Type- & Example-Driven Program Synthesis

Steve Zdancewic
WG 2.8, August 2014
• Joint work with Peter-Michael Osera
CAVEATS

• Work in progress
  – Similar work been done before
  – This is our attempt to understand some of the basic issues, maybe make some advances

• We have:
  – Some theory that describes our approach
  – A couple of (incompatible, likely buggy) implementations
  – Implementations that don’t (yet) agree with all of our theory

• Feedback welcome!
  – Connections to things like Quickcheck, Agda, …?
  – Suggestions for application domains
Background: Program Synthesis

• Recent Highlights:
  – Gulwani et al. (Spreadsheets, …)
  – Solar-Lazama et al. (Program Sketching)
  – Torlak (Rosette, …)

• ExCAPE
  – Robotics control (synthesize plans)
  – Cache coherence protocols
  – Education (synthesize feedback based on buggy student code)
  – …

• Syntax-guided Synthesis (SyGus) competition
  – Surprisingly effective “brute force” enumeration of program snippets by syntax
Inductive Program Synthesis

• Summary: Use proof search to generate programs

• Old idea: 1960's, 70's, 80's
  – Application of theorem proving to problem solving. [Green 1969]
  – Synthesis: Dreams → Programs. [Manna & Waldinger 1979]
  – A deductive approach to program synthesis. [Manna & Waldinger 1980]

• More modern incarnations:
  – Haskell’s Djinn [Augustsson 2008]
  – Escher [Albarghouthi, Gulwani, Kincaid 2013]
  – Synthesis modulo recursive functions [Kuncak et al. 2013]

• Good recent survey
  – Inductive programming: A survey of program synthesis techniques. [Kitzelmann 2010]
Our Approach

- Apply ideas from intuitionistic theorem proving
  - Treat programs as proof terms
  - Search only for normal forms, not arbitrary terms
  - Use substructural logic (relevance)
- Use concrete *examples* as a partial specification
- Search for terms in order of the size of their ASTs

- Intuition / Hope:
  - Simple (i.e. small), well-typed programs that satisfy a few well-chosen tests are likely to be correct.

- Start simple
(Hopeless?) Ideal Goals

• Completeness
  – Enumerate in order of size *all* distinct programs that do not contradict the examples

• Soundness
  – Synthesized programs are well-typed
  – Synthesized programs should agree with the examples
(Realizable?) Goals

• **Completeness**
  – Enumerate in order of size (a prefix of) all programs that do not contradict the examples (after a “reasonable” amount of observation time)
  – May enumerate non-distinct (i.e. contextually equivalent) programs.

• **Soundness**
  – Synthesized programs are well-typed
  – Synthesized programs (if they terminate in a “reasonable” time) should agree with the examples
Simplifications (For Now)

- Pure (except for divergence), functional programs

- Simple, algebraic types and higher-order functions only
  - No polymorphism (though this would strongly constrain search)
  - Monomorphic programs are still interesting

- Specification via examples, not logical properties
  - Good starting point
  - Probably not sufficient in the long run

- Future work: relax these simplifications
(Simple) Target Language

\[ e ::= x \mid e_1 e_2 \mid \text{fix } f \; x. \; e \mid \text{ctr}^n_b \; e_1 \ldots e_n \]

match e with \[ \text{pat}_i \rightarrow e_i \]

- Recursive, algebraic datatypes
- Arbitrary recursion
- Standard (monomorphic) type system
Proof System for Normal Forms

- Factor terms into intro and elim forms:

\[
E \ ::= \ x \ I_1 .. I_n \\
I \ ::= \ E \mid \text{fix } f \ x. \ I \mid \text{ctr}_b^I \ I_1 .. I_n \\
\mid \text{match } E \text{ with } \text{pat}_i \rightarrow I_i^i
\]

- Inference rules enforce the separation:

\[
\begin{align*}
\Gamma \vdash \ I_1 : t_1 & \quad \ldots \quad \Gamma \vdash \ I_n : t_n \\
\Gamma \vdash \ x \ I_1 .. I_n : b \\
\Gamma \vdash \ E : b & \quad \text{APP} \\
\Gamma \vdash \text{fix } f \ x. \ I : t_1 \rightarrow t_2 \\
\Gamma \vdash \ I : t_2 & \quad \text{FIX} \\
\Sigma (\text{ctr}_b^i) = t_1 \rightarrow \ldots \rightarrow t_n \rightarrow b & \quad \text{CTR} \\
\Gamma \vdash \ I_1 : t_1 & \quad \ldots \quad \Gamma \vdash \ I_n : t_n \\
\Gamma \vdash \text{ctr}_b^i \ I_1 .. I_n : b \\
\Gamma \vdash \ E : b & \quad \text{ELIM} \\
\Gamma \vdash \ E : b & \quad \Gamma \vdash \text{pat}_i : b \Rightarrow \Gamma_i^i \\
\Gamma_i \vdash \ I_i : t_i & \quad \text{MATCH} \\
\Gamma \vdash \text{match } E \text{ with } \text{pat}_i \rightarrow I_i^i : t
\end{align*}
\]
Strategies for Enumeration

- **Representation:**
  - hash-consed locally nameless (closed = Debruijn)
  - terms keep track of their free variables (makes closing/substitution faster)

- **Memoize the generation functions**

```ocaml
let gen_elim (s : Sig.t) (g : GenCtxt.t) (goal_t : typ) (size : int) : elim Rope.t = ...
and gen_intro (s : Sig.t) (g : GenCtxt.t) (goal_t : typ) (size : int) : intro Rope.t = ...
```

- **Relevance logic:**
  - Fix and match introduce new variable bindings to the context: \( G, x : u \vdash E : t \)
  - Memoization won’t work (the context changes)
  - Split the judgment into two parts
    - General rule that uses context arbitrarily
    - A “relevance” rule that requires a particular variable to be used at least once
    - Original rule recovered by:\( G, x : u \vdash E : t \) = \( G \vdash E : t \) + \( G, x : u > \vdash E : \)
Strategies for Pruning

- Eliminate "redundant" matches:

```
match x with
| 0 -> match x with ...
| S y -> ...
```

- Prune matches with redundant branches:

```
match x with
| 0 -> e = e
| S y -> e
```

- Question: How much impact does moving from lambda to fix have?
(Super) Exponential Growth

- Closed normal terms of type `nat -> nat`
- Nodes in AST

<table>
<thead>
<tr>
<th>#nodes in AST</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>nat -&gt; nat</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>16</td>
</tr>
</tbody>
</table>

Graph showing the exponential growth with different lines indicating different conditions.
Pushing Examples Around

• Extend the language grammar with examples
  – Examples are first-class values
  – They can be given types
  – At function type, consist of input/output pairs:

```
{ sum  => ( 0  => (  []  =>  0
|    [1]  =>  1
|    [2; 1]  =>  3 ) )
| incr  => ( 0  => (  []  =>  0
|    [1]  =>  1
|    [2; 1]  =>  2 ) ) ]
```

• “math” notation: \( X, \text{ex} ::= \{ \cdot v_1 v_2 v_3 = v, \cdot u_1 u_2 u_3 = u, \ldots \} \)

  e.g. \{ \cdot \text{sum} 0 [ ] = 0, \cdot \text{sum} 0 [1] = 1, \ldots, \}
Adding Examples to Typechecking

**Synthesis contexts**

\[ \Theta ::= \cdot \quad \Theta, \, x : t \triangleright X \]

**Old: Constructors without examples**

\[
\begin{align*}
\Sigma (ctr^k_b) &= t_1 \to \ldots \to t_n \to b \\
\Gamma \vdash I_1 : t_1 & \quad \ldots \\
\Gamma \vdash I_n : t_n \\
\hline
\Gamma \vdash ctr^k_b I_1 \ldots I_n : b
\end{align*}
\]

(wfI_CTR)

**New: Constructors with examples**

\[
\begin{align*}
\Sigma (c) &= t_1 \to \ldots \to t_n \to b \\
X &= \{ ctr^u_j \overline{w_{j1}}^{j \in 1 \ldots n}, \ ctr^u_j \overline{w_{jk}}^{j \in 1 \ldots n} \} \\
\Theta \vdash ? : t_1 \triangleright \{ w_{11}, \ldots, w_{1k} \} & \sim I_1 \\
\quad \ldots \\
\Theta \vdash ? : t_n \triangleright \{ w_{n1}, \ldots, w_{nk} \} & \sim I_n \\
\Theta \vdash ? : b \triangleright X & \sim ctr^u_b I_1 \ldots I_n
\end{align*}
\]

(synthI_Ctr)
Pushing Examples Through Functions

Old: Functions without examples

\[
\Gamma,f : t_1 \rightarrow t_2, x : t_1 \vdash I : t_2
\]

\[
\Gamma \vdash \text{fix} \ f \ x. \ I : t_2
\]

\[\text{wfl}_{-\text{Fix}}\]

New: Functions with examples

\[
X = \{w_{n1} = w_1, \ldots, w_{nk} = w_k\}
\]

\[
\Theta' = \Theta, f : t_1 \rightarrow t_2 \triangleright X^k, x : t_2 \triangleright \{w_{11}, \ldots, w_{1k}\}
\]

\[
\Theta' \vdash ? : t_2 \triangleright \{w_{2i} = w_i \mid i \in 1..k\} \sim I
\]

\[\Theta \vdash ? : t_1 \rightarrow t_2 \triangleright X \sim \text{fix} \ f \ x. \ I\]

\[\text{synthI}_{-\text{Fun}}\]
Examples through Elim Forms

\[ x : t_1 \rightarrow \ldots \rightarrow t_n \rightarrow b \in \Gamma \]
\[ \Gamma \vdash ? : t_1 \triangleright \{ \} \leadsto I_1 \]
\[ \vdots \]
\[ \Gamma \vdash ? : t_n \triangleright \{ \} \leadsto I_n \]
\[ \Gamma \vdash ? : b \leadsto xI_1 \ldots I_n \]  

\text{SYNTHÉ_APP}

\[ [\Theta] \vdash ? : b \leadsto xI_1 \ldots I_n \]
\[ \Theta_i \vdash xI_1 \ldots I_n \leadsto X_i : b_{i \in 1..k} \]
\[ \Theta'(k) \vdash ? : b \triangleright X_{(k)} \leadsto xI_1 \ldots I_n \]  

\text{SYNTHI_APP}

New: Compatibility requirement – application must respect the provided examples.
Compatibility

- Evaluator: an abstract interpreter for the nonstandard language
- + approximation to equivalence.

\[
\begin{align*}
\Theta(E) &= e & \vdash e \sim ex \\
\Theta &\vdash E \sim ex : b & \text{COMPAT}_\Theta \text{_COMPAT} \\
\vdash e \downarrow w & w \sim ex \\
\vdash e \sim ex & \text{SAT}_E
\end{align*}
\]

- See inference rules.
Heuristics

- May compromise completeness, but can greatly reduce search space.

- Maximum number of evaluation steps for compatibility checking.
  - Prevents infinite loops
  - May miss correct programs

- Size restrictions

- Limit recursion to “well-behaved” subsets:
  - e.g. structural recursion

- For the demo: Stop at first “good” program
Conclusions / Future

• Program synthesis is experiencing a resurgence.
  – Some old ideas are new again
• Fun to think about automatic program generation.
  – Many limitations too: sensitivity to particular examples

• Future work:
  – Experiments:
    • i.e. can’t yet measure impact of “example pushing” on size of search space
  – Think about richer ways to “push” example information through the search.
    • might require “negative” constraints
  – Thing about richer specifications
    • something like Quickcheck properties
    • suites of related functions
  – Polymorphism? Dependency?
  – Interactivity?
  – Connect to other kinds of work (e.g. SMT-solver based approaches)