Type-and-Example-Directed Program Synthesis

Peter-Michael Osera   Steve Zdancewic

University of Pennsylvania, USA
{posera, stevez}@cis.upenn.edu

Abstract

This paper presents an algorithm for synthesizing recursive functions that process algebraic datatypes. It is founded on proof-theoretic techniques that exploit both type information and input–output examples to prune the search space. The algorithm uses *refinement trees*, a data structure that succinctly represents constraints on the shape of generated code. We evaluate the algorithm by using a prototype implementation to synthesize more than 40 benchmarks and several non-trivial larger examples. Our results demonstrate that the approach meets or outperforms the state-of-the-art for this domain, in terms of synthesis time or attainable size of the generated programs.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory—Semantics; F.4.1 [*Mathematical Logic and Formal Languages*]: Mathematical Logic—Proof Theory; I.2.2 [*Artificial Intelligence*]: Automatic Programming—Program Synthesis

General Terms Languages, Theory

Keywords Functional Programming, Proof Search, Program Synthesis, Type Theory

1. Synthesis of Functional Programs

This paper presents a novel technique for synthesizing purely functional, recursive programs that process algebraic datatypes. Our approach refines the venerable idea [9, 17] of treating program synthesis as a kind of proof search: rather than using just type information, our algorithm also uses concrete input–output examples to dramatically cut down the size of the search space. We exploit this extra information to create a data structure, called a refinement tree, that enables efficient synthesis of non-trivial programs.

Figure 1 shows a small example of this synthesis procedure in action. For concreteness, our prototype implementation uses OCaml syntax, but the technique is not specific to that choice. The inputs to the algorithm include a type signature, the definitions of any needed auxiliary functions, and a synthesis *goal*. In the figure, we define `nat` (natural number) and `list` types without any auxiliary functions. The goal is a function named `stutter` of type `list -> list`, partially specified by a series of input–output examples given after the `|>` marker, evocative of a refinement of the goal type. The examples suggest that `stutter` should produce a list that duplicates each element of the input list. The third example, for instance, means that `stutter [1;0]` should yield `[1;1;0;0]`.

```
(* Type signature for natural numbers and lists *)
type nat =              type list =
  | O                     | Nil
  | S of nat              | Cons of nat * list
(* Goal type refined by input/output examples *)
let stutter : list -> list |>
  { []    => []
  | [0]   => [0;0]
  | [1;0] => [1;1;0;0]
  } = ?
```

```
(* 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
```

Figure 1. An example program synthesis problem (above) and the resulting synthesized implementation (below).

The bottom half of Figure 1 shows the output of our synthesis algorithm which is computed in negligible time (about 0.001s). Here, we see that the result is the "obvious" function that creates two copies of each `Cons` cell in the input list, stuttering the tail recursively via the call to `f1 l2`.

General program synthesis techniques of this kind have many potential applications. Recent success stories utilize synthesis in many scenarios: programming spreadsheet macros by example [11]; code completion in the context of large APIs or libraries [12, 19]; and generating cache-coherence protocols [26], among others. In this paper, we focus on the problem of synthesizing programs involving structured data, recursion, and higher-order functions in typed programming languages—a domain that largely complements those mentioned above.

The Escher system (by Albarghouthi *et al.* [1]) and the Leon system (by Kneuss *et al.* [15]) inspires this work and also tackle problems in this domain, but they do so via quite different methods. Compared to the previous work, we are able to synthesize higher-order functions like `map` and `fold`, synthesize programs that use higher-order functions, and work with large algebraic datatypes.

PLDI'15, June 13–17, 2015, Portland, OR, USA
ACM. 978-1-4503-3468-6/15/06
http://dx.doi.org/10.1145/2737924.2738007

Combining Types and Examples As a proof-search-based synthesis technique, our algorithm relies crucially on the type structure of the programming language to avoid generating ill-typed terms. It also uses the proof-theoretic idea of searching only for programs in β-*normal*, η-*long* form [6], a technique also used by Gvero *et al.* [12].

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."

Intuitively this strategy is important when searching for programs that construct algebraic datatypes for the simple reason that it cuts down the space combinatorially. In the `stutter` example, suppose that we're searching for program terms to fill the `Cons` branch of the **`match`**, which must be of `list` type. One set of expressions of this type has the form `Cons(e1, e2)`, where `e1` is a term of type `nat` and `e2` is a term of type `list`. If there are N_1 well-typed candidates for `e1` and N_2 candidates for `e2`, following the naïve strategy of "enumerate *then* evaluate" would call our interpreter on all $N_1 \times N_2$ possibilities. The smarter strategy of "evaluate as you enumerate" decomposes the problem into two *independent* sub-problems which finds the same answer using only $N_1 + N_2$ calls to the interpreter. The challenge is to push examples not just through constructors, but also through other expressions as well.

Roadmap After walking through the synthesis process in more detail by example, we introduce an ML-like type system that incorporates input–output examples to constrain the set of legal program derivations. The type system can be thought of as a non-deterministic specification of the synthesis problem, where the goal is to find a term that has a valid typing derivation. To turn the rules into an algorithm, we observe that the type system decomposes naturally into two pieces: a data structure, called the *refinement tree*, that reifies the parts of the search space that are shared among many candidate solutions, and an enumerative search that fills in the remaining holes.

We then discuss our prototype implementation of this synthesis procedure: a tool called MYTH, which synthesizes code for a subset of OCaml. We used this prototype to evaluate the effectiveness of the approach on more than 40 benchmarks, and several non-trivial larger examples. Our experiments show that our prototype meets or out-performs the state-of-the-art for synthesis problems in this domain in terms of synthesis time or attainable size of the generated programs.

Throughout our discussion, we demonstrate the close connection of our synthesis algorithm to the analysis of normal form proofs in constructive logic. One consequence of this connection is that we can exploit existing ideas and algorithms from this literature: for instance, we use the idea of *proof relevance* [3] to optimize our implementation (Section 4). Moreover, the type-theoretic foundation developed here may extend naturally to richer type systems—polymorphic types for generic programs or linear types for stateful or concurrent programs.

Companion Technical Report and Source Code In the interest of brevity, we omit the presentation of non-essential components of our calculus and implementation as well as proofs. Links to both our expanded technical report with a full account of our synthesis systems and complete source code to the implementation and examples can be found on the authors' websites.

2. Type-and-Example-Directed Synthesis

Consider a core ML-like language featuring algebraic data types, **`match`**, top-level function definitions, and explicitly recursive functions. A synthesis problem is defined by: (1) the data type definitions and top-level let-bindings, (2) a goal type, and (3) a collection of examples of the goal type. The synthesis task is to find a program of the goal type that is consistent with the examples.

Type-directed synthesis performs two major operations: *refining* the constraints—the goal type and examples—and *guessing* a term of the goal type that is consistent with the example values.

Type Refinement As an example, consider searching for solutions to the `stutter` example of Figure 1. Already from the goal type `list -> list`, we know that the top-level structure must be of the form **`let rec`** `f1 (l1:list) : list = ?`. When synthesizing the body of the function, the three examples tell us that in the "world" (a hypothetical evaluation of `stutter`) where `l1` is `[]` the expected answer is `[]`, in the world where `l1` is `[0]` the expected answer is `[0;0]`, and finally in the world where `l1` is `[1;0]` the expected answer is `[1;1;0;0]`. So when synthesizing the body, each input–output pair is *refined* into a new example "world" where the output value is the goal and `l1` is bound to the corresponding input value.

Guessing To fill in the body of the function, which must be a term of type `list`, we observe that no single list constructor agrees with the examples (since there are examples starting with both `Nil` and `Cons`). At this point, we can try to enumerate or *guess* well-typed terms that involve variables to fill the body, *e.g.*, `l1`. However, `l1` does not agree with all of the examples; for example, in the world where `l1` is `[0]`, we instead require `l1` to be `[0;0]`. Other terms are either ill-typed or are well-typed but not structurally recursive (a restriction of the system we discuss further in Section 3.4) so guessing fails at this point.

Note that while we rule out ill-typed and non-structurally recursive terms, there are still many other well-typed terms that we also wish to avoid. For example, a term that matches against a constant constructor, like **`match`** `Nil` **`with`** `Nil -> e1 | Const -> e2` is equivalent to a smaller term that has no **`match`** (in this case `e1`). And to make matters worse, there are an infinite number of these bad terms, obtainable by introducing additional lambdas and applications. To avoid these sorts of redundant terms, guessing only generates β-*normal forms*: terms that can be reduced no further.

Match Refinement With these possibilities exhausted, we next consider introducing a **`match`** expression. We first guess possible scrutinees to **`match`** against: they must have algebraic types and be terms constructed from variables in the context. The only such term in this case is `l1`, so we consider how to complete the term:

```
match l1 with Nil -> ? | Cons(n1, l2) -> ?
```

When pattern matching, we distribute the examples according to how the **`match`** evaluates in each example world. In this case, `l1` evaluates to `Nil` in the first world and `Cons(_,_)` in the other two worlds. Therefore, we send the first example to the `Nil` branch and other two examples to the `Cons` branch. We are left with two sub-goals in refined worlds. The `Nil` case follows immediately: we can now guess the expression `l1` which satisfies the example world `[]` (because `l1` has value `[]` in this world).

Recursive Functions The `Cons` case proceeds with another round of guessing, but the solution requires a recursive call to `f1`. How might we generate a recursive function in this type-directed, example-based style? The answer is that when introducing a recursive function like `f1`, the input-output examples for the function itself, interpreted as a partial function, serve as a reasonable approximation of its behavior. Here, the example for `f1` in all three worlds would be the partial function given by all three examples. Given

$$
\begin{array}{lcl}
\tau & ::= & \tau \mid \tau_1 \to \tau_2 \\
e & ::= & x \mid C(e_1, .., e_k) \mid e_1\ e_2 \\
 & \mid & \mathsf{fix}\, f(x : \tau_1) : \tau_2 = e \mid pf \\
 & \mid & \mathsf{match}\ e\ \mathsf{with}\ p_1 \to e_1 \mid .. \mid p_m \to e_m \\
p & ::= & C(x_1, .., x_n) \\
u, v & ::= & C(v_1, .., v_k) \mid \mathsf{fix}\, f(x : \tau_1) : \tau_2 = e \mid pf \\
ex & ::= & C(ex_1, .., ex_k) \mid pf \\
pf & ::= & v_1 \Rightarrow ex_1 \mid .. \mid v_m \Rightarrow ex_m \\
E & ::= & x \mid E\ I \\
I & ::= & E \mid C(I_1, .., I_m) \mid \mathsf{fix}\, f(x : \tau_1) : \tau_2 = I \\
 & \mid & \mathsf{match}\ E\ \mathsf{with}\ p_1 \to I_1 \mid .. \mid p_m \to I_m \\
\Gamma & ::= & \cdot \mid x : \tau, \Gamma \\
\Sigma & ::= & \cdot \mid C : \tau_1 * .. * \tau_n \to T, \Sigma \\
\sigma & ::= & \cdot \mid [v/x]\sigma \\
X & ::= & \cdot \mid \sigma \mapsto ex +\!\!+\ X
\end{array}
$$

Figure 2. λ_{syn}: syntax.

this information in the context, the guessing procedure can quickly determine that `Cons(n1, Cons(n1, f1 l2))` is a solution for the `Cons` branch of the match.

3. Synthesis Language

With a better intuition of how type-directed synthesis works, let's examine the procedure in detail. The formalism λ_{syn} that we develop for this purpose is a sound, yet highly non-deterministic synthesis procedure. By using this formalism as a starting point, we are able to make explicit the connection between program synthesis and proof search.

3.1 λ_{syn}: A Type-Directed Synthesis Language

Figure 2 gives the syntax of λ_{syn}. As described in Section 2, our target is a core ML-like language featuring algebraic data types and recursive functions. Consequently, types are user-defined algebraic data types T and function types $\tau_1 \to \tau_2$. The syntax of expressions e is standard: C ranges over data type constructors, application is written $e_1\ e_2$, and we use ML-style pattern match expressions in which each pattern p binds the subcomponents of the constructor C. We require that every match is complete; there exists one branch for each constructor of the data type being matched. A recursive function f is given by $\mathsf{fix}\, f(x : \tau_1) : \tau_2 = e$, and we write $\lambda x : \tau_1 . e$ instead when f is not among the free variables of e. Signatures Σ contain the set of declared constructors, and contexts Γ contain information about the free variables of the program.

The only new syntactic form is that of partial functions pf, written $\overline{v_i \Rightarrow ex_i}^{i<m}$. Here, and throughout the paper, the overbar notation $\overline{e_i}^{i<m}$ denotes a sequence of m syntactic elements, $e_1..e_m$. Thus, the notation for a partial function $\overline{v_i \Rightarrow ex_i}^{i<m}$ denotes a list of input–output examples $v_1 \Rightarrow ex_1 \mid .. \mid v_m \Rightarrow ex_m$. We interpret such partial functions to mean that if we supply the argument v_i then the partial function produces the result ex_i.

Synthesis problems are specified through example values ex that are made up of constructor values (for data types) and partial functions (for arrow types). Explicit use of fix values as examples is not permitted because that would amount to supplying a definition for the function being synthesized. However, inputs to example partial functions are allowed to be values, which is useful when giving examples of higher-order functions. The syntax of input–output examples reflects this restriction where the domain of a partial function is unrestricted values v and the range is restricted to be example values ex.

To keep track of the evolution of variable values during the synthesis process, we tie to each ex an execution environment σ which maps variables to values. We can think of σ as a *substitution* where $\sigma(e)$ denotes the expression obtained by applying the substitution σ to e. A pair $\sigma \mapsto ex$ constitutes an *example world* we described in Section 2, and X is a list of example worlds, shortened to just "examples" when we don't need to distinguish between individual environments and example values. The goal of the synthesis procedure is to derive a program that satisfies each example world given in X.

Because there are many list syntactic forms in the formalism, we use a metavariable convention for their lengths: k for constructor arity, n for the number of examples $|X|$, and m for the number of cases in a match expression or partial function. We use i and j for generic indices.

3.2 Typechecking and Synthesis

Normal Forms Rather than synthesizing e terms directly, we instead restrict synthesized programs to β-normal form [6]. Such terms do not contain any β-redexes, ruling out redundant programs that pattern match against known constructors (and hence have dead branches) or that apply a known function. To do this, we split the syntax of expressions into introduction forms I and elimination forms E. Note that every E is of the form $x\ I_1 \ldots I_k$, with a free variable x at its head.

This syntactic split carries through into the type system, where, as in bidirectional typechecking [21], the rules make explicit when we are checking types (I-forms) versus generating types (E-forms), respectively.[1] 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.

Typechecking for λ_{syn} is divided among four judgments:

$\Sigma ; \Gamma \vdash e : \tau$	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.

Figure 3 gives the definition of the last three judgments. Because the typechecking judgment for regular expressions is both standard and immediately recoverable from the bidirectional typechecking system, we omit its rules here. The only new rule typechecks partial functions:

$$
\frac{\overline{\Sigma ; \Gamma \vdash v_i : \tau_1}^{i<n} \qquad \overline{\Sigma ; \Gamma \vdash ex_i : \tau_2}^{i<n}}{\Sigma ; \Gamma \vdash \overline{v_i \Rightarrow ex_i}^{i<n} : \tau_1 \to \tau_2} \ \text{T\_PF}
$$

A partial function is well-typed at $\tau_1 \to \tau_2$ if its domain typechecks at τ_1 and its range typechecks at τ_2.

The final judgment typechecks examples X at some goal type τ. This amounts to typechecking each example world $\sigma \mapsto ex \in X$ which ensures that:

1. The environment is well-typed, written $\Sigma ; \Gamma \vdash \sigma$ and
2. The example value is well-typed at type τ.

Figure 4 describes our synthesis system as a non-deterministic relation where complete derivations correspond to synthesized programs. This relation is broken up into two judgments:

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

[1] This fact is why `match` is an I-form: when typechecking a `match` we *check* the types of the branches against a given type.

$$\boxed{\begin{array}{c}\Sigma\,;\,\Gamma \vdash E \Rightarrow \tau\\ \Sigma\,;\,\Gamma \vdash I \Leftarrow \tau\end{array}}$$

T_EVAR

$$\frac{x : \tau \in \Gamma}{\Sigma\,;\,\Gamma \vdash x \Rightarrow \tau}$$

T_EAPP

$$\frac{\begin{array}{c}\Sigma\,;\,\Gamma \vdash E \Rightarrow \tau_1 \to \tau_2\\ \Sigma\,;\,\Gamma \vdash I \Leftarrow \tau_1\end{array}}{\Sigma\,;\,\Gamma \vdash E\,I \Rightarrow \tau_2}$$

T_IMATCH

$$\frac{\begin{array}{l}\Sigma\,;\,\Gamma \vdash E \Rightarrow T \qquad \overline{\mathbf{binders}\,(\Sigma, p_i) = \Gamma_i}^{\,i<m}\\ \overline{\Sigma\,;\,\Gamma_i +\!\!+\, \Gamma \vdash I_i \Leftarrow \tau}^{\,i<m}\end{array}}{\Sigma\,;\,\Gamma \vdash \mathsf{match}\ E\ \mathsf{with}\ \overline{p_i \to I_i}^{\,i<m} \Leftarrow \tau}$$

T_IELIM

$$\frac{\Sigma\,;\,\Gamma \vdash E \Rightarrow \tau}{\Sigma\,;\,\Gamma \vdash E \Leftarrow \tau}$$

T_IFIX

$$\frac{\Sigma\,;\,f : \tau_1 \to \tau_2, x : \tau_1, \Gamma \vdash I \Leftarrow \tau_2}{\Sigma\,;\,\Gamma \vdash \mathsf{fix}\,f\,(x : \tau_1) : \tau_2 = I \Leftarrow \tau_1 \to \tau_2}$$

T_ICTOR

$$\frac{\begin{array}{l}C : \tau_1 * \,..\, * \tau_k \to T \in \Sigma\\ \Sigma\,;\,\Gamma \vdash I_1 \Leftarrow \tau_1 \qquad .. \qquad \Sigma\,;\,\Gamma \vdash I_k \Leftarrow \tau_k\end{array}}{\Sigma\,;\,\Gamma \vdash C(I_1, \,..\,, I_k) \Leftarrow T}$$

$$\boxed{\begin{array}{c}\Sigma\,;\,\Gamma \vdash \sigma\\ \Sigma\,;\,\Gamma \vdash X : \tau\end{array}}$$

TENV_EMPTY

$$\frac{}{\Sigma\,;\,\Gamma \vdash \cdot}$$

TENV_CONS

$$\frac{\begin{array}{l}x : \tau \in \Gamma \qquad \Sigma\,;\,\Gamma \vdash v : \tau\\ \Sigma\,;\,\Gamma \vdash \sigma\end{array}}{\Sigma\,;\,\Gamma \vdash [v/x]\sigma}$$

TEX_EMPTY

$$\frac{}{\Sigma\,;\,\Gamma \vdash \cdot : \tau}$$

TEX_CONS

$$\frac{\begin{array}{l}\Sigma\,;\,\Gamma \vdash \sigma \qquad \Sigma\,;\,\Gamma \vdash ex : \tau\\ \Sigma\,;\,\Gamma \vdash X : \tau\end{array}}{\Sigma\,;\,\Gamma \vdash \sigma \mapsto ex +\!\!+\, X : \tau}$$

$$\mathbf{binders}(\Sigma, C(x_1, \ldots, x_k)) = x_1 : \tau_1, \ldots, x_k : \tau_k \quad \text{where } C : \tau_1 * \ldots * \tau_k \to T \in \Sigma$$

Figure 3. λ_{syn}: typechecking rules.

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

EGUESS_VAR

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

EGUESS_APP

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

$$\boxed{I \vDash X}$$

SATISFIES

$$\frac{\forall \sigma \ \mapsto\ ex \in X.\sigma(I) \longrightarrow^{*} v \wedge v \simeq ex}{I \vDash X}$$

IREFINE_CTOR

$$\frac{\begin{array}{l}X = \overline{\sigma_i \mapsto C(ex_{1n}, \ldots, ex_{kn})}^{\,i<n}\\ C : \tau_1 * \,..\, * \tau_k \to T \in \Sigma\\ \mathbf{proj}(X) = X_1, \ldots, X_k\\ \overline{\Sigma\,;\,\Gamma \vdash \tau_i \rhd X_j \overset{I}{\rightsquigarrow} I_i}^{\,j<k}\end{array}}{\Sigma\,;\,\Gamma \vdash T \rhd X \overset{I}{\rightsquigarrow} C(I_1, \,..\,, I_k)}$$

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

IREFINE_GUESS

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

IREFINE_FIX

$$\frac{\begin{array}{l}X = \sigma_1 \mapsto pf_1\,;\ ..\ ;\,\sigma_n \mapsto pf_n\\ X' = \mathbf{apply}\,(f, x, \sigma_1 \mapsto pf_1) +\!\!+\ ..\ +\!\!+\, \mathbf{apply}\,(f, x, \sigma_n \mapsto pf_n)\\ \Sigma\,;\,f : \tau_1 \to \tau_2, x : \tau_1, \Gamma \vdash \tau_2 \rhd X \overset{I}{\rightsquigarrow} I\end{array}}{\Sigma\,;\,\Gamma \vdash \tau_1 \to \tau_2 \rhd X \overset{I}{\rightsquigarrow} \mathsf{fix}\,f\,(x : \tau_1) : \tau_2 = I}$$

IREFINE_MATCH

$$\frac{\begin{array}{l}\Sigma\,;\,\Gamma \vdash T \overset{E}{\rightsquigarrow} E \qquad \mathbf{distribute}\,(\Sigma, T, X, E) = \overline{(p_i, X_i')}^{\,i<m}\\ \overline{\mathbf{binders}\,(\Sigma, p_i) = \Gamma_i}^{\,i<m}\\ \overline{\Sigma\,;\,\Gamma_i +\!\!+\, \Gamma \vdash \tau \rhd X_i' \overset{I}{\rightsquigarrow} I_i}^{\,i<m}\end{array}}{\Sigma\,;\,\Gamma \vdash \tau \rhd X \overset{I}{\rightsquigarrow} \mathsf{match}\ E\ \mathsf{with}\ \overline{p_i \to I_i}^{\,i<m}}$$

Figure 4. λ_{syn}: synthesis rules.

Note that these two judgments are very similar to the bidrectional typechecking system for this language! We have simply changed perspectives: rather than checking or producing types given a term, synthesis produces a term given a type.

The EGUESS rules correspond to *guessing* E-forms. "Guessing" in this context amounts to well-typed *term enumeration*. Because the ill-typed terms vastly outnumber the well-typed terms, restricting enumeration in this manner makes enumeration much more efficient [10]. Enumeration proceeds by choosing a particular term shape and then recursively generating its components. Generating an application (EGUESS_APP) consists of generating a function that produces the desired goal type and then generating a compatible argument. Generating a variable (EGUESS_VAR) requires no recursive generation—we simply choose any variable from the context of the appropriate type.

In order to enumerate I-forms, the EGUESS judgment calls into the IREFINE judgment with an empty example list (written $\cdot$). In the absence of examples, the IREFINE judgment also performs well-typed term enumeration. To see this, ignore all occurrences of examples X in the IREFINE rules. IREFINE_FIX generates a fix by recursively generating its body, under the additional assumption of variables f and x. IREFINE_CTOR generates a constructor value by recursively generating arguments to that constructor, and IREFINE_MATCH generates a match by recursively generating the different pieces of the match. Because Es are also syntactically considered Is, IREFINE_GUESS generates an E by using the EGUESS judgment.

Example Propagation When X is non-empty, the IREFINE judgment corresponds to *refinement*, which is restricted to I-forms. In addition to generating well-typed terms, the judgment also incorporates the list of examples X given by the user.

Recall that X is a list of example worlds $\sigma \mapsto ex$ where ex is the goal example value and σ is the execution environment that gives values to each of the free variables in Γ. For this to work, we enforce the following invariant in our system:

Invariant 1. *Example-Value Consistency.* $\forall x : \tau \in \Gamma.\ \forall \sigma \mapsto ex \in X.\ \exists v.\ \Sigma\,;\,\cdot \vdash v : \tau$ *and* $[v/x] \in \sigma$.

The invariant that says that each example world gives a value to each variable in Γ.

Rules IREFINE_FIX and IREFINE_CTOR perform type-directed example refinement. Typechecking ensures that X only contains partial function values in the IREFINE_FIX case and constructor

$$\begin{aligned}
&\mathbf{proj}(X) = \overline{\sigma_i \mapsto ex_{1i}}^{\,i<n}, \ldots, \overline{\sigma_i \mapsto ex_{ki}}^{\,i<n} \\
&\quad \text{where } X = \overline{\sigma_i \mapsto C(ex_{1i}, \ldots, ex_{ki})}^{\,i<n} \\[1ex]
&\mathbf{apply}(f, x, \sigma \mapsto pf) = \overline{[pf/f][v_i/x]\,\sigma \mapsto ex_i}^{\,i<m} \\
&\quad \text{where } pf = \overline{v_i \Rightarrow ex_i}^{\,i<m} \\[1ex]
&\mathbf{binders}(\Sigma, C(x_1, \ldots, x_k)) = x_1 : \tau_1, \ldots, x_k : \tau_k \\
&\quad \text{where } C : \tau_1 * \ldots * \tau_k \to T \in \Sigma \\[1ex]
&\mathbf{distribute}(\Sigma, T, X, E) = (p_1, X'_1), \ldots, (p_n, X'_n) \\
&\quad \text{where} \\
&\qquad \mathbf{ctors}(\Sigma, T) = C_1, \ldots, C_n \\
&\qquad \forall i \in 1, \ldots, n.\ p_i = \mathbf{pattern}(\Sigma, C_i) \\
&\qquad \forall i \in 1, \ldots, n.\ X'_i = \ [\sigma'\sigma \mapsto ex \mid \sigma \mapsto ex \in X, \\
&\qquad\quad \sigma(E) \longrightarrow^* C_i(\overline{ex}),\ \mathbf{vbinders}(p_i, \overline{ex}) = \sigma'] \\[1ex]
&\mathbf{ctors}(\Sigma, T) = C_1, \ldots, C_n \\
&\quad \text{where } \forall i \in 1, \ldots, n.\ \ C_i : \overline{\tau} \to T \in \Sigma \\[1ex]
&\mathbf{pattern}(\Sigma, C) = C(x_1, \ldots, x_k) \\
&\quad \text{where } C : \tau_1 * \ldots * \tau_k \to T \in \Sigma \\[1ex]
&\mathbf{vbinders}(p, e_1, \ldots, e_n) = [e_1/x_1] \ldots [e_n/x_n] \\
&\quad \text{where } p = C(x_1, \ldots, x_n)
\end{aligned}$$

Figure 5. λ_{syn}: auxiliary synthesis functions.

$$\boxed{v \simeq v'}$$

EQ_CTOR

$$\frac{\overline{u_i \simeq v_i}^{\,i<k}}{C(\,\overline{u_i}^{\,i<k}\,) \simeq C(\,\overline{v_i}^{\,i<k}\,)}$$

EQ_REFL

$$\frac{}{v \simeq v}$$

EQ_PF_FIX

$$\frac{\begin{array}{c} v_2 = \mathsf{fix}\, f(x : \tau_1) : \tau_2 = e \\ \forall i \in 1, \ldots, n.\ e_1\ v_i \longrightarrow^* v \wedge v \simeq ex_i \end{array}}{\overline{v_i \Rightarrow ex_i}^{\,i<n} \simeq v_2}$$

EQ_PF_PF

$$\frac{\begin{array}{c} \forall i \in 1, \ldots, n.\ \exists j \in 1, \ldots, m.\ v_i \simeq v'_j \wedge ex_i \simeq ex'_j \\ \forall j \in 1, \ldots, m.\ \exists i \in 1, \ldots, n.\ v_i \simeq v'_j \wedge ex_i \simeq ex'_j \end{array}}{\overline{v_i \Rightarrow ex_i}^{\,i<n} \simeq \overline{v'_j \Rightarrow ex'_j}^{\,j<m}}$$

$$\boxed{e \longrightarrow e'}$$

EVAL_APP_PF_GOOD

$$\frac{v \simeq v_j \qquad j \in 1, \ldots, n}{\overline{v_i \Rightarrow ex_i}^{\,i<n}\ v \longrightarrow ex_j}$$

EVAL_APP_PF_BAD

$$\frac{\forall j \in 1, \ldots, n.\ v \not\simeq ex_j}{\overline{v_i \Rightarrow ex_i}^{\,i<n}\ v \longrightarrow \mathsf{NoMatch}}$$

Figure 6. Selected λ_{syn} evaluation and compatibility rules for partial functions.

values in the IREFINE_CTOR case. This refinement is delegated to helper functions **apply** and **proj**, respectively, which are specified in Figure 5.

proj assumes that the example values of X consist of constructor values with a shared head C that has arity k. **proj** creates k new Xs, corresponding to the k arguments that must be synthesized for C. The example values for X_k which corresponds to the kth argument are drawn from the kth example values from each of the example values found in the original X. The environment σ_i are copied, unchanged, to each of the new X_k.

apply assumes that the example values of X consist of partial functions. **apply** operates over a single example world $\sigma \mapsto pf \in X$ and produces a new X' that refines the examples for synthesizing the function body I. If $pf = \overline{v_i \Rightarrow ex_i}^{i<m}$, then **apply** produces m new example worlds, one for each such $v_i \Rightarrow ex_i$ pair. The goal example value for each of these worlds is ex_i. To fulfill our invariant that the σ_i has a value for each variable bound in Γ, we must provide values for f and x. x is simply bound to v_i. f is bound to the *entire* partial function that generated the example world—this means that synthesis can use examples of f's behavior on any of the given sample inputs when generating the function body. In IREFINE_FIX, we append the results of **apply**ing each example world in X together to form the final X' used to synthesize I.

IREFINE_MATCH does not refine examples like IREFINE_FIX and IREFINE_CTOR. Instead, IREFINE_MATCH *distributes* the example worlds in X to each of the branches of the match. This behavior is delegated to the **distribute** function, which takes the signature Σ, the data type T the match covers, the examples to refine X, and the scrutinee of the match E. **distribute** generates an X_m for each branch of the match by distributing the examples X among the m branches of the pattern match. Intuitively, **distribute** evaluates the scrutinee E using each of the σ_n drawn from the n worlds. Because E must be a data type T, then it must evaluate to one of T's constructors. **distribute** then sends that example world to that constructor's branch in the match.

Finally, we bridge the refinement and guessing judgments with IREFINE_GUESS. For E-forms, we are unable to push examples through term generation. This is because the shape of an E does not tell us anything about its type and correspondingly, how to refine its examples. However, after generating an E, we can check that it satisfies each example world. This "satisfies" relation, written $E \vDash X$, ensures that for all example worlds $\sigma_n \mapsto ex_n$ in X that $\sigma_n(E) \longrightarrow^* v$ and $v \simeq ex_n$ where $\sigma_n(E)$ substitutes the environment σ_n into E.

3.3 Evaluation and Compatibility

The synthesis rules require that we evaluate terms and check them against the goal examples. Figure 6 shows just two of the small-step evaluation rules. The omitted rules for β-reduction and pattern-matching are completely standard; they rely on the usual notion of capture-avoiding substitution.

There are two ways that the interpreter can go wrong: it might diverge or it might try to use a partial function that is undefined at a particular input. In the first case, our multi-step evaluation relation $e \longrightarrow^* v$ does not terminate which implies that the synthesis process itself could diverge. However, in practice, our synthesis algorithm imposes restrictions on recursion to guarantee termination which we discuss further in Section 3.4.

The rules governing partial functions are shown in Figure 6. In the case that the partial function is used at an undefined input, an exception value NoMatch is raised. NoMatch does not equal any value, so synthesis fails. At first glance, this seems overly restrictive as the absence of a particular input in a partial function might seem to imply that we can synthesize *anything* for that case, rather than nothing. 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.

Value compatibility, written $v \simeq u$, is also shown in Figure 6. This relation, which is defined only on closed terms, is used to determine when a guessed E is compatible with the examples. Due to the presence of higher-order functions, its definition is a bit delicate. As usual, two values that have the same head constructor applied to compatible arguments are compatible (rule EQ_CTOR). A specified function is compatible with a partial function example if running the function on each of the given inputs produces an output compatible with the corresponding example output, as shown in EQ_PF_F (omitting the symmetric rule). Two partial functions are compatible if they specify equivalent sets of pairs of input/output examples. However, a concrete function is compatible only with itself, via reflexivity.

Compatibility is an approximation to extensional equality, which is undecidable in general. Our approximation is conservative in that our use of $\simeq$ rather than (unattainable) extensional equality only permits *fewer* programs to be synthesized, in particular, when higher-order functions are involved. For example, consider E-guessing a term (via the IREFINE_GUESS rule) involving an function `f` whose value is the partial function `id => O`. If `f` is applied to some function value `v` where `v` is contextually equivalent, yet syntactically distinct, from `id`, evaluation will produce NoMatch rather than `O`.

3.4 Metatheory

With the definition of λ_{syn} in hand, we give an overview of the properties of λ_{syn}.

Termination Because synthesis relies on evaluation (through the compatibility relation), termination of any synthesis algorithm based on λ_{syn} relies on the termination of evaluation. In the presence of recursive functions and data types, we must enforce two restrictions to ensure termination of evaluation:

1. A syntactic *structural recursion* check that enforces that recursive calls are only made on structurally decreasing arguments.
2. A positivity restriction on data types that prevents recursive occurrences of a data type to the left of an arrow in the type of an argument to a constructor.

Both of these restrictions are present in languages that require totality, *e.g.*, Agda [18] and the Coq theorem prover [24]. We elide the details of the structural recursion check here to simplify the presentation of λ_{syn}.

Example Consistency Example consistency demands that the examples that the user provides do not contradict each other. For example, if the goal type is `nat`, then the two examples `O` and `S(O)` are contradictory because we cannot synthesize a single I of type `nat` that can both be `O` and `S(O)` (in the empty context).

Checking for inconsistent examples proves to be difficult because of the undecidability of function equality in λ_{syn}. For example, consider the partial function `id1 => O | id2 => 1` where `id1` and `id2` are syntactically distinct implementations of the identity function. This partial function has contradictory alternatives, however, we cannot determine that `id1` $\simeq$ `id2`.

Therefore, in λ_{syn}, we simply assume that the set of examples that the user provides is not contradictory. If λ_{syn} is supplied with an contradictory set of examples, then there will be no I that fulfills the examples. In an implementation of a synthesis algorithm based on λ_{syn} (such as the one we present in Section 4), this results in the algorithm not terminating because it can never find a program that satisfies the examples.

Soundness Soundness of λ_{syn} ensures that synthesized programs are correct.

Theorem 1. *Type Soundness. If* $\Sigma \ ; \ \Gamma \vdash X : \tau$ *and* $\Sigma \ ; \ \Gamma \vdash \tau \rhd X \overset{I}{\rightsquigarrow} I$ *then* $\Sigma \ ; \ \Gamma \vdash I \Leftarrow \tau$.

Theorem 2. *Example Soundness. If* $\Sigma \ ; \ \Gamma \vdash \tau \rhd X \overset{I}{\rightsquigarrow} I$ *then* $I \vDash X$.

Type soundness states that synthesized programs are well-typed, and example soundness states that synthesized programs agree with the examples that are given.

Proving type soundness is straightforward: the synthesis rules erase to typing rules for the language, so we always produce well-typed programs. Proving example soundness amounts to showing that the IREFINE rules manipulate the examples soundly. The example refinement performed by IREFINE_FIX, IREFINE_CTOR, and IREFINE_MATCH all correspond to single-step evaluation for fix, constructor, and match expressions, respectively, over the examples. The base case, IREFINE_GUESS, ensures compatibility directly.

Completeness Completeness of λ_{syn} ensures that we are able to synthesize all programs.

Theorem 3. *Completeness of Term Enumeration.*

1. *If* $\Sigma \ ; \ \Gamma \vdash E \Rightarrow \tau$ *then* $\Sigma \ ; \ \Gamma \vdash \tau \overset{E}{\rightsquigarrow} E$.
2. *If* $\Sigma \ ; \ \Gamma \vdash I \Leftarrow \tau$ *then* $\Sigma \ ; \ \Gamma \vdash \tau \rhd \cdot \overset{I}{\rightsquigarrow} I$.

In the absence of examples, λ_{syn} can synthesize any well-typed term. This follows from the fact that the λ_{syn} is simply an inversion of the inputs and outputs of the standard typing judgment.

Theorem 3 implies that, for any well-typed I, there always exists an example set X that allows us to synthesize I, namely the empty set. Therefore, we would like to state that λ_{syn} satisfies the following property

Claim 1. *Completeness of Synthesis. If* $\Sigma \ ; \ \Gamma \vdash X : \tau$, $\Sigma \ ; \ \Gamma \vdash I \Leftarrow \tau$, $I \vDash X$, *then* $\Sigma \ ; \ \Gamma \vdash \tau \rhd X \overset{I}{\rightsquigarrow} I$.

which says that if X and I are well-typed and I satisfies X, then we can use X to synthesize I. However, it turns out that this claim does not hold for λ_{syn}. The problem resides in our use of partial functions as the value for a recursive function during I-refinement. In the IREFINE_FIX case, we end up needing to claim that a partial function can be substituted for the fix. While the partial function agrees with the fix on the values that the partial function is defined, it does not agree on the other, unspecified values (which all raise NoMatch errors). This makes such a substitution unsound in general and requires stronger assumptions about how the partial function and fix are related.

4. Implementation

So far, we have described type-directed synthesis as a logical system, λ_{syn}, that leverages types and examples in a push-down manner. This presentation illustrates the close connections between type-directed program synthesis and bidirectional typechecking and proof search. However, λ_{syn} itself is not a synthesis procedure on its own. We now discuss how we translate λ_{syn} into an efficient synthesizer, MYTH.

4.1 Consistency, Termination, and Determinism

The system described in Figure 4 is highly non-deterministic. Furthermore, we assume that the examples provided to the system are consistent. Finally, it relies on evaluation to verify E-guesses, which may not terminate in the presence of recursive functions. To move towards an implementation, we must address these issues.

Consistency Because we allow function values in the domain of partial functions, checking for contradictory example sets is undecidable (because checking function equality is undecidable in our system). Therefore, in MYTH, like λ_{syn}, we are unable to detect upfront whether an input X is consistent.

However, in MYTH, we search the space of programs in (roughly) increasing program size. For a given set of consistent examples, there always exists at least one program that satisfies those examples. This guaranteed program specializes to exactly the examples (*e.g.*, with a collection of nested pattern matches) and has arbitrary behavior on all other values.

Therefore one way to detect an inconsistent example set is to simply generate terms up to the size of the guaranteed program—whose size is proportional to the size and amount of input examples. If any of these programs satisfies the example set, then the example set is consistent. Otherwise, the example set is inconsistent. Thus, the size of the guaranteed program serves as an upper bound to the synthesis algorithm.

While this works in theory, in practice the size of the guaranteed program usually exceeds the size of programs that MYTH can synthesize in a reasonable amount of time. Thus, we avoid introducing such a check in MYTH; inconsistent example sets simply cause MYTH to not terminate.

Termination As discussed in Section 3.4, we must add structural recursion checks to recursive function calls and enforce a positivity restriction on data types to obtain termination of evaluation. The structural recursion check is a simpler, more coarse-grained form of the syntactic check used in the Coq theorem prover [24]. Recall that our functions (Figure 2) are all curried, *i.e.*, take only a single argument. We require that this value passed to a `fix` be structurally decreasing. Such a structurally decreasing value can only be obtained by pattern matching on the `fix`'s original argument. This is more restrictive than Coq which performs some compile-time β-reductions to admit more arguments, but is, nevertheless, sound and serves as a reasonable over-approximation to ensure termination.

Determinism The non-determinism in λ_{syn} arises from multiple rules applying at once. For example, we will certainly get into the situation when both the IREFINE_MATCH and IREFINE_GUESS rules are potentially valid derivations since they both apply at base type. How do we pick which rule to invoke? And how do we choose one if both are valid derivations? In general, we don't know *a priori* how to proceed.

Our approach is to exhaustively search all of these possibilities that have an appropriate type and agree with the examples in turn. This search may not terminate, especially if the user has not provided sufficient examples, so the procedure is governed by an iterative deepening strategy dictated by term size. That is, we try to synthesize a valid program starting with size one and increase the size if the search fails. This procedure continues until we find the program or reach a user-defined size limit.

Note that this strategy produces the smallest program that satisfies the examples. The smallest such program is desirable because it most likely generalizes to the complete behavior of the function that we want. This is because smaller programs use more variables and less constants which are more likely to generalize to more complex behavior.

Blind search of derivations is straightforward and works well for small functions but does not scale well to more complicated code. Our two main modifications to this naïve search strategy are memoizing term generation and *refinement trees*, an efficient data structure for representing λ_{syn} derivations.

4.2 Efficient Contextual Term Generation

We can think of E-guessing as raw E-term enumeration coupled with evaluation. E-guessing occurs at every step of synthesis (because IREFINE_GUESS applies at any time), so it pays to make term enumeration as efficient as possible. On top of this, we will certainly generate the same terms repeatedly during the course of synthesis. For example, when synthesizing a function that takes an argument x, we will potentially enumerate expressions containing x at any point when E-guessing subexpressions within the body of the function.

Therefore, we define term generation functions:

$$\texttt{gen}_E(\Sigma;\ \Gamma;\ \tau;\ n) = \mathcal{E} \quad \text{and} \quad \texttt{gen}_I(\Sigma;\ \Gamma;\ \tau;\ n) = \mathcal{I}$$

that realize term enumeration in λ_{syn} as a collection semantics that enumerates sets of E-terms $\mathcal{E}$ and I-terms $\mathcal{I}$ of type τ having exactly size n.

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

$$\begin{aligned}
\texttt{gen}_E^{x:\tau_1}(\Sigma;\ \Gamma;\ \tau;\ 0) &= \{\} \\
\texttt{gen}_E^{x:\tau}(\Sigma;\ \Gamma;\ \tau;\ 1) &= \{x\} \\
\texttt{gen}_E^{x:\tau_1}(\Sigma;\ \Gamma;\ \tau;\ 1) &= \{\} \qquad (\tau \neq \tau_1) \\
\texttt{gen}_E^{x:\tau_1}(\Sigma;\ \Gamma;\ \tau;\ n) &= \bigcup_{\tau_2 \to \tau \in \Gamma} \bigcup_{k=1}^{n-1} \text{(see below)}
\end{aligned}$$

$$\begin{array}{lllll}
 & (\texttt{gen}_E^{x:\tau_1}(\Sigma;\ \Gamma;\ \tau_2 \to \tau;\ k) & \underset{app}{\otimes} & \texttt{gen}_I(\Sigma;\ \Gamma;\ \tau_2;\ n-k)) \\
\cup & (\texttt{gen}_E(\Sigma;\ \Gamma;\ \tau_2 \to \tau;\ k) & \underset{app}{\otimes} & \texttt{gen}_I^{x:\tau_1}(\Sigma;\ \Gamma;\ \tau_2;\ n-k)) \\
\cup & (\texttt{gen}_E^{x:\tau_1}(\Sigma;\ \Gamma;\ \tau_2 \to \tau;\ k) & \underset{app}{\otimes} & \texttt{gen}_I^{x:\tau_1}(\Sigma;\ \Gamma;\ \tau_2;\ n-k))
\end{array}$$

Figure 7. Selected relevant E-term generation functions.

For performance, we should cache the results of E-guessing at a particular goal type so that we never enumerate a term more than once. However, in λ_{syn} we E-guess under a context that grows as we synthesize under binders, *i.e.*, functions and **match**es. To maximize cache hits, we should try to create invocations of $\texttt{gen}_E$ and $\texttt{gen}_I$ that maximize sharing of the contexts Γ. Noting that the synthesis procedure only extends the context and never shuffles its contents, we can factor the term generation functions as follows:

$$\begin{aligned}
\texttt{gen}_E(\Sigma;\ \Gamma, x\!:\!\tau_1;\ \tau;\ n) &= \texttt{gen}_E^{x:\tau_1}(\Sigma;\ \Gamma;\ \tau;\ n) \\
&\cup\ \ \texttt{gen}_E(\Sigma;\ \Gamma;\ \tau;\ n) \\[1ex]
\texttt{gen}_I(\Sigma;\ \Gamma, x\!:\!\tau_1;\ \tau;\ n) &= \texttt{gen}_I^{x:\tau_1}(\Sigma;\ \Gamma;\ \tau;\ n) \\
&\cup\ \ \texttt{gen}_I(\Sigma;\ \Gamma;\ \tau;\ n)
\end{aligned}$$

This factorization ensures that for a given goal type and size, two calls to $\texttt{gen}_E$ in different contexts $\Gamma, x_1 : \tau_1$ and $\Gamma, x_2 : \tau_2$ will still have a cache hit for the shared prefix context Γ.

Here, $\texttt{gen}_E^{x:\tau_1}$ and $\texttt{gen}_I^{x:\tau_1}$ are *relevant term generation functions*. Inspired by relevance logic [3], these functions require that all gathered terms must contain the relevant variable x. Figure 7 gives the definition of $\texttt{gen}_E^{x:\tau_1}$; the definition of $\texttt{gen}_I^{x:\tau_1}$ follows analogously from the type-directed IREFINE judgment ignoring example refinement. In particular, relevant generation bottoms out when either the requested term size is zero or one. In the latter case, we enumerate only the relevant variable x at size one even though there may be other variables in the context that meet the goal type. We will enumerate those variables when they become relevant, *i.e.*, appear at the beginning position of the context.

The only other E-term form is function application which we enumerate in the recursive case of the $\texttt{gen}_E^{x:\tau_1}$. Enumerating application terms consists of forming the Cartesian product $\underset{app}{\otimes}$ of applications from recursive calls both the $\texttt{gen}_E$ and $\texttt{gen}_I$ functions. In order to respect relevance, we take the union of the terms where the relevant variable must appear in either side of the application or both sides. Finally we enumerate these applications by taking all the ways we can partition the size between the function and argument expression and all the possible function types that generate our goal type τ_2.

4.3 Refinement Trees

λ_{syn} operates in two distinct modes: E-guessing, which involves term generation, and I-refinement, which involves pushing down examples in a type-directed manner. A natural question is whether we can cache I-refinements in a manner similar to how we cache E-guesses. The answer is yes, but to do so, we need to introduce

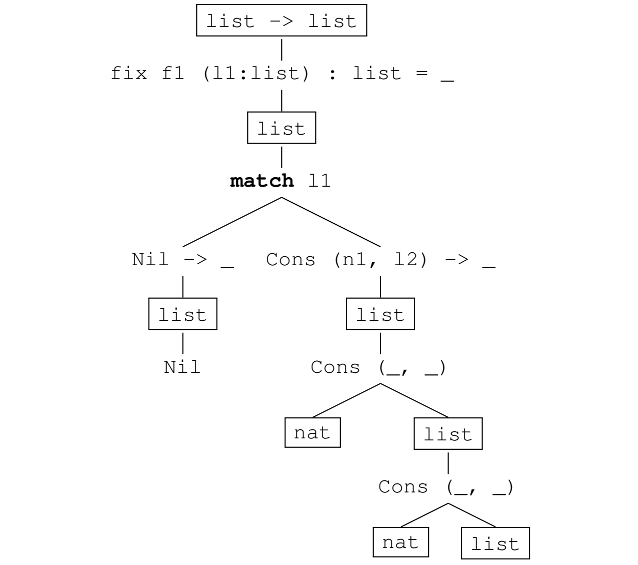


Figure 8. Example refinement tree for `stutter`.

a data structure called a *refinement tree*, which captures all the possible refinements of the examples that the synthesis procedure could perform.

Figure 8 gives an example refinement tree generated while synthesizing `stutter` with the examples from Figure 1. The tree is specialized to the case of one **`match`**. There are two types of nodes in the refinement tree:

1. *Goal nodes* contain goal types and represent places in the synthesis derivation where we can E-guess. In Figure 8, they are boxed. For example, the top-most node containing `list -> list` is a goal node.
2. *Refinement nodes* represent valid I-refinements. In Figure 8 they are unboxed, for example, the node containing **`match`** `l1` is a refinement node that represents a **`match`** on `l1`. Its two children represent the synthesis sub-problems for synthesizing its `Nil` and `Cons` branches. Likewise, the children of the refinement node containing `Cons (_, _)` represent the sub-problems for synthesizing each of the arguments to `Cons`

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 figure, **`match`** has been specialized to case on `l1`, the only informative scrutinee justified by the examples.

When using refinement trees, our synthesis strategy differs significantly from pure enumeration. It proceeds in three steps:

1. Create a refinement tree from the initial context, goal type, and provided examples.
2. By IREFINE_GUESS, E-guessing is always an alternative derivation at any point during I-refinement. Therefore, perform E-guessing at each node (using that node's context, goal type, and examples).
3. Bubble successful E-guesses and refined nullary constants upwards in the tree to try to construct a program that meets the top-level goal.

We repeat these steps in an iterative deepening fashion governed by a number of metrics, not just the obvious choice of term size, that we will discuss in more detail shortly.

In this search, applications of the IREFINE_MATCH rule deserve special attention. A **`match`** has the effect of introducing new information into the context, *i.e.*, additional binders gathered from destructing values. However, it otherwise does not further the goal state (note how IREFINE_MATCH only distributes the goal examples among the branches of the **`match`** unmodified). If left unchecked, the search procedure wastes a lot of time **`match`**ing on values that add no information to the context. Therefore, we check to see whether a **`match`** might help make progress towards the goal:

Definition 1. **`match`** *informativeness. An application of the* IREFINE_MATCH *rule is* informative *if whenever the* **`match`** *contains at least two branches, at least two branches of the* **`match`** *contain non-empty sets of goal examples.*

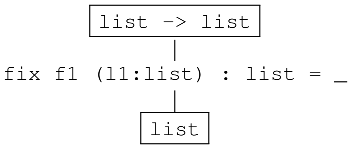
In our search, we modify IREFINE_MATCH to apply only when the generated match is informative. This restriction is sufficient to rule out many unproductive matches, *i.e.*, those matches that send all the goal examples to a single branch. And, moreover, this restriction contributes to an important property of refinement trees:

Lemma 1. *Finiteness of refinement trees. Given an upper bound on the size of any* `match` *scrutinee, all refinement trees are finite.*

Proof sketch. We observe that for successive applications of IREFINE_FIX and IREFINE_CTOR, the goal examples monotonically decrease in size as we repeatedly destruct them. So there is always a point at which refinement bottoms out and can no longer apply these rules. The same logic applies to IREFINE_MATCH, but for examples in the context. Informativeness ensures that successive matches on the same (or a less informative) scrutinee are never informative as they will always send the remaining set of examples to the same branch. □

Because refinement trees correspond to IREFINE derivations minus applications of IREFINE_GUESS, the lemma implies that there are only a finite number of such derivations possible when the size of **`match`** scrutinees is bounded. Thus, our iterative deepening search over refinement trees uses the maximum **`match`** scrutinee size and the maximum E-guess term size as metrics. While **`match`** depth is known to be finite according to Lemma 1, we also use it as a final metric in our search. This is because **`match`** expressions potentially add many new binders which causes the search space to grow very quickly.

As a concrete example, let's consider how we would synthesize the `stutter` from Section 1 using refinement trees. Initially, we construct a simple refinement tree with match depth zero. This tree



contains three nodes: a goal node corresponding to the initial goal type of `list -> list`, a refinement node corresponding to applying IREFINE_FIX, and a goal node corresponding to synthesizing the body of that `fix`. We then E-guess up to a reasonable term size limit, say 15, at each of the nodes, but fail to discover any programs that are compatible with the examples. So we grow the refinement tree by increasing the match depth, which results in the tree found in Figure 8. E-guessing over this tree yields a variety solutions to the various synthesis sub-problems found in the tree. In particular, the search finds `Cons (n1, Cons (n1, f1 l2))` in the `Cons` branch by E-guessing `n1` and `f1 l2` at their respective nodes in the refinement tree, yielding the complete `stutter` program.

Refinement trees provide a number of benefits:

1. They act as a cache for I-refinement derivations.
2. They separate I-refinement from E-guessing, which in turn allows us to exploit the finiteness of I-refinement derivations to search the space of derivations in a more intelligent way.
3. They dramatically reduce the average size of E-guessed terms. For example, for `stutter`, rather than trying to enumerate whole terms up to size 11, we only need to enumerate two terms of size one and one term of size three.

4.4 Additional Optimizations

In addition to these two main optimizations we also introduce several other additional optimizations to improve MYTH's performance.

No* `match` *Term Enumeration Because a **`match`** statement does not refine the type of our goal, we could theoretically enumerate a **`match`** at any point during term enumeration. This is potentially very inefficient, so we disallow **`match`** generation during term enumeration, *i.e.*, IREFINE_MATCH only applies when `X` is non-empty. Note that we do not lose expressiveness because contextually equivalent terms can be synthesized by using IREFINE_MATCH to generate an outer **`match`** during example refinement.

Short-circuiting A goal node in a refinement tree represents a position where we can obtain a solution either through I-refinement and solving synthesis sub-problems or E-guessing a program directly. In either case, once we find a solution for a particular goal node, our synthesis procedure ensures that this local solution is valid for the overall program. Therefore, once we discover a satisfying expression at a goal node, we no longer E-guess at that node or any of its children to avoid doing unnecessary work.

Base Type Restrictions IREFINE_MATCH allows us to **`match`** at any type, and IREFINE_GUESS allow us to guess an E at any type. This is overly permissive because we can always apply IREFINE_FIX first until the goal is a base type. This is because `fix` is *invertible* in the sense that whenever the `fix` itself can be synthesized, its body can also be synthesized. Therefore, in MYTH, we restrict IREFINE_MATCH and IREFINE_GUESS to only operate at base type in order to cut down the search space further. This results in the synthesis of β-normal, η-long terms without loss of expressiveness (up to β, η-equivalence).

5. Results

So far, we have shown how to turn λ_{syn}, a non-deterministic synthesis calculus into a algorithmic synthesis procedure. To assess the strengths and weaknesses of the approach, we have implemented a prototype tool, MYTH, in OCaml, and exercised it on a number of benchmarks and larger examples.

5.1 Search Procedure Tuning

Section 4 defines an iterative deepening search over derivations of synthesized terms in λ_{syn} controlled by three parameters: **`match`** depth, **`match`** scrutinee size, and E-term size. How we explore them affects both the performance of the search and the synthesized programs. For example, choosing too large of a maximum E-term size might cause the algorithm to spend a long time generating terms when adding a **`match`** would be more profitable. Conversely, adding **`match`**es too quickly may over-specialize the synthesized program. For example, synthesizing `stutter` starting from a refinement tree at **`match`** depth two yields:

```
let stutter : list -> list =
  fun (l1:list) ->
   match l1 with
     | Nil -> l1
     | Cons (n1, l2) ->
     (match n1 with
     | O -> Cons (n1, l1)
     | S (n2) ->
      Cons (n1, Cons (n1, Cons (n2, l2))))
```

which is discovered before the program in Figure 1 because the size of the largest E-term in the above program is one versus size three (for `f1 l2`) in the desired program.

Thus, our strategy for iterating over these three parameters is critical for ensuring that we synthesize the best programs—those whose behavior correctly generalizes the given examples without overspecialization—in the least amount of time. Nevertheless, the current implementation of MYTH, uses a relatively naïve strategy: simply alternate between E-guessing and growing the refinement tree. An E-guessing round enumerates terms up to a constant maximum size or a pre-determined timeout (set to 0.25 seconds in the current implementation), whichever occurs first. A tree growing round either extends the **`match`** depth by one or increases the potential scrutinee size of all possible matches by five (chosen because binary function application, `f e1 e2`, has size five). The first three rounds grow the **`match`** depth, and the last two rounds increase the scrutinee size. Certainly, there are smarter, more adaptive search strategies we could adopt, but for the set of examples we've explored so far, this simple strategy is sufficient.

5.2 Benchmarks and Extended Examples

Our benchmark suite exercises the functionality of MYTH on a number of basic datatypes and scenarios. Each test is a synthesis problem—a collection of data type definitions, top-level function definitions, a goal type, and set of examples. MYTH is configured to report the first function that it synthesizes. Figure 9 summarizes the results. We categorize each test by its domain and report the number of examples provided to the synthesizer, the size of the generated program (in expression AST nodes), and the time taken to synthesize the program.[2] The reported times count only time spent in the synthesizer which excludes other tasks such as parsing and typechecking, which take a negligible amount of additional time.

For each test, we developed its corresponding input example set by starting with a small set of initial examples and iteratively refining them against MYTH until the tool produced a function that correctly generalizes the desired behavior implied by the example set. We manually verified that the final function has the correct behavior for all inputs. In most cases the function generated by MYTH was precisely the function that we would have written by hand. In a handful of cases, the synthesized function's correctness was non-obvious, but nevertheless correct. We discuss these tests in more detail later in this section.

The MYTH benchmark suite focuses on core functional programming tasks: manipulation of common inductive data types—`bool`, `nat`, `list`, and `trees`—pattern matching, and the creation and manipulation of higher-order functions. The tests include basic non-recursive functions (*e.g.*, `bool_xor`), recursive functions (*e.g.*, `list_length`), generation of higher-order recursive functions (*e.g.*, `list_fold`, `tree_map`), and use of higher-order functions (*e.g.*, `list_sum`).

Beyond the benchmark suite, we have also included several examples that use more interesting data types to explore MYTH's

[2] All reported data is generated on a Linux desktop machine with an Intel i7-4770 @ 3.4 GHz and 16 Gb of ram. Note that MYTH is a single-threaded application because OCaml cannot natively take advantage of multiple CPU cores due to the presence of a global interpreter lock.

Test	#Ex	#N	T-Ctx (sec)	T-Min (sec)
Booleans				
bool_band	4	6	0.003	0.002
bool_bor	4	6	0.004	0.001
bool_impl	4	6	0.004	0.002
bool_neg	2	5	0.001	0.0
bool_xor	4	9	0.003	0.002
Lists				
list_append	12	12	0.011	0.003
list_compress	13	28	128.339	0.073
list_concat	6	11	0.019	0.006
list_drop	13	13	1.29	0.013
list_even_parity	7	13	0.518	0.004
list_filter	10	17	4.320	0.067
list_fold	9	13	0.504	0.139
list_hd	3	5	0.019	0.001
list_inc	4	8	0.004	0.00
list_last	6	11	0.093	0.00
list_length	3	8	0.019	0.001
list_map	8	12	0.082	0.008
list_nth	24	16	0.96	0.013
list_pairwise_swap	20	19	10.227	0.007
list_rev_append	5	13	0.028	0.011
list_rev_fold	5	12	0.014	0.007
list_rev_snoc	5	11	0.018	0.006
list_rev_tailcall	14	12	0.004	0.004
list_snoc	8	14	0.07	0.003
list_sort_sorted_insert	7	11	0.009	0.008
list_sorted_insert	12	24	22.016	0.122
list_stutter	3	11	0.018	0.001
list_sum	3	8	0.005	0.002
list_take	12	15	1.147	0.112
list_tl	3	5	0.02	0.001
Nats				
nat_add	9	11	0.023	0.002
nat_iseven	4	10	0.014	0.001
nat_max	9	14	0.002	0.011
nat_pred	3	5	0.004	0.001
Trees				
tree_binsert	20	31	9.034	0.374
tree_collect_leaves	6	15	0.033	0.016
tree_count_leaves	7	14	0.042	0.008
tree_count_nodes	6	14	0.037	0.009
tree_inorder	5	15	0.036	0.012
tree_map	7	15	0.035	0.014
tree_nodes_at_level	24	22	4.917	1.093
tree_postorder	9	32	7.929	1.136
tree_preorder	5	15	0.023	0.009

Figure 9. Aggregated benchmark suite results. For each test, we report the number of given examples (#Ex), the size of the result (#N), and times taken in seconds to synthesize in a populated context (T-Ctx) and minimal context (T-Min).

limits. Figure 10 showcases our results. `Arith` is an interpreter for a small calculator language. The `fvs` examples calculate the set of free variables in an untyped lambda-calculus representation. The `small` variant includes constructors for variables (both bound and free), lambdas, and application. `medium` adds pairs, numeric constants, and binary operations. The `large` version adds sums and pattern matching over them.

Revisiting non-termination, recall from Section 3.4 that we ensure termination of evaluation through a structural recursion check and a positivity restriction on data types. The structural recursion check is necessary because it is easy to generate trivial infinite loops such as `fix f (x:t) : t = f x` without the check. Introducing an infinite loop by breaking the positivity restriction is more difficult by comparison. For example, consider the `dyn` type:

Test	#Ex	#N	**T** (sec)
arith	22	47	11.139
dyn_app_twice	6	11	0.715
dyn_sum	25	23	19.420
fvs_large	31	75	3.905
fvs_medium	22	45	0.751
fvs_small	6	16	0.029

Figure 10. Aggregated extended examples results.

```
type dyn = Error
       | Base of nat
       | Dyn of (dyn -> dyn)
```

which represents an encoding of type dynamic over `nat`s and functions. The simplest such program [7] requires calling the `bad` function:

```
let rec bad (d:dyn) : dyn =
  match d with
  | Error -> Error
  | Base (n) -> Error
  | Dyn (f) -> f d
```

with the argument `Dyn(bad)`. However, in Myth, this requires that `bad` is bound in the context already for the synthesizer to create the offending function call. It is likely that the `dyn` type can cause Myth to go into an infinite loop with another program, but we posit that the offending program would be much larger, likely out of the range of programs that Myth can synthesize.

To explore this hypothesis, we lifted the positivity restriction in Myth and wrote a pair of tests involving `dyn` to see if Myth goes into an infinite loop. `dyn_app_twice` generates a program that applies the argument of a `Dyn` twice, and `dyn_sum` sums two `Dyn` values that are `nat`s. In the case that the functions do not receive the proper types, they return `Error`. Myth is able to synthesize correct programs for both tests which suggests that the positivity restriction may be unnecessary for the sizes of programs that Myth currently synthesizes.

5.3 Analysis

Compared with recent work in the space of synthesizing recursive functions, most notably Escher [1] and Leon [15], our benchmark suite contains programs of similar complexity (15–30 AST nodes), and Myth performs at comparable speeds or faster (negligible time in most cases with some larger examples taking significantly more time). We benchmark a larger number of programs than prior work (43 examples versus the 22 reported by Escher and 23 reported by Leon), and to our knowledge `fvs_large` is the largest example of a fully synthesized recursive function in the literature at 75 AST nodes.

However, we stress that such comparisons should be made with care: the choice of target language heavily affects the AST size, and the choice of synthesizing context and number of required examples (not reported in the Escher work) both affect performance. Thus, while we discuss Escher and Leon to provide context, we are not making apples-to-apples comparisons with our own work. We merely want to show that our type-directed approach can perform similarly to previous approaches without claiming one approach is better than the other. Note that our "best" test, `fvs_large`, benefits greatly from Myth's support of algebraic datatypes, but that is the point of this work: taking advantage of type structure can be fruitful for synthesizing larger programs.

Our experience with Myth also revealed a number of insights that our formal development did not predict:

Traces and Example Generation Recall from Section 3 that we evaluate recursive functions currently being synthesizing with the original input partial function example. This requires that the partial function specifies for each input–output pair, all the additional input–output pairs necessary to complete execution of that original call. This is why in the `stutter` example we chose the particular example set `[] => [] | [0] => [0;0] | [1;0] => [1;1;0;0]`. Calling `stutter [1;0]` requires a recursive call to `stutter [0]`. This in turn requires a recursive call to `stutter []`.

Developing the example sets for each test with this restriction in mind proved to be difficult initially. Rather than being completely oblivious of internal behavior of the function we were synthesizing, we needed to reason a bit about how it might make its recursive calls and structure our examples accordingly. Discovering ways to get around this restriction, *e.g.*, by factoring in the parts of the program that you have already synthesized, would greatly help in converting this type-directed synthesis style into a usable tool.

Context Size Myth relies on top-level function definitions in the context—it cannot discover helper functions on its own, so we provide them as inputs to the synthesis process. The `list_rev_fold`, `list_rev_snoc`, and `list_rev_append` tests have their respective helper functions in the context. However, because it relies heavily on term generation during E-guessing, Myth's performance depends on the size of the context. In practice, types help rule out many irrelevant context terms, but there can still be a combinatorial explosion of possibilities.

To assess the impacts of context size, in addition to testing the synthesis time in a minimal context (containing only the necessary type and function bindings), we also tested synthesis time in a larger, shared context. This shared context contains common functions for the basic data types: **`and`**, **`or`**, `plus`, `div2`, and `append`. Other functions, *e.g.*, `inc` and `is_empty`, are implied by constructors and pattern matching. Our choice of components is similar to that used by Albarghouthi *et al.* [1], but adapted to our typed domain.

These times are listed alongside the minimal times in Figure 9 for easy comparison. Overall, there is a 55% average increase in runtime in the larger context. In almost all cases, the change is imperceptible because the time was already small. However, for a few examples, the runtime explodes, *e.g.*, `list_compress`, `list_sorted_insert`, and `tree_binsert`. In these cases, the programs exhibit several levels of nested matches, coupled with large scrutinee sizes. In a larger context, Myth enumerates many more possible **`match`**es, many of which are not pruned by our branch informativeness heuristic.

There are some ways to mitigate this problem that we will explore in future work, *e.g.*, different parameter search strategies or optimizing term enumeration and evaluation further. However, these results demonstrate that the exponentially large search space still poses a fundamental problem, making some programs difficult or even impossible to synthesize given our current search strategy and available computing resources.

Surprising Synthesized Programs In most cases, Myth produces the program that we expect. However, in some cases it finds correct, but less obvious solutions. One frequent pattern is *inside-out recursion*, which matches on the results of a recursive call rather than performing the recursion inside the branches. It turns out that such a re-factoring saves an AST node, so Myth favors it in certain scenarios. A similar property allows Myth to find programs that are not so straightforward for people to understand.

```
let list_pairwise_swap : list -> list =
let rec f1 (l1:list) : list = match l1 with
 | Nil -> Nil
 | Cons (n1, l2) -> (match f1 l2 with
   | Nil -> (match l2 with
              | Nil -> Nil
              | Cons (n2, l3) ->
                Cons (n2, Cons (n1, f1 l3)))
   | Cons (n2, l3) -> Nil)
in f1
```

This program swaps adjacent pairs of elements for lists of even length, and returns `[]` for lists of odd length. The call `f1 l2` used as the scrutinee of the second **`match`** implicitly computes the length of `l2` (`Nil` means "even" and `Cons` means "odd"). Even though we are working in a small domain, Myth can still produce some results that we, as a human beings, would likely not derive on our own.

Prototype Limitations The Myth prototype has no built-in primitive datatypes (like `int` or `string`). We excluded them for simplicity and to focus on algebraic datatypes—other synthesis approaches such as those discussed in Section 6 are likely more suitable for numeric or other specialized domains. Our implementation also relies on a simple, unoptimized interpreter for the subset of OCaml that we support. Because evaluation is integral to the synthesis process, improving the interpreter could help increase performance significantly. Finally, Myth notably lacks support richer types, *e.g.*, products and polymorphic types. Thankfully, our type-directed style provides a simple way to integrate them into the system, something that we will pursue in future work.

6. Related Work and Conclusion

λ_{syn} and Myth begin to bridge the gap between the worlds of modern proof search and verification-based program synthesis techniques. No prior work has tried to synthesize typed functional programs with recursive data types utilizing types and examples in the style proposed here. Nevertheless, there is much related work that considers similar problems in isolation or in conjunction with other desiderata.

Proof- and Type-theoretic Synthesis At its core, λ_{syn} is a proof theory-inspired synthesis algorithm. We are certainly not the first to realize the potential of applying proof theory for program synthesis. Some of the earliest program synthesis work was rooted in resolution-based theorem proving, using axiomatic specifications [9] or even input–output examples [23]. Recent systems have embraced the availability of computational power compared to the 70s and explored enumerative, rather than deductive approaches. Djinn derives Haskell programs from types [4] (but not examples) according to a modification of Gentzen's LJ [8]. In contrast, Katayama's MagicHaskeller [13] enumerates programs using a set of logical rules and permitted components, caching them for later lookup. It is fast, but does not generate programs that branch. Finally, systems such as Igor2 [14] attempt to combine the best of the deductive and enumerative worlds. They differ from our work in that they do not take advantage of type information directly.

Beyond proof theory, many synthesis tools, in particular, auto-completion systems, take advantage of the specifications that types provide. Prospector [16], the type-directed auto-completion work of Perelman, *et al.* [19], and InSynth [12] all create code snippets by completing chains of library function calls. InSynth is particularly noteworthy because it also adapts the type inhabitation problem for synthesis. InSynth performs a backwards-style proof search similar to Myth. However, it neither takes nor refines additional forms of specification, *i.e.*, input–output examples.

Functional-Recursive Program Synthesis Several other tools work in the domain of functional-recursive program synthesis. Most notable are Escher [1] and Leon [15]. Our investigation into type-directed program synthesis started with the observation that Escher could benefit from explicit type information. In that sense, Myth can be thought of as an evolution of Escher. Although the two differ in that Escher builds terms from the bottom-up whereas Myth builds terms top-down.

Leon, being Scala-based, has type information available but it too does not push constraints inward. More broadly, Leon focuses on properties, rather than concrete examples, discharged via counterexample guided inductive synthesis (CEGIS) [22] and a SMT solver [5]. Other synthesis systems not necessarily focused on functional-recursive programs also use a solver, *e.g.*, Sketch [22] and Rosette [25]. We believe that a solver-based approach is largely orthogonal to our type-directed synthesis procedure. In particular, λ_{syn} shows how to push constraints (*i.e.*, types and examples) into the leaves of an expression. In contrast, solver-based algorithms generally bubble up constraints until they can be collected and discharged globally. Combining the two approaches, perhaps with additional insight from the syntax-guided synthesis efforts of Alur *et al.* [2] or Perelman *et al.* [20], seems like a fruitful way to get the best of both worlds.

Conclusion λ_{syn} is a core synthesis calculus that combines foundational techniques from proof theory with ideas from modern example-driven program synthesis. Our prototype, Myth, is competitive, both in terms of size and complexity, with other general-purpose functional program synthesizers. While Myth is only a prototype, our results show that type-directed program synthesis is a viable approach to synthesis that takes advantage of the rich structure present in typed, functional programming languages.

Acknowledgments

We would like to thank Jonathan Frankle, Rohan Shah, David Walker, the members of the UPenn PL Club, and the anonymous reviewers for their comments and feedback on our work. This work was supported by the Expeditions in Computer Augmented Program Engineering (ExCAPE, NSF Award CCF-1138996).

References

[1] Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. 2013. Recursive Program Synthesis. In *Proceedings of the 25th Conference on Computer-Aided Verification*.

[2] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishk Udupa. 2013. Syntax-guided Synthesis. In *13th International Conference on Formal Methods in Computer-aided Design*.

[3] Alan Ross Anderson, Nuel D. Belnap, and J. Michael Dunn. 1992. *Entailment: The logic of relevance and necessity, vol. II*. Princeton University Press, Princeton.

[4] Lennart Augustsson. 2004. [Haskell] Announcing Djinn, version 2004-12-11, a coding wizard. Mailing List. (2004). `http://www.haskell.org/pipermail/haskell/2005-December/017055.html`.

[5] Clark Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. 2008. *Satisfiability Modulo Theories*. IOS Press.

[6] John Byrnes. 1999. *Proof search and normal forms in natural deduction*. Ph.D. Dissertation. Carnegie Mellon University.

[7] Adam Chlipala. 2013. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. The MIT Press.

[8] Gerhard Gentzen. 1935. Untersuchungen uber das logische Schließen. II. *Mathematische Zeitschrift* 39, 1 (1935), 405–431.

[9] Cordell Green. 1969. Application of Theorem Proving to Problem Solving. In *International Joint Conference on Artificial Intelligence*.

[10] Katarzyna Grygiel and Pierre Lescanne. 2013. Counting and Generating Lambda Terms. *Journal of Functional Programming* 23 (9 2013), 594–628. Issue 05.

[11] Sumit Gulwani. 2011. Automating String Processing in Spreadsheets Using Input-output Examples. In *Proceedings of the 38th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*.

[12] Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. 2013. Complete Completion Using Types and Weights. In *Proceedings of the 2013 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[13] Susumu Katayama. 2012. An Analytical Inductive Functional Programming System That Avoids Unintended Programs. In *Proceedings of the ACM SIGPLAN 2012 Workshop on Partial Evaluation and Program Manipulation (PEPM '12)*. ACM, New York, NY, USA, 43–52.

[14] Emanuel Kitzelmann. 2010. *A Combined Analytical and Search-based Approach to the Inductive Synthesis of Functional Programs*. Ph.D. Dissertation. Fakul at f ur Wirtschafts-und Angewandte Informatik, Universit at Bamberg.

[15] Etienne Kneuss, Ivan Kuraj, Viktor Kuncak, and Philippe Suter. 2013. Synthesis Modulo Recursive Functions. In *Proceedings of the 28th ACM SIGPLAN on Object-oriented Programming Systems, Languages, and Applications (OOPSLA)*.

[16] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. 2005. Jungloid Mining: Helping to Navigate the API Jungle. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[17] Zohar Manna and Richard Waldinger. 1980. A Deductive Approach to Program Synthesis. *ACM Trans. Program. Lang. Syst.* 2, 1 (Jan. 1980), 90–121.

[18] Ulf Norell. 2007. *Towards a practical programming language based on dependent type theory*. Ph.D. Dissertation. Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden.

[19] Daniel Perelman, Sumit Gulwani, Thomas Ball, and Dan Grossman. 2012. Type-directed Completion of Partial Expressions. In *Proceedings of the 2012 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[20] Daniel Perelman, Sumit Gulwani, Dan Grossman, and Peter Provost. 2014. Test-driven Synthesis. In *Proceedings of the 2014 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[21] Benjamin C. Pierce and David N. Turner. 2000. Local Type Inference. *ACM Trans. Program. Lang. Syst.* 22, 1 (Jan. 2000), 1–44.

[22] Armando Solar-Lezama. 2008. *Program Synthesis by Sketching*. Ph.D. Dissertation. University of California, Berkeley.

[23] Phillip D. Summers. 1976. A Methodology for LISP Program Construction From Examples. In *Proceedings of the 3rd ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*.

[24] The Coq Development Team. 2012. *The Coq Proof Assistant: Reference Manual*. INRIA, http://coq.inria.fr/distrib/current/refman/.

[25] Emina Torlak and Rastislav Bodik. 2014. A Lightweight Symbolic Virtual Machine for Solver-aided Host Languages. In *Proceedings of the 2014 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[26] Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M.K. Martin, and Rajeev Alur. 2013. TRANSIT: Specifying Protocols with Concolic Snippets. In *Proceedings of the 2013 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.