Type- & Example-Driven Program Synthesis

Steve Zdancewic

WG 2.8, August 2014

- Joint work with Peter-Michael Osera

ExCAPE

Expeditions in Computer Augmented Program Engineering

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)
  - …

ExCAPE
Expeditions in Computer Augmented Program Engineering

- 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]

DEMO

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

$$\begin{array}{lll} e & ::= & x \mid e_1\ e_2 \mid \mathsf{fix}\ f\ x.\ e \mid \mathsf{ctr}^n_{\mathsf{b}}\ e_1\ ..\ e_n \\ & \mid & \mathsf{match}\ e\ \mathsf{with}\ \overline{pat_i \to e_i}^{\,i} \end{array}$$

- Recursive, algebraic datatypes
- Arbitrary recursion
- Standard (monomorphic) type system

Proof System for Normal Forms

- Factor terms into intro and elim forms:

$$\begin{array}{lcll} E & ::= & x\, I_1 \,..\, I_n & \text{elimination forms} \\ I & ::= & E \mid \mathsf{fix}\, f\ x.\ I \mid \mathsf{ctr}^n_{\mathsf{b}}\, I_1 \,..\, I_n & \text{introduction forms} \\ & \mid & \mathsf{match}\ E\ \mathsf{with}\ \overline{pat_i \to I_i}^{\,i} & \end{array}$$

- Inference rules enforce the separation:

$$\frac{\begin{array}{c} x : t_1 \to \,..\, \to t_n \to \mathsf{b} \in \Gamma \\ \Gamma \vdash I_1 : t_1 \quad .. \quad \Gamma \vdash I_n : t_n \end{array}}{\Gamma \Vdash x\, I_1 \,..\, I_n : \mathsf{b}}\ \textsc{App} \qquad \frac{\Gamma, f : t_1 \to t_2, x : t_1 \vdash I : t_2}{\Gamma \vdash \mathsf{fix}\, f\ x.\ I : t_1 \to t_2}\ \textsc{Fix}$$

$$\frac{\Gamma \Vdash E : \mathsf{b}}{\Gamma \vdash E : \mathsf{b}}\ \textsc{Elim} \qquad \frac{\begin{array}{c} \Sigma\,(\mathsf{ctr}^k_{\mathsf{b}}) = t_1 \to \,..\, \to t_n \to \mathsf{b} \\ \Gamma \vdash I_1 : t_1 \quad .. \quad \Gamma \vdash I_n : t_n \end{array}}{\Gamma \vdash \mathsf{ctr}^k_{\mathsf{b}}\, I_1 \,..\, I_n : \mathsf{b}}\ \textsc{Ctr}$$

$$\frac{\Gamma \Vdash E : \mathsf{b} \quad \overline{\Gamma \vdash pat_i : \mathsf{b} \Rightarrow \Gamma_i}^{\,i} \quad \overline{\Gamma_i \vdash I_i : t}^{\,i}}{\Gamma \vdash \mathsf{match}\ E\ \mathsf{with}\ \overline{pat_i \to I_i}^{\,i} : t}\ \textsc{Match}$$

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

```
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: $G, x{:}u \vdash E : t \quad = \quad G \vdash E : t \quad + \quad G, \langle x{:}u \rangle \vdash E{:}$

Strategies for Pruning

- Eliminate “redundant” matches:

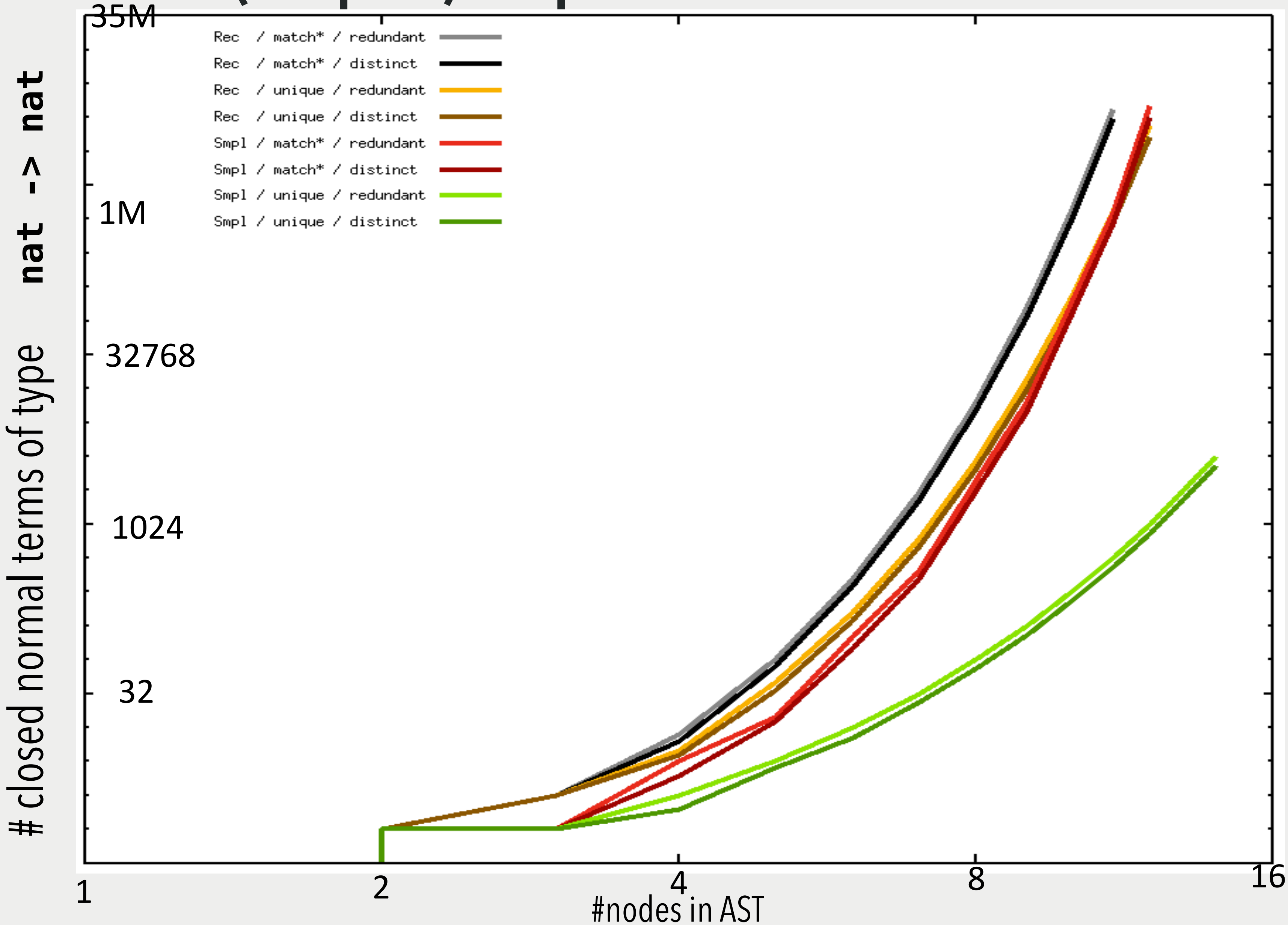
```
match x with
| O -> match x with ...
| S y -> ...
```

- Prune matches with redundant branches:

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

- Question: How much impact does moving from lambda to fix have?

(Super) Exponential Growth



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: $X, ex ::= \{ \cdot\, v_1\, v_2\, v_3 = v, \quad \cdot\, u_1\, u_2\, u_3 = u, \ldots \}$

e.g. $\{ \cdot \text{ sum } 0\, [\,] = 0, \cdot \text{ sum } 0\, [1] = 1, \ldots, \}$

Adding Examples to Typechecking

$$\begin{array}{lll} \Theta & ::= & \\ & | & \cdot \\ & | & \Theta,\ x{:}t \rhd X \end{array}$$

Synthesis contexts

$$\frac{\Sigma\,(\mathsf{ctr}^k_{\mathsf{b}}) = t_1 \to\ ..\ \to t_n \to \mathsf{b} \qquad \Gamma \vdash I_1 : t_1 \quad ..\quad \Gamma \vdash I_n : t_n}{\Gamma \vdash \mathsf{ctr}^k_{\mathsf{b}}\, I_1\, ..\, I_n : \mathsf{b}}\ \textsc{wfI\_Ctr}$$

Old: Constructors without examples

$$\frac{\begin{array}{l}\Sigma\,(c) = t_1 \to\ ..\ \to t_n \to \mathsf{b} \\ X = \{\mathsf{ctr}^n_{\mathsf{T}}\ \overline{w_{j1}}^{j\in 1..n}\ ..\ \mathsf{ctr}^n_{\mathsf{T}}\ \overline{w_{jk}}^{j\in 1..n}\} \\ \Theta \vdash ? : t_1 \rhd \{w_{11}, .., w_{1k}\} \rightsquigarrow I_1 \\ \ldots \\ \Theta \vdash ? : t_n \rhd \{w_{n1}, .., w_{nk}\} \rightsquigarrow I_n\end{array}}{\Theta \vdash ? : \mathsf{b} \rhd X \rightsquigarrow \mathsf{ctr}^n_{\mathsf{b}}\, I_1\, ..\, I_n}\ \textsc{synthI\_Ctr}$$

New: Constructors with examples

Pushing Examples Through Functions

$$\frac{\Gamma, f : t_1 \to t_2, x : t_1 \vdash I : t_2}{\Gamma \vdash \mathsf{fix}\, f\ x.\ I : t_1 \to t_2} \quad \textsc{wfI\_Fix}$$

Old: Functions without examples

$$\frac{\begin{array}{l} X = \{\cdot w_{11} \,..\, w_{n1} = w_1 \ \ ..\ \ \cdot w_{1k} \,..\, w_{nk} = w_k\} \\ \Theta' = \Theta,\ f : t_1 \to t_2 \rhd X^k,\ x : t_2 \rhd \{w_{11}, .., w_{1k}\} \\ \Theta' \vdash ? : t_2 \rhd \{\ \overline{\cdot w_{2i} \,..\, w_{ni} = w_i}^{\,i \in 1..k}\ \} \rightsquigarrow I \end{array}}{\Theta \vdash ? : t_1 \to t_2 \rhd X \rightsquigarrow \mathsf{fix}\, f\ x.\ I} \quad \textsc{synthI\_Fun}$$

New: Functions with examples

Examples through Elim Forms

$$\frac{\begin{array}{l} x : t_1 \to .. \to t_n \to \mathsf{b} \in \Gamma \\ \lceil \Gamma \rceil \vdash ? : t_1 \rhd \{\} \leadsto I_1 \\ \ldots \\ \lceil \Gamma \rceil \vdash ? : t_n \rhd \{\} \leadsto I_n \end{array}}{\Gamma \Vdash ? : \mathsf{b} \leadsto x\, I_1 \mathinner{..} I_n} \quad \textsc{synthE\_App}$$

$$\frac{\begin{array}{c} \lfloor \Theta \rfloor \Vdash ? : \mathsf{b} \leadsto x\, I_1 \mathinner{..} I_n \\ \overline{\Theta|_i \Vdash x\, I_1 \mathinner{..} I_n \sim X|_i : \mathsf{b}}^{\,i \in 1..k} \end{array}}{\Theta_{(k)} \vdash ? : \mathsf{b} \rhd X_{(k)} \leadsto x\, I_1 \mathinner{..} I_n} \quad \textsc{synthI\_App}$$

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

Compatibility

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

$$\frac{\Theta(E) = e \quad \Vdash e \sim ex}{\Theta \Vdash E \sim ex : \mathsf{b}} \quad \textsc{compat}E\_\textsc{compat}$$

$$\frac{\vdash e \Downarrow w \quad w \sim ex}{\Vdash e \sim ex} \quad \textsc{sat}\_E$$

- 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)