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 (Green 1969; Manna and Waldinger 1980) 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 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;0].

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 12.

General program synthesis techniques of this kind have many potential applications. Recent success stories utilize synthesis in many scenarios: programming spreadsheet macros by example (Gulwani 2011); code completion in the context of large APIs or libraries (Perelman et al. 2012; Gvero et al. 2013); and generating cache-coherence protocols (Udupa et al. 2013), 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. [2013]) and the Leon system (by Kneuss et al. [2013]) 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.

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 (Byrnes 1999), a technique also used by Gvero et al. 2013.

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 naive 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 N₁ well-typed candidates for e₁ and N₂ candidates for e₂, following the naïve strategy of “enumerate then evaluate” would call our interpreter on all N₁ × N₂ possibilities. The smarter strategy of “evaluate as you enumerate” decomposes the problem into two independent sub-problems which finds the same answer using only N₁ + N₂ 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 refines 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: (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 : 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;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 only the 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:

\[ \text{match } l₁ \text{ with } \text{Nil } \rightarrow ? | \text{Cons}(n₁, l₂) \rightarrow ? \]

When pattern matching, we distribute the examples according to the branch of the match. In this case, l₁ evaluates to Nil in the first example 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 [0] (because l₁ has value [0] 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 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.

---

**2. Type-and-Example-directed Synthesis**

Consider a core ML-like language featuring algebraic data types, match, top-level function definitions, and explicitly recursive func-
\[ \tau ::= \pi \mid \tau_1 \rightarrow \tau_2 \\
\ e ::= C(e_1, \ldots, e_k) \mid e_1 e_2 \mid \text{fix}(f(x : \tau_1) : \tau_2 = e \mid pf) \mid \text{match} \ e \ 	ext{with} \ p_1 \rightarrow e_1 \mid \ldots \mid p_m \rightarrow e_m \]
\[ \ p ::= C(x_1, \ldots, x_n) \]
\[ \ u, v ::= C(v_1, \ldots, v_n) \mid \text{fix}(f(x : \tau_1) : \tau_2 = e \mid pf) \]
\[ \ ex ::= C(e_1, \ldots, e_k) \mid pf \]
\[ \ E ::= x \mid E I \\
\ I ::= E \mid C(I_1, \ldots, I_m) \mid \text{fix}(f(x : \tau_1) : \tau_2 = I) \mid \text{match} \ E \ 	ext{with} \ p_1 \rightarrow I_1 \mid \ldots \mid p_m \rightarrow I_m \\
\ \Gamma ::= \cdot \mid \ x : \tau, \Gamma \\
\ \Sigma ::= \cdot \mid C : \tau_1 \ast \ldots \ast \tau_n \rightarrow T, \Sigma \\
\ \sigma ::= \cdot \mid [u/x] \sigma \\
\ \ X ::= \cdot \mid \sigma \mapsto ex + X \]

\textbf{Figure 2}. $\lambda_{syn}$: syntax.

3. Synthesis Language

With a better intuition of how type-directed synthesis works, let’s examine the procedure in detail. The formalism $\lambda_{syn}$ 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 $\lambda_{syn}$: A Type-directed Synthesis Language

Figure 2 gives the syntax of $\lambda_{syn}$. 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 $\tau_1 \rightarrow \tau_2$. 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 $\text{fix}(f(x : \tau_1) : \tau_2 = e)$, and we write $\lambda x : \tau_1 . e$ instead when $f$ is not among the free variables of $e$. Signatures $\Sigma$ contain the set of declared constructors, and contexts $\Gamma$ contain information about the free variables of the program.

The only new syntactic form is that of partial functions $pf$, written $\tau \rightarrow pf <^m \tau$. Here, and throughout the paper, the overbar notation $\tau <^m \tau$ denotes a sequence of $m$ syntactic elements, $e_1 \ldots e_m$. Thus, the notation for a partial function $\sigma \mapsto ex \leftarrow e_i <^m \tau$ denotes a list of input–output examples $e_i \mapsto ex \leftarrow e_i \ldots \mapsto ex \leftarrow e_m$. We interpret such partial functions to mean that if we supply the argument $e_i$, then the partial function produces the result $ex$.

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 $\tau$ 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 $\sigma$ which maps variables to values. We can think of $\sigma$ as a substitution where $\sigma(e)$ denotes the expression obtained by applying the substitution $\sigma$ 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 $\beta$-normal form (Byrnes 1999). Such terms do not contain any $\beta$-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 $x_1 I_1 \ldots I_k$, with a free variable $x$ at its head.

This syntactic split carries through into the type system, where, as in bidirectional typechecking (Pierce and Turner 2000), the rules make explicit when we are checking types ($I$-forms) versus generating types ($E$-forms), respectively. We can think of type information flowing into $I$-forms whereas type informations flow 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 $\lambda_{syn}$ is divided among four judgments:

\[ \Sigma ; \Gamma \vdash e : \tau \quad e \text{ is well-typed.} \]
\[ \Sigma ; \Gamma \vdash E \Rightarrow \tau \quad E \text{ produces type } \tau. \]
\[ \Sigma ; \Gamma \vdash I \leftarrow \tau \quad I \text{ checks at type } \tau. \]
\[ \Sigma ; \Gamma \vdash X \rightarrow \tau \quad X \text{ checks at type } \tau. \]

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:

\[ \Sigma ; \Gamma \vdash e_1 : \tau_1 <^m \tau_2 \]
\[ \Sigma ; \Gamma \vdash e_2 : \tau_1 <^m \tau_2 \]
\[ \Sigma ; \Gamma \vdash e_3 : \tau_1 <^m \tau_2 \]
\[ \text{T}_{PF} \]

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

The final judgment typechecks examples $X$ at some goal type $\tau$. 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 $\tau$.

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:

\[ \Sigma ; \Gamma \vdash \tau \Rightarrow E \ (\text{EGuess}) \text{ guess an } E \text{ of type } \tau. \]
\[ \Sigma ; \Gamma \vdash \tau \leftarrow X \ (\text{IRefine}) \text{ refine and synthesize an } I \text{ of type } \tau \text{ that agrees with examples } X. \]

Note that these two judgments are very similar to the bidirectional typechecking system for this language! We have simply changed

\footnote{\ This fact is why match is an $I$-form: when typechecking a match we check the types of the branches against a given type.}
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 (Grygiel and Lescanne 2013). 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 _). 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 E is also syntactically considered I, 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 σ → ex where ex is the goal example value and σ is the execution environment that gives values to each of the free variables in I. For this to work, we enforce the following invariant in our system:

**Invariant 1.** Example-Value Consistency: ∀X : τ ∈ Γ. ∀σ → ex ∈ X. ∀v : τ. σ → v ∈ σ ∧ [v/x] ∈ σ.

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

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 values in the IREFINE_CTOR case. This refinement is delegated to

---

**Figure 3.** λsyn: typechecking rules.

**Figure 4.** λsyn: synthesis rules.
help 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 \( X \)'s, corresponding to the \( k \) arguments that must be synthesized for \( C \). The example values for \( X \) which corresponds to the \( k \)th argument are drawn from the \( k \)th example value from each of the example values found in the original \( X \). The environment \( \sigma \) are copied, unchanged, to each of the new \( X \).

apply assumes that the example values of \( X \) consist of partial functions. apply operates over a single example world \( \sigma \rightarrow pf \in X \) and produces a new \( X' \) that refines the examples for synthesizing the function body \( I \). If \( pf = pf = \tau \rightarrow ex < n \), then apply produces \( n \) new example worlds, one for each such \( v_i \Rightarrow ex \) pair. The example value for each of these worlds is \( ex \). To fulfill our invariance that the \( \sigma \) has a value for each variable bound in \( I \), we must provide values for \( f \) and \( x \). \( x \) is simply bound to \( v_i \). \( f \) is bound to the entire function from Section 2 but with the example set \( \{\} \rightarrow \{\} \). Instead, \( \sigma \rightarrow pf \) 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 \( \Sigma \), the data type \( T \) the match covers, the examples to refine \( X \), and the scrutinees 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 match pattern. Intuitively, distribute evaluates the scrutinees \( E \) using each of the \( \sigma \) drawn from the \( n \) worlds. Because \( E \) must be a data type \( T \), then it must evaluate to one of \( T \)'s constructors. distribute finds the example value to that constructor's branch in the match.

Finally, we bridge the refinement and guessing judgments with refine, 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 value. This “satisfies” relation, written \( \sigma \vdash ex \) ensures that for all example worlds \( \sigma \rightarrow ex \) in \( X \) that \( \sigma \rightarrow ex \rightarrow* v \) and \( v \equiv ex \), where \( \sigma \rightarrow ex \) substitutes the environment \( \sigma \) 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 \( \beta \)-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 %v →* 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 \( \{\} \rightarrow \{\}, \{1\} \rightarrow \{1\} \}. If we synthesized the term \%f \text{12} \} rather than Cons(\text{1}, Cons(\text{1}, \text{1} 12)), \} then we will encounter a NoMatch exception because \%12 \}. This is because our example set for \%1 contains no example for \%0. If we simply accepted \%f \text{12} 12, then we would admit a term that contradicted our examples since \%f \text{12 12 actually evaluates to \%1 once plugged into the overall recursive function.

Value compatibility, written \( v \equiv u \), 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 EQCTOR). 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 $\text{EQ}_\text{PP}_F$ (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 $\text{IREFINE}_G$ rule) involving a function \( f \) whose value is the partial function $\text{id} \Rightarrow \circ$. If \( f \) is applied to some function value \( v \) where \( v \) is contextually equivalent, yet syntactically distinct, from \( \text{id} \), evaluation will produce \text{NoMatch} rather than \( \circ \).

3.4 Metatheory

With the definition of $\lambda_{\text{syn}}$ in hand, we give an overview of the properties of $\lambda_{\text{syn}}$.

**Termination** Because synthesis relies on evaluation (through the compatibility relation), termination of any synthesis algorithm based on $\lambda_{\text{syn}}$ 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 (Norell 2007) and the Coq theorem prover (The Coq Development Team 2012). We elide the details of the structural recursion check here to simplify the presentation of $\lambda_{\text{syn}}$.

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

Checking for inconsistent examples proves to be difficult because of the undecidability of function equality in $\lambda_{\text{syn}}$. For example, consider the partial function $\text{id}1 \Rightarrow \circ \mid \text{id}2 \Rightarrow 1$ where $\text{id}1$ and $\text{id}2$ are syntactically distinct implementations of the identity function. This partial function has contradictory alternatives, however, we cannot determine that $\text{id}1 \simeq \text{id}2$.

Therefore, in $\lambda_{\text{syn}}$, we simply assume that the set of examples that the user provides is not contradictory. If $\lambda_{\text{syn}}$ 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 $\lambda_{\text{syn}}$ (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 $\lambda_{\text{syn}}$ ensures that synthesized programs are correct.

**Theorem 1. Type Soundness.** If $\Sigma ; \Gamma \vdash X : \tau$ and $\Sigma ; \Gamma \vdash \tau \triangleright X \simeq I$ then $\Sigma ; \Gamma \vdash I \triangleleft \tau$.

**Theorem 2. Example Soundness.** If $\Sigma ; \Gamma \vdash \tau \triangleright X \simeq I$ then $\Gamma \vdash X$.

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 $\text{IREFINE}$ rules manipulate the examples soundly. The example refinement performed by $\text{IREFINE}_\text{FIX}$, $\text{IREFINE}_\text{CTOR}$, and $\text{IREFINE}_\text{MATCH}$ all correspond to single-step evaluation for fix, constructor, and match expressions, respectively, over the examples. The base case, $\text{IREFINE}_\text{GUESS}$, ensures compatibility directly.

**Completeness** Completeness of $\lambda_{\text{syn}}$ ensures that we are able to synthesize all programs.

**Theorem 3. Completeness of Term Enumeration.**

1. If $\Sigma ; \Gamma \vdash E \Rightarrow \tau$ then $\Sigma ; \Gamma \vdash \tau \simeq E$.
2. If $\Sigma ; \Gamma \vdash I \triangleleft \tau$ then $\Sigma ; \Gamma \vdash \tau \triangleright \simeq I$.

In the absence of examples, $\lambda_{\text{syn}}$ can synthesize any well-typed term. This follows from the fact that the $\lambda_{\text{syn}}$ 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 $\lambda_{\text{syn}}$ satisfies the following property.

**Claim 1. Completeness of Synthesis.** If $\Sigma ; \Gamma \vdash X : \tau$, $\Sigma ; \Gamma \vdash I \triangleleft \tau$, $I \triangleright X$, then $\Sigma ; \Gamma \vdash \tau \triangleright \simeq I$.

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 $\lambda_{\text{syn}}$. The problem resides in our use of partial functions as the value for a recursive function during $\text{I-refinement}$. In the $\text{IREFINE}_\text{FIX}$ 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 \text{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, $\lambda_{\text{syn}}$, 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, $\lambda_{\text{syn}}$, itself is not a synthesis procedure on its own. We now discuss how we translate $\lambda_{\text{syn}}$ into an efficient synthesizer, \text{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 \text{MYTH}, like $\lambda_{\text{syn}}$, we are unable to detect upfront whether an input $X$ is consistent.

However, in \text{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 (The Coq Development Team 2012). 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 λsyn 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 λsyn 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:

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

that realize term enumeration in λsyn as a collection semantics that enumerates sets of E-terms E and I-terms I of type τ having exactly size n.

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 λsyn we E-guess under a context that grows as we synthesize under binders, i.e., functions and matches. To maximize cache hits, we should try to create invocations of gen_E and 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:

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

$$\text{gen}_I(\Sigma; \Gamma; x : \tau_1; \tau; n) = \text{gen}_{I \tau_1}(\Sigma; \Gamma; \tau; n)$$

This factorization ensures that for a given goal type and size, two calls to gen_E in different contexts Γ, x1 : τ1 and Γ, x2 : τ2 will still have a cache hit for the shared prefix context Γ.

Here, gen_{E \tau_1} and gen_{I \tau_1} are relevant term generation functions. Inspired by relevancy logic (Andersen et al. 1992), these functions require that all gathered terms must contain the relevant variable x.

Figure 7 gives the definition of gen_{E \tau_1}, the definition of gen_{I \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 function is function application which we enumerate in the recursive case of the gen_{E \tau_1}. Enumerating application terms consists of forming the Cartesian product ⊗ of applications from recursive calls both the gen_E and gen_I functions. In order to respect relevancy, 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 τ2.

### 4.3 Refinement Trees

λsyn 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
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 $\rightarrow$ 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 nil, 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 matching 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 IREFINECTOR, 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 $\rightarrow$ 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 (nil, Cons (nil, f1 12)) in the Cons branch by $E$-guessing nil and f1 12 at their respective nodes in the refinement tree, yielding the complete stutter program.

---

**Figure 8.** Example refinement tree for stutter.
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 \( \beta \)-normal, \( \eta \)-long terms without loss of expressiveness (up to \( \beta, \eta \)-equivalence).

5. Results

So far, we have shown how to turn \( \lambda_{syn} \), 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 \( \lambda_{syn} \) 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 matches too quickly may over-specialize the synthesized program. For example, synthesizing stutter starting from a refinement tree at match depth two yields:

```plaintext
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 \( \text{fix} \ l2 \)) 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 naive 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, \( \text{f} \) \( \text{e}_1 \) \( \text{e}_2 \), 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. 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 tree—pattern matching, and the creation and manipulation of higher-order functions. The tests include basic non-recursive functions (e.g., \( \text{bool_xor} \)), recursive functions (e.g., \( \text{list_length} \)), generation of higher-order recursive functions (e.g., \( \text{list_fold} \), \( \text{tree_map} \)), and use of higher-order functions (e.g., \( \text{list_sum} \)).

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.
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).

<table>
<thead>
<tr>
<th>Test</th>
<th>#Ex</th>
<th>#N</th>
<th>T-Ctx (sec)</th>
<th>T-Min (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Booleans</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bool_band</td>
<td>4</td>
<td>6</td>
<td>0.003</td>
<td>0.002</td>
</tr>
<tr>
<td>bool_bor</td>
<td>4</td>
<td>6</td>
<td>0.004</td>
<td>0.001</td>
</tr>
<tr>
<td>bool_impl</td>
<td>4</td>
<td>6</td>
<td>0.004</td>
<td>0.002</td>
</tr>
<tr>
<td>bool_neg</td>
<td>2</td>
<td>5</td>
<td>0.001</td>
<td>0.0</td>
</tr>
<tr>
<td>bool_xor</td>
<td>4</td>
<td>4</td>
<td>0.003</td>
<td>0.002</td>
</tr>
<tr>
<td><strong>Lists</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>list_append</td>
<td>12</td>
<td>12</td>
<td>0.011</td>
<td>0.003</td>
</tr>
<tr>
<td>list_compress</td>
<td>13</td>
<td>28</td>
<td>128.339</td>
<td>0.073</td>
</tr>
<tr>
<td>list_concat</td>
<td>6</td>
<td>11</td>
<td>0.019</td>
<td>0.006</td>
</tr>
<tr>
<td>list_drop</td>
<td>13</td>
<td>13</td>
<td>1.29</td>
<td>0.013</td>
</tr>
<tr>
<td>list_even_parity</td>
<td>7</td>
<td>13</td>
<td>0.578</td>
<td>0.004</td>
</tr>
<tr>
<td>list_filter</td>
<td>10</td>
<td>17</td>
<td>4.320</td>
<td>0.067</td>
</tr>
<tr>
<td>list_fold</td>
<td>9</td>
<td>13</td>
<td>0.504</td>
<td>0.139</td>
</tr>
<tr>
<td>list_hud</td>
<td>3</td>
<td>5</td>
<td>0.019</td>
<td>0.001</td>
</tr>
<tr>
<td>list_int</td>
<td>4</td>
<td>8</td>
<td>0.004</td>
<td>0.0</td>
</tr>
<tr>
<td>list_list</td>
<td>6</td>
<td>11</td>
<td>0.093</td>
<td>0.0</td>
</tr>
<tr>
<td>list_length</td>
<td>3</td>
<td>8</td>
<td>0.019</td>
<td>0.001</td>
</tr>
<tr>
<td>list_map</td>
<td>8</td>
<td>12</td>
<td>0.082</td>
<td>0.008</td>
</tr>
<tr>
<td>list_nth</td>
<td>24</td>
<td>16</td>
<td>0.96</td>
<td>0.013</td>
</tr>
<tr>
<td>list_pairwise_swap</td>
<td>20</td>
<td>19</td>
<td>10.227</td>
<td>0.007</td>
</tr>
<tr>
<td>list_rev_append</td>
<td>5</td>
<td>13</td>
<td>0.028</td>
<td>0.011</td>
</tr>
<tr>
<td>list_rev_fold</td>
<td>5</td>
<td>12</td>
<td>0.014</td>
<td>0.007</td>
</tr>
<tr>
<td>list_rev_snoc</td>
<td>5</td>
<td>11</td>
<td>0.018</td>
<td>0.006</td>
</tr>
<tr>
<td>list_rev_tailcall</td>
<td>14</td>
<td>12</td>
<td>0.004</td>
<td>0.004</td>
</tr>
<tr>
<td>list_snoc</td>
<td>8</td>
<td>14</td>
<td>0.07</td>
<td>0.003</td>
</tr>
<tr>
<td>list_sort_sorted insert</td>
<td>7</td>
<td>11</td>
<td>0.009</td>
<td>0.008</td>
</tr>
<tr>
<td>list_sorted_insert</td>
<td>12</td>
<td>24</td>
<td>22.016</td>
<td>0.122</td>
</tr>
<tr>
<td>list_stutter</td>
<td>3</td>
<td>11</td>
<td>0.018</td>
<td>0.001</td>
</tr>
<tr>
<td>list_sum</td>
<td>3</td>
<td>8</td>
<td>0.005</td>
<td>0.002</td>
</tr>
<tr>
<td>list_take</td>
<td>12</td>
<td>15</td>
<td>1.147</td>
<td>0.112</td>
</tr>
<tr>
<td>list_tl</td>
<td>3</td>
<td>5</td>
<td>0.02</td>
<td>0.001</td>
</tr>
<tr>
<td><strong>Nats</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nat_add</td>
<td>9</td>
<td>11</td>
<td>0.023</td>
<td>0.002</td>
</tr>
<tr>
<td>nat_even</td>
<td>4</td>
<td>10</td>
<td>0.014</td>
<td>0.001</td>
</tr>
<tr>
<td>nat_max</td>
<td>9</td>
<td>14</td>
<td>0.002</td>
<td>0.011</td>
</tr>
<tr>
<td>nat_pred</td>
<td>3</td>
<td>5</td>
<td>0.004</td>
<td>0.001</td>
</tr>
<tr>
<td><strong>Trees</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tree_insert</td>
<td>20</td>
<td>31</td>
<td>9.034</td>
<td>0.374</td>
</tr>
<tr>
<td>tree_collect_leaves</td>
<td>6</td>
<td>15</td>
<td>0.033</td>
<td>0.016</td>
</tr>
<tr>
<td>tree_count_leaves</td>
<td>7</td>
<td>14</td>
<td>0.042</td>
<td>0.008</td>
</tr>
<tr>
<td>tree_count_nodes</td>
<td>6</td>
<td>14</td>
<td>0.037</td>
<td>0.009</td>
</tr>
<tr>
<td>tree_inorder</td>
<td>5</td>
<td>15</td>
<td>0.036</td>
<td>0.012</td>
</tr>
<tr>
<td>tree_map</td>
<td>7</td>
<td>15</td>
<td>0.035</td>
<td>0.014</td>
</tr>
<tr>
<td>tree_nodes_at_level</td>
<td>24</td>
<td>22</td>
<td>4.917</td>
<td>1.093</td>
</tr>
<tr>
<td>tree_postorder</td>
<td>9</td>
<td>32</td>
<td>7.929</td>
<td>1.136</td>
</tr>
<tr>
<td>tree_prefix</td>
<td>5</td>
<td>15</td>
<td>0.023</td>
<td>0.009</td>
</tr>
</tbody>
</table>

Figure 10. Aggregated extended examples results.

<table>
<thead>
<tr>
<th>Test</th>
<th>#Ex</th>
<th>#N</th>
<th>T (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>arith</td>
<td>22</td>
<td>47</td>
<td>11.139</td>
</tr>
<tr>
<td>dyn_app_twice</td>
<td>6</td>
<td>11</td>
<td>0.715</td>
</tr>
<tr>
<td>dyn_sum</td>
<td>25</td>
<td>23</td>
<td>19.420</td>
</tr>
<tr>
<td>fvs_large</td>
<td>31</td>
<td>75</td>
<td>3.905</td>
</tr>
<tr>
<td>fvs_medium</td>
<td>22</td>
<td>45</td>
<td>0.751</td>
</tr>
<tr>
<td>fvs_small</td>
<td>6</td>
<td>16</td>
<td>0.029</td>
</tr>
</tbody>
</table>

an infinite loop by breaking the positivity restriction is more difficult by comparison. For example, consider the `type` field:

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

which represents an encoding of type dynamic over nats and functions. The simplest such program (Chlipala 2013) requires calling the `bad` function:

```haskell
let rec bad (d:dyn) = dyn ~ Error |
match d with |
| Error -> Error |
| Dyn (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 `nats`. 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 (Albarghouthi et al. 2013) and Leon (Kneuss et al. 2013), 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, div, and and. 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. [2013], 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_binset. 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 matches, 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-factorings 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 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 [Green 1969] or even input–output examples [Summers 1976]. 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 [Augustsson 2004] (but not examples) according to a modification of Gentzen’s LJ [Gentzen 1935]. In contrast, Katayama’s MagicHaskeller [2012] 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 [Kitzelmann 2010] 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, automatic-completion systems, take advantage of the specifications that types provide. Prospector [Mandelin et al. 2005], the type-directed auto-completion work of Perelman, et al. [2012], and InSynth (Gvero et al. 2013) 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 (Albarghouthi et al. 2013) and Leon (Kneuss et al. 2013). 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) (Solar-Lezama 2008) and a SMT solver (Barrett et al. 2008). Other synthesis systems not necessarily focused on functional-recursive programs also use a solver, e.g., Sketch (Solar-Lezama 2008) and Rosette (Torlak and Bodik 2014). We believe that a solver-based approach is largely orthogonal to our type-directed synthesis procedure. In particular, $\lambda_{\text{syn}}$ 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. [2013] or Perelman et al. [2014], seems like a fruitful way to get the best of both worlds.

**Conclusion**

$\lambda_{\text{syn}}$ 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**


