Type (and Example)
Directed Program Synthesis

Steve Zdancewic
with
Peter-Michael Osera

PENN PL
ExCAPE
Expeditions in Computer Augmented Program Engineering
Program Synthesis

Program Search + Specification

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

[Lau; Weimer; Seshia; ...]

PLDI 2015:

MYTH [Osera & Zdancewic]

$\lambda^2$ [Feser, Chaudhuri, Dillig]

Kuncak; Kuncak, Piskac, ...]
Types: What are they good for?

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

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

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

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"

1,949,031,274 Untyped Programs

201,998 Typed Programs

21,704 Typed, Normal Programs
Myth, a program synthesizer for typed, functional programs (OCaml).
(One type per expression)

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

⇒ 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**

**T-VAR**

\[
\frac{x \in \Gamma}{\Gamma \vdash x \, : \, \tau}
\]

**T-LAM**

\[
\frac{x:\tau_1, \Gamma \vdash e \, : \, \tau_2}{\Gamma \vdash \lambda x:\tau_1. e \, : \, \tau_1 \rightarrow \tau_2}
\]

**T-APP**

\[
\frac{\Gamma \vdash e_1 \, : \, \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 \, : \, \tau_1}{\Gamma \vdash e_1 e_2 \, : \, \tau_2}
\]

**T-BASE**

\[
\frac{\Gamma \vdash c \, : \, T}{\Gamma \vdash c}
\]
Type Checking / Inference

\[
\begin{align*}
g : B \rightarrow C & \in \Gamma \\
\Gamma & \vdash g : B \rightarrow C
\end{align*}
\]

\[
\begin{align*}
f : A \rightarrow B & \in \Gamma \\
\Gamma & \vdash f : A \rightarrow B
\end{align*}
\]

\[
\begin{align*}
x : A & \in \Gamma \\
\Gamma & \vdash x : A
\end{align*}
\]

\[
\begin{align*}
f : A \rightarrow B, g : B \rightarrow C, x : A & \vdash g \,(f \,x) : C \\
f : A \rightarrow B, g : B \rightarrow C & \vdash \lambda x : A. \,g \,(f \,x) : A \rightarrow C \\
f : A \rightarrow B & \vdash \lambda g : B \rightarrow C. \,\lambda x : A. \,g \,(f \,x) : (B \rightarrow C) \rightarrow A \rightarrow C \\
\vdash \lambda f : A \rightarrow B. \,\lambda g : B \rightarrow C. \,\lambda x : A. \,g \,(f \,x) : (A \rightarrow B) \rightarrow (B \rightarrow C) \rightarrow A \rightarrow C
\end{align*}
\]
(Many expressions per type)

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

⇒ 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
\[
\lambda x: \tau. \ e \\
C(e_1, \ldots, e_k)
\]

match \( e \) with \( p_i \rightarrow e_i \)\(^{i<m}\)
\[ \lambda x: \tau \cdot I \]
\[ C(I_1, \ldots, I_k) \]

match \( E \) with \( \rho_i \rightarrow I_i^{i<m} \)

\[ x \]
\[ E \]
\[ I \]

"Introduction" forms, \( I \)

"Elimination" forms, \( E \)
\[ \lambda x: \tau. \; I \quad C(I_1, \ldots, I_k) \]

(1) Type-directed **Refinement**

Type \( \rightarrow \) Term Shape \( \rightarrow \) Example Refinement.

match \( E \) with \( p_i \rightarrow I_i^{i<m} \)
\[
\lambda x : \tau. \ I \\
C(I_1, \ldots, I_k)
\]

(1) Type-directed **Refinement**

\[
x E I
\]

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

Normalize-and-compare strategy.

match \(E\) with \(p_i \rightarrow I_i^{i<m}\)
We are able to provide a value binding for each argument that we record in variables of Recursive Functions as a function order to simplify our presentation without loss of expressiveness. In addition to algebraic data types, functions must be fully saturated, that is, it is always provided all its arguments. 

Decompose data to learn more information.

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

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

match $E$ with $\rho_i \rightarrow I_i^{i<m}$

(3) **Learning** via pattern matching
\[
\lambda x : \tau. \ I \\
C(I_1, \ldots, I_k)
\]

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

Breadth-first search for valid derivations.  

\[ \Rightarrow \]

Synthesize the smallest satisfying program.  

match \( E \) with \( \rho_i \rightarrow I_i \)

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

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

⇒ Type-and-example refinement procedure
Derivation → Satisfying Program
stutter : list -> list

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

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

Example “World”
let rec stutter (l:list) : list =
  l

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

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

Goal Examples:

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

Context

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

Context
1=[]

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

Goal Examples:

Context
l=[]

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

<table>
<thead>
<tr>
<th>Context</th>
</tr>
</thead>
<tbody>
<tr>
<td>l=[0], x=0, l'=[]</td>
</tr>
<tr>
<td>l=[1, 0], x=1, l'=[0]</td>
</tr>
</tbody>
</table>

| 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]  )

<table>
<thead>
<tr>
<th>Context</th>
<th>Goal Examples:</th>
</tr>
</thead>
<tbody>
<tr>
<td>l=[0], x=0, l'=[]</td>
<td>[]</td>
</tr>
<tr>
<td>l=[1, 0], x=1, l'=[0]</td>
<td>[0, 0]</td>
</tr>
</tbody>
</table>
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] )

"Trace Completeness"

  e.g., [1, 0] → [0] → []
let rec stutter (l:list) : list =
match l with
  | Nil -> nil
  | Cons (x, l') ->
    stutter
    (Cons (x, l'))
$\lambda_{\text{syn}}$, a logical foundation for program synthesis.

Soundness

\[ \Gamma \vdash e : \tau \] (Type Soundness)

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

**Completeness**

\[
\Gamma \vdash e : \tau \\
\Gamma \vdash \tau \triangleright X \leadsto e
\]

(Due to deciding equality between recursive functions.)
\( \lambda_{\text{syn}} \), a logical foundation for program synthesis.

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

```bash
kambing-mobile:~ synml 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
;;
```
let rec stutter \( (l: \text{list}) : \text{list} = \)

match \( l \) with 

\( \text{Nil} \rightarrow \) Cons(\( x \), stutter \( (l') \) ) 

\( \text{Cons}(x, l') \rightarrow \) stutter \( (l') \)

Typechecking \( \Rightarrow \) Synthesis

How to implement this efficiently?
Refinement Trees

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

\[
\text{let rec stutter (l:list) : list =}
\begin{align*}
\text{match l with Nil -> } & \\
\text{| Cons (x, l') -> } & \\
\end{align*}
\]
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 \(n\)”

\[
\begin{align*}
\text{gen}_E^{x:\tau_1}(\Sigma; \Gamma; \tau; 0) &= \{\} \\
\text{gen}_E^{x:\tau}(\Sigma; \Gamma; \tau; 1) &= \{x\} \\
\text{gen}_E^{x:\tau_1}(\Sigma; \Gamma; \tau; 1) &= \{\} \quad (\tau \neq \tau_1)
\end{align*}
\]

\[
\text{gen}_E^{x:\tau_1}(\Sigma; \Gamma; \tau; n) = \bigcup_{\tau_2 \rightarrow \tau \in \Gamma} \bigcup_{k=1}^{n-1} \left( \text{gen}_E^{x:\tau_1}(\Sigma; \Gamma; \tau_2 \rightarrow \tau; k) \otimes_{\text{app}} \text{gen}_I(\Sigma; \Gamma; \tau_2; n-k) \right)
\]

\[
\bigcup \left( \text{gen}_E(\Sigma; \Gamma; \tau_2 \rightarrow \tau; k) \otimes_{\text{app}} \text{gen}_I^{x:\tau_1}(\Sigma; \Gamma; \tau_2; n-k) \right)
\]

\[
\bigcup \left( \text{gen}_E^{x:\tau_1}(\Sigma; \Gamma; \tau_2 \rightarrow \tau; k) \otimes_{\text{app}} \text{gen}_I^{x:\tau_1}(\Sigma; \Gamma; \tau_2; n-k) \right)
\]
• **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, \textit{e.g.}, polymorphism.

• Reigning in \#/examples requires additional info.
  – Ex. taking advantage of the program being synthesized...
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
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
  ;;
```
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)
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] ⇒ []
Research Directions / Questions

• Richer type systems:
  – Polymorphism, refinement types, dependent types

• Richer specification languages
  – Beyond I/O: fewer examples, more general constraints, negative information

• 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):
  – *Type-and-Example-Driven Program Synthesis*
    [Osera, Zdancewic 2015]
  – *Synthesizing Data Structure Transformations from Input-Output Examples*
    [Feser, Chaudhuri, Dillig 2015]
  – *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.