0 -&gt; 0</th>
<th>1 -&gt; 0</th>
<th>2 -&gt; 1</th>
<th>goal</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

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
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
References

• Introductory Type Theory:
  – *Types and Programming Languages* [Pierce]

• 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]
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.
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
  ;;