Program Sketching with Live Bidirectional Evaluation

JUSTIN LUBIN, University of Chicago, USA
NICK COLLINS, University of Chicago, USA
CYRUS OMAR, University of Michigan, USA
RAVI CHUGH, University of Chicago, USA

We present a system called Smyth for program sketching in a typed functional language whereby the concrete evaluation of ordinary assertions gives rise to input-output examples, which are then used to guide the search to complete the holes. The key innovation, called *live bidirectional evaluation*, propagates examples "backward" through partially evaluated sketches. Live bidirectional evaluation enables Smyth to (a) synthesize recursive functions without trace-complete sets of examples and (b) specify and solve interdependent synthesis goals. Eliminating the trace-completeness requirement resolves a significant limitation faced by prior synthesis techniques when given partial specifications in the form of input-output examples.

To assess the practical implications of our techniques, we ran several experiments on benchmarks used to evaluate Myth, a state-of-the-art example-based synthesis tool. First, given expert examples (and no partial implementations), we find that Smyth requires on average 66% of the number of expert examples required by Myth. Second, we find that Smyth is robust to randomly-generated examples, synthesizing many tasks with relatively few more random examples than those provided by an expert. Third, we create a suite of small sketching tasks by systematically employing a simple sketching strategy to the Myth benchmarks; we find that user-provided sketches in Smyth often further reduce the total specification burden (i.e. the combination of partial implementations and examples). Lastly, we find that Leon and Synquid, two state-of-the-art logic-based synthesis tools, fail to complete several tasks on which Smyth succeeds.

CCS Concepts: • **Software and its engineering** → *General programming languages*; *Programming by example*; *Search-based software engineering*; *Automatic programming*; • **Theory of computation** → *Type theory*.

Additional Key Words and Phrases: Program Synthesis, Sketches, Examples, Bidirectional Evaluation

ACM Reference Format:
Justin Lubin, Nick Collins, Cyrus Omar, and Ravi Chugh. 2020. Program Sketching with Live Bidirectional Evaluation. *Proc. ACM Program. Lang.* 4, ICFP, Article 109 (August 2020), 29 pages. https://doi.org/10.1145/3408991

1 INTRODUCTION

Program synthesis is closer than ever to making its way into the working programmer's toolbox. Synthesis techniques that operate on fine-grained logical specifications—such as Sketch [Solar-Lezama 2008], Rosette [Torlak and Bodik 2013], Leon [Kneuss et al. 2013], and Synquid [Polikarpova et al. 2016]—as well as techniques that operate on input-output examples—such as Escher [Albarghouthi et al. 2013], λ^2 [Feser et al. 2015] Myth [Osera and Zdancewic 2015], and "Myth2" [Frankle et al. 2016]—can synthesize a variety of challenging tasks, from subtle bit-manipulating computations in imperative languages to recursive functions over inductive datatypes in functional languages.

Authors' addresses: Justin Lubin, University of Chicago, USA, justinlubin@uchicago.edu; Nick Collins, University of Chicago, USA, nickmc@uchicago.edu; Cyrus Omar, University of Michigan, USA, comar@umich.edu; Ravi Chugh, University of Chicago, USA, rchugh@cs.uchicago.edu.

2475-1421/2020/8-ART109
https://doi.org/10.1145/3408991

```
stutter_n : Nat -> NatList -> NatList
stutter_n n xs =
  case xs of
    []     -> []
    x::xs' -> replicate n x ++ stutter_n n xs'

replicate : Nat -> Nat -> NatList
replicate n x =
  case n of
    Z    -> ??
    S n' -> ??

assert (stutter_n 1 [1, 0] == [1, 0])
assert (stutter_n 2 [3]    == [3, 3])
```

Fig. 1. A program sketch in Smyth to "stutter" each element of a list `n` times. The desired solutions for the holes in `replicate` are `[]` for the `Z` branch and `x :: replicate n' x` for the `S` branch.

However, there remain commonplace program synthesis tasks that cannot be completed by state-of-the-art techniques. Figure 1 shows an incomplete program (a.k.a. "program sketch"), written in an ML-style functional language. The implementation of the `stutter_n` function itself—which is intended to "stutter" each element of a given list `n` times—is complete. However, it depends on an incomplete helper function `replicate` with *holes* (written ??) denoting missing expressions that the programmer might hope to automatically synthesize. The two `assert` statements provide simple test cases that constrain the behavior of `stutter_n`. Because `stutter_n` applies `replicate`, these assertions indirectly constrain the holes in `replicate` as well. Unfortunately, the aforementioned synthesis techniques are not able to synthesize the desired hole completions shown in blue boxes in Figure 1. In what ways do the prior techniques fall short for this task?

Logic-Based Program Synthesis. Leon [Kneuss et al. 2013] and Synquid [Polikarpova et al. 2016] support sketching for richly-typed, general-purpose functional languages (as used in Figure 1). As pioneered in Sketch [Solar-Lezama 2008], Leon and Synquid are solver-based techniques that fill holes such that given specifications are satisfied. Both systems synthesize many challenging benchmarks involving complex data invariants, yet neither can complete the task in Figure 1.

The approach to synthesis and verification in Leon does not decompose the `assert` constraints on `stutter_n` into constraints on `replicate`, so the holes remain unspecified. By using an approach based on *liquid types* [Rondon et al. 2008; Vazou et al. 2013], Synquid is able to systematically decompose the given constraints into the following specification:

```
replicate :: (n : Nat) -> (x : Nat)
          -> { out : NatList | (n = 1 ∧ x = 0 ⇒ out = [0])
                             ∧ (n = 1 ∧ x = 1 ⇒ out = [1])
                             ∧ (n = 2 ∧ x = 3 ⇒ out = [3, 3]) }
```

However, because this specification is not inductive—it provides no information about `replicate 0 0`, `replicate 0 1`, `replicate 1 3`, or `replicate 0 3`—Synquid cannot type check the desired solution for `replicate`, let alone synthesize it.

Evaluator-Based Program Synthesis. In contrast to logic-based techniques, another class of techniques operate on input-output examples and rely on concrete evaluation to "guess-and-check" candidate terms. We choose the term *evaluator-based* to describe such techniques—rather than *example-based* or *programming-by-example*—to distinguish how the underlying algorithms work (using concrete evaluation) from the specification mechanism they provide to users (examples). Examples can also be encoded as partial logical specifications, as just discussed.

Among evaluator-based techniques, Escher [Albarghouthi et al. 2013] and Myth [Osera and Zdancewic 2015] can synthesize recursive functions, and Myth employs several type-directed

optimizations to navigate the search space. (We discuss the remaining systems in §7.) However, there are two fundamental reasons why these tools cannot complete the task in Figure 1.

Limitation A: Trace-Complete Examples. The user must provide input-output examples for recursive calls internal to the eventual solution—this is the "example analog" to Synquid's requirement for inductive logical specifications. Osera and Zdancewic [2015] acknowledge that providing *trace-complete examples* (i.e. serving as an *oracle* [Albarghouthi et al. 2013]) "proved to be difficult initially" even for experts, and "discovering ways to get around this restriction ... would greatly help in converting this type-directed synthesis style into a usable tool." Miltner et al. [2020] also observe the need to "manage Myth's requirement for trace completeness."

Limitation B: Independent, Top-Level Goals. The user must factor all synthesis tasks into completely unimplemented top-level functions, each of which must be equipped directly with (trace-complete) example sets. The system attempts to synthesize each of these functions separately. Granular sketching, where holes appear in arbitrary positions and are simultaneously solved, is not supported.

Our Approach: Live Bidirectional Evaluation. In this paper, we present a new evaluator-based synthesis technique that addresses Limitations A and B. Holes can appear in arbitrary expression positions and are constrained by types and `assert` statements which give rise to example constraints. Given the sketch in Figure 1, our implementation—called Smyth—synthesizes the desired expressions to fill the holes. (Our exposition employs certain syntactic conveniences not currently implemented. These are described in §5.)

In order to make evaluator-based synthesis techniques compatible with sketching, we must formulate hole-aware notions of (1) *concrete evaluation* and (2) *example satisfaction*—which form the central term enumeration search strategy (i.e. *guess-and-check*) for evaluator-based synthesis. Our solution, called *live bidirectional evaluation*, comprises two parts:

(1) A *live evaluator* $e \Rightarrow r$ that partially evaluates a sketch e by proceeding around holes, producing a result r which is either a value or a "paused" expression that, when the necessary holes are filled, will "resume" evaluating; and

(2) A *live unevaluator* $r \Leftarrow ex \dashv K$ that, given a result r to be checked against example ex, computes constraints K (over possibly many holes in the sketch) that, if satisfied, ensure the result will eventually produce a value satisfying ex.

Live evaluation is adapted from Omar et al. [2019] to our setting and is not a technical contribution of our work. Live unevaluation is the key novel mechanism that—together with live evaluation—enables us to "combine sketching with Myth-style synthesis" (hence the name Smyth). Compared to the aforementioned logic-based and other symbolic evaluation techniques (e.g. [Bornholt and Torlak 2018; Feng et al. 2017a; Wang et al. 2020]), live bidirectional evaluation employs *concrete* evaluation to collect example constraints "globally" across multiple holes in the sketch.

Contributions. This paper generalizes the theory of Myth [Osera and Zdancewic 2015]—the state-of-the-art in type-directed, evaluator-based program synthesis—to support sketches and live bidirectional evaluation. Formally, we present a calculus of recursive functions, algebraic datatypes, and holes—called Core Smyth—which includes the following technical contributions:

- We present *live unevaluation*, a novel technique that checks example satisfaction of sketches. The combination of *live evaluation* to partially evaluate sketches [Omar et al. 2019] and live unevaluation—which we call *live bidirectional evaluation*—forms a core guess-and-check strategy for programs with holes. Our formulation generalizes Myth, but the notion of live bidirectional evaluation can also be developed for other evaluator-based synthesizers. (§3.5)

- We use live bidirectional evaluation to simplify program assertions into input-output constraints and generalize the Myth hole synthesis algorithm to employ live bidirectional evaluation. The resulting synthesis algorithm (a) alleviates the trace-completeness requirement and (b) globally solves the examples that arise from multiple interdependent tasks. (§4)

For simplicity, our formal system accounts only for top-level `assert`s, but we describe how subsequent work may extend our approach to allow assertions in arbitrary program positions.

To empirically evaluate our approach, we implement Smyth and perform several experiments:

- We synthesize 38 of 43 tasks from the Myth benchmark suite [Osera 2015; Osera and Zdancewic 2015] in Smyth. Given expert examples (without sketches), Smyth requires 66% of the number of expert examples required by Myth. Moreover, Smyth typically requires only a slightly larger set of examples if they are generated randomly, rather than by an expert. (§6.2)
- To create a suite of sketching tasks, we identify a simple *base case sketching strategy* and apply it systematically to the Myth benchmarks. As expected, base case sketches further reduce the number of examples that Smyth requires to complete many tasks. Furthermore, the total specification size with sketching (partial implementation plus examples) is often smaller than without (just examples). (§6.3)
- We identify a handful of additional sketching tasks, similar in size and flavor to `stutter_n`, which Smyth can complete. (§2)
- To situate our experimental results in a broader context, we run Leon and Synquid on our benchmarks. We find several tasks for which Smyth succeeds but these tools do not. (§6.4)

The experimental results demonstrate (i) that the theoretical advances in Smyth address Limitations A and B of prior evaluator-based synthesizers, and (ii) that even though examples can generally be encoded as logical specifications, current logic-based synthesizers are not necessarily strictly more powerful than evaluator-based ones.

Because our approach generalizes Myth, we provide comparison throughout the paper. We further discuss related work in §7. Additional definitions, proofs, and experimental data are available in an extended technical report [Lubin et al. 2020]; in the rest of the paper, we write §A, §B, and §C to refer to appendices in the technical report.

2 OVERVIEW

In this section, we work through several small programs to introduce how Smyth: (1) employs live bidirectional evaluation to check example satisfaction of guessed expressions (which, in our formulation, may include holes) (§2.1); (2) supports user-defined sketches (§2.2); and (3) derives examples from `assert`s in the program (§2.3).

We write holes $??_h$ below with explicit names h; our implementation automatically generates names for holes as in Figure 1. Literals `0`, `1`, `2`, etc. are syntactic sugar for the corresponding naturals of `type Nat = Z | S Nat`. Some judgement forms below are simplified for expositional purposes.

2.1 Synthesis without Trace-Completeness

Consider the task to synthesize `plus` given the three test cases on the right. Given this specification, the resulting *example constraint* $K_0 = (- \vdash \bullet_0 \models \{0\ 1 \to 1, 2\ 0 \to 2, 1\ 2 \to 3\})$ requires that $??_0$ (hole name 0 generated for the definition of `plus`) be filled with a function expression that, in the empty environment, $-$, conforms to the given input-output examples. (We write $\bullet_h$ to distinguish the concrete syntax of constraints from expression holes $??_h$.)

```
plus : Nat -> Nat -> Nat
plus = ??

assert (plus 0 1 == 1)
assert (plus 2 0 == 2)
assert (plus 1 2 == 3)
```

Given a set of constraints K_h, Smyth employs the *hole synthesis* search procedure $K_h \rightsquigarrow e_h \dashv K'$ to fill the hole $??_h$ with an expression e_h that is valid assuming new constraints K' over other holes in the program. Following Myth [Osera and Zdancewic 2015], hole synthesis begins with a guess-and-check approach that enumerates increasingly large terms comprising variables and functions applied to variables. This naïve search is limited to small terms, i.e., starting with AST size 1 in early "stages" of the search and increasing to size 13 in latter stages. When enumerative search fails to find a solution in a particular stage, hole synthesis performs *example-directed refinement and branching*: introductory forms and case analyses are considered, and the examples are *distributed* to create subgoals for new holes that arise.

We will describe the following search path—among many that Smyth will consider—that yields the solution `fix plus λm n -> case m of {Z -> n; S m' -> S (plus m' n)}` for `plus`.

$$\begin{array}{rlll} K_0 & \rightsquigarrow_{\text{refine}} & ??_0 = \texttt{fix plus } (\lambda \texttt{m n}.\, ??_1) & \dashv\ K_1 \\ K_1 & \rightsquigarrow_{\text{branch}} & ??_1 = \texttt{case m } \{\texttt{Z} \rightarrow ??_2\,;\, \texttt{S m'} \rightarrow ??_3\} & \dashv\ K_2, K_3 \\ K_3 & \rightsquigarrow_{\text{refine}} & ??_3 = \texttt{S } ??_4 & \dashv\ K_4 \\ K_4 & \rightsquigarrow_{\text{guess}} & ??_4 = \texttt{plus m' n} & \dashv\ K_2' \\ K_2, K_2' & \rightsquigarrow_{\text{guess}} & ??_2 = \texttt{n} & \dashv\ - \end{array}$$

First, because the goal is a function type, Smyth synthesizes a recursive function literal, with subgoal $??_1$ for the body. The constraint set K_1 (not shown) consists of three constraints created from the three input-output examples in K_0 by binding the input values to `m` and `n` in the environment and constraining the new subgoal with the corresponding output value.

Second, after guessing-and-checking fails to solve $??_1$, Smyth attempts to *branch* by guessing the scrutinee `m`. This scrutinee is evaluated in each environment of the three constraints in K_1. One constraint from K_1 is distributed to subgoal $??_2$ for the base case branch (this constraint $K_{2.1}$ is shown below), and the other two constraints from K_1 are distributed to subgoal $??_3$ for the recursive case (these constraints K_3 are not shown).

Third, Smyth chooses to work on the recursive branch, for which the two constraints in K_3 involve output examples `2` and `3` (i.e. `S (S Z)` and `S (S (S Z))`). Smyth *refines* the task by synthesizing the literal `S` $??_4$; the new subgoal is constrained by two examples (in K_4, shown below) obtained by removing the shared constructor head `S` from the output examples in K_3. (Smyth synthesizes a literal of the form `S (S` $??_4$`)` along other search paths, but those paths do not yield a solution as quickly as the one being described.)

$$\begin{aligned} K_{2.1} &= ((\texttt{plus} \mapsto \ldots, \texttt{m} \mapsto \texttt{0}, \texttt{n} \mapsto \texttt{1} \quad\quad\ ) \vdash \bullet_2 \models \texttt{1}) \\ K_{4.1} &= ((\texttt{plus} \mapsto \ldots, \texttt{m} \mapsto \texttt{2}, \texttt{n} \mapsto \texttt{0}, \texttt{m'} \mapsto \texttt{1}) \vdash \bullet_4 \models \texttt{1}) \\ K_{4.2} &= ((\texttt{plus} \mapsto \ldots, \texttt{m} \mapsto \texttt{1}, \texttt{n} \mapsto \texttt{2}, \texttt{m'} \mapsto \texttt{0}) \vdash \bullet_4 \models \texttt{2}) \end{aligned}$$

The remaining two subgoals, $??_4$ and $??_2$, are filled via guess-and-check as discussed below.

Live Bidirectional Example Checking. To decide whether a guessed expression e conforms to a constraint $(E \vdash \bullet_h \models ex)$ in Smyth, the procedure $E\,e \Rightarrow r$ applies the substitution (i.e. environment) E to the expression and evaluates it to a result r, and the *live unevaluation* procedure $r \Leftarrow ex \dashv K$ checks satisfaction modulo new constraints K.

Consider guesses to fill $??_4$. Notice that `plus`—the function Smyth is working to synthesize—is recursive and thus bound in the constraint environments above. In addition to variables and calls to existing functions, Smyth enumerates structurally-decreasing recursive calls (`plus m' n`, `plus m n'`, and `plus m' n'`).

When considering `plus m' n`, the name `plus` binds the following value comprising the first three fillings and the "current" guess:

$$\texttt{fix plus } (\lambda \texttt{m n.case m } \{\texttt{Z} \to ??_2 \,;\, \texttt{S m'} \to \texttt{S (plus m' n)}\})$$

Given the environment in constraint $K_{4.1}$, the guess evaluates and unevaluates as follows:

$$\begin{aligned} \texttt{plus m' n} &\to^* \texttt{plus 1 0} \\ &\to^* \texttt{S (plus 0 0)} \\ &\Rightarrow \texttt{S } ([(\texttt{plus} \mapsto \ldots, \texttt{m} \mapsto \texttt{0}, \texttt{n} \mapsto \texttt{0})]\, ??_2) \;\Leftarrow\; \texttt{1} \;\dashv\; K_{2.2} \end{aligned}$$

(We write $e \to^* e' \Rightarrow r$ to display intermediate steps of the big-step evaluation, but $e \to^* e'$ does not appear in the formal system.) Although the function is incomplete, *live evaluation* [Omar et al. 2019] resolves two recursive calls to `plus`, before the hole $??_2$ in the base case reaches evaluation position; the resulting *hole closure*, of the form $[E]\, ??_h$, captures the environment at that point. Comparing the result to `1` (i.e. `S Z`), unevaluation removes an `S` from each side and creates a new constraint $K_{2.2}$ (shown below) for the base case.

Similarly, the guess checks against constraint $K_{4.2}$, adding another new constraint $K_{2.3}$ (shown below) on the base case.

$$\begin{aligned} \texttt{plus m' n} &\to^* \texttt{plus 0 2} \\ &\Rightarrow [(\texttt{plus} \mapsto \ldots, \texttt{m} \mapsto \texttt{0}, \texttt{n} \mapsto \texttt{2})]\, ??_2 \;\Leftarrow\; \texttt{2} \;\dashv\; K_{2.3} \end{aligned}$$

Both checks succeed, so the fourth step of the search commits to the guess, returning the two new constraints in K_2'.

$$K_{2.2} = ((\texttt{plus} \mapsto \ldots, \texttt{m} \mapsto \texttt{0}, \texttt{n} \mapsto \texttt{0}) \vdash \bullet_2 \models \texttt{0})$$
$$K_{2.3} = ((\texttt{plus} \mapsto \ldots, \texttt{m} \mapsto \texttt{0}, \texttt{n} \mapsto \texttt{2}) \vdash \bullet_2 \models \texttt{2})$$

The fifth and final step is to fill the base case $??_2$, subject to constraints $K_{2.1}$, $K_{2.2}$, and $K_{2.3}$. The guess `n` evaluates to the required values (`0`, `1`, and `2`, respectively), without assumption. Together, the five filled holes comprise the final solution.

Notice that the test cases used to synthesize `plus` were *not* trace-complete: live bidirectional example checking recursively called `plus 1 0`, `plus 0 0`, and `plus 0 2`, none of which were included in the examples. Instead, Smyth *generated* additional constraints that the user would be required to provide in prior systems (i.e. Escher, Myth, Myth2, and Synquid).

2.2 User-Defined Sketches

Smyth is the first evaluator-based synthesis technique to support sketching, thus allowing users to split domain knowledge naturally across a partial implementation and examples. For instance, if the user sketches the zero cases for `max`, as shown in Figure 2, just a few examples are sufficient for Smyth to complete the recursive case. (The library function `spec2` asserts input-output examples for a binary function, as was written out fully for `plus` above.)

Sketches from the user are handled in the same way as the sketches, described above, created internally by the Smyth algorithm. Myth and several other evaluator-based techniques (cf. §7) can also be described as creating sketches internally, but Smyth uniquely supports *concrete evaluation* of sketches—with holes in arbitrary positions—as a way to generate new example constraints.

2.3 Deriving Examples from Assertions

For the `plus` and `max` programs so far, evaluating assertions provided examples "directly" on holes. In general, however, an assertion may involve more complicated results.

```
max  m      Z     = m
max  Z      n     = n
max (S m') (S n') = S (max m' n')

spec2 max
  [(1, 1, 1), (1, 2, 2), (3, 1, 3)]
```

```
odd n =
  case n of
    Z      -> False
    S Z    -> True
    S S n'' -> odd n''

assert (odd (unJust Just 1 ) == True)
```

```
minus (S a') (S b') = minus a' b'
minus  a      b     = a

spec2 minus
  [(2, 0, 2), (3, 2, 1), (3, 1, 2)]
```

```
mult p q =
  case p of
    Z    -> Z
    S p' -> plus q (mult p' q )

spec2 mult
  [(2, 1, 2), (3, 2, 6)]
```

```
unJust mx =
  case mx of
    Nothing -> 0
    Just x  -> x
```

Fig. 2. Smyth fills the holes ?? (not shown) with the code shown in blue boxes.

For instance, consider the definitions of `odd : Nat -> Bool` and `unJust : MaybeNat -> Nat` in Figure 2, and the evaluation of the expression `odd (unJust ??`$_5$`)`:

$$\begin{aligned} \texttt{odd}\ (\texttt{unJust}\ ??_5) &\rightarrow^* \texttt{odd}\ (\texttt{unJust}\ ([-]\,??_5)) \\ &\rightarrow^* \texttt{odd}\ (\texttt{case}\ ([-]\,??_5)\ \mathit{unJust}) \\ &\Rightarrow \texttt{case}\ (\texttt{case}\ ([-]\,??_5)\ \mathit{unJust})\ \mathit{odd} \end{aligned}$$

(For clarity, we omit the recursive environment bindings for odd and unJust.) First, evaluation produces the hole closure $[-]\,??_5$, which is passed to unJust. Then, the case expression in unJust—we write *unJust* to refer to its two branches—scrutinizes the hole closure. The form of the constructor application has not yet been determined, so evaluation "pauses" by returning the *indeterminate* [Omar et al. 2019] result `case` $([-]\,??_5)$ *unJust*, which records the fact that, when the scrutinee resumes to a constructor head `Nothing` or `Just`, evaluation of the case will proceed down the appropriate branch. This indeterminate case result is passed to the odd function. Finally, the case inside odd—we write *odd* to refer to its three branches—scrutinizes it, building up a nested indeterminate result.

How can we "indirectly" constrain the expression $??_5$ to ensure that the partially evaluated expression `case (case` $([-]\,??_5)$ *unJust*`)` *odd* evaluates to `True` as asserted?

Unevaluating Case Expressions. Unevaluation will run each of the three branches of *odd* "in reverse," attempting to reconcile each with the required example, `True`; we write ①, ②, ③, etc. to help discuss different branches of the search considered by Smyth:

$$\texttt{case}\ (\texttt{case}\ ([-]\,??_5)\ \mathit{unJust})\ \mathit{odd} \Leftarrow \texttt{True} \dashv ①②③$$

① The first branch expression, `False`, is inconsistent with `True` (i.e. `False` $\Leftarrow$ `True` $\not\dashv$).

② The second branch expression, `True`, is equal to the example. However, to take this branch, unevaluation must ensure that the scrutinee—an indeterminate case result itself—will match the pattern `S Z` (i.e. `1`); that is, `case` $([-]\,??_5)$ *unJust* $\Leftarrow$ `1` $\dashv$ ②a②b.

②a The first branch expression, `0`, is inconsistent with `1`.

②b Reasoning about the second branch expression is more involved: the variable x must bind the argument of `Just`, but we have not yet ensured that this branch will be taken! To bridge the gap, we bind x to the symbolic, and indeterminate, *inverse constructor application*

`Just` $^{-1}$ $(\![-]\,??_5)$ when evaluating the branch expression; unevaluation "transfers" the resulting example from the symbolic result to the scrutinee:

$$\texttt{x} \;\Rightarrow\; \texttt{Just}^{-1}\ (\![-]\,??_5) \;\Leftarrow\; \texttt{1} \;\dashv\; (- \vdash \bullet_5 \models \texttt{Just 1})$$

This constraint ensures that the `case` in `unJust` will resolve to the second branch (`Just x`) and that its expression will produce `1`, and thus that the `case` in `odd` will resolve to the second branch (`S Z`) and produce `True`, as asserted.

(3) By recursively unevaluating the third branch, `odd n''`, case unevaluation can derive additional solutions: `Just 3`, `Just 5`, etc. Naïvely unevaluating all branches, however, would introduce a significant degree of non-determinism—even non-termination. Therefore, our formulation and implementation impose simple restrictions—described in §3 and §5—on case unevaluation to trade expressiveness for performance.

Altogether, live bidirectional evaluation untangles the interplay between indeterminate branching and assertions so that Smyth can, for instance, fill the holes in `minus` and `mult` in Figure 2.

3 LIVE BIDIRECTIONAL EVALUATION

In this section, we formally define *live evaluation* $E\,;\,F \vdash e \Rightarrow r$ and *live unevaluation* $F \vdash r \Leftarrow ex \dashv K$ for a calculus called Core Smyth. We choose a natural semantics (big-step, environment-style) presentation [Kahn 1987], though our techniques can be re-formulated for a small-step, substitution-style model. Compared to our earlier notation, here we refer to environments E and F—often typeset in light gray, because environments would "fade away" in a substitution-style presentation.

Our formulation proceeds as follows. First, in §3.1 and §3.2, we define the syntax and type checking judgements of Core Smyth. Next, in §3.3, we present live evaluation, which adapts the *live programming with holes* technique [Omar et al. 2019] to our setting; minor differences are described in §7.1. Lastly, we define example satisfaction in §3.4 and live unevaluation in §3.5. In §4, we build a synthesis pipeline around the combination of live evaluation and unevaluation.

3.1 Syntax

Figure 3 defines the syntax of Core Smyth, a calculus of recursive functions, unit, pairs, and (named, recursive) algebraic datatypes. We say "products" to mean unit and pairs.

Datatypes. We assume a fixed datatype context Σ. A datatype D has some number n of constructors C_i, each of which carries a single argument of type T_i—the type of C_i is $T_i \to D$.

Expressions and Holes. The expression forms on the first three lines are standard function, product, and constructor forms, respectively. The expressions $\texttt{prj}_1\ e$ and $\texttt{prj}_2\ e$ project the first and second components of a pair. Each `case` expression has one branch for each of the n constructors C_i corresponding to the type of the scrutinee e; for simplicity, nested patterns are not supported.

Holes $??_h$ can appear anywhere in expressions (i.e. expressions are sketches). We assume each hole in a sketch has a unique name h, but we sometimes write `??` when the name is not referred to. Hole contexts Δ define a *contextual type* $(\Gamma \vdash \bullet : T)$ to describe the type and the type context that is available to expressions that can "fill" a given hole [Nanevski et al. 2008; Omar et al. 2019].

Results. We define a separate grammar of *results* r—with evaluation environments E that map variables to results—to support the definition of big-step, environment-style evaluation $E \vdash e \Rightarrow r$ below. Because of holes, results are not conventional values. Terminating evaluations produce two kinds of *final* results; neither kind of result is stuck (i.e. erroneous).

The four result forms on the first line of the result grammar would—on their own—correspond to values in a conventional natural semantics (without holes). In Core Smyth, these *determinate* results

can be eliminated in a type-appropriate position; the appendix (§A.1) defines a simple predicate r `det` to identify such results, and type checking is discussed below. Note that a recursive function closure $[E]\,\texttt{fix}\ f\ (\lambda x.e)$ stores an environment E that binds the free variables of the function body e, except the name f of the function itself. We sometimes write $\lambda x.e$ for non-recursive functions.

The four *indeterminate* result forms on the second line of the grammar are unique to the presence of holes. Rather than aborting evaluation with an error when a hole reaches elimination position (e.g., `raise "Hole"`), an indeterminate result r (defined by the predicate r `indet` (§A.1)) serves as a placeholder for where to continue evaluation if and when the hole is later filled (either by the programmer or synthesis engine) with a well-typed expression. The primordial indeterminate result is a *hole closure* $[E]\,??_h$—the environment binds the free variables that a hole-filling expression may refer to. An indeterminate application $r_1\ r_2$ appears when the function has not yet evaluated to a function closure (i.e. r_1 `indet`); we require that r_2 be final in accordance with our eager evaluation semantics, discussed below. An indeterminate projection $\texttt{prj}_{\,i\in[2]}\ r$ appears when the argument has not yet evaluated to a pair (i.e. r `indet`). An indeterminate case closure $[E]\,\texttt{case}\ r\ \texttt{of}\ \{C_i\ x_i \to e_i\}^{\,i\in[n]}$ appears when the scrutinee has not yet evaluated to a constructor application (i.e. r `indet`)—like with function and hole closures, the environment E is used when evaluation resumes with the appropriate branch. Because they record how "paused" expressions should "resume," we sometimes refer to indeterminate results as "partially evaluated expressions."

The *inverse constructor application* form $C^{-1}\ r$ on the third line of the result grammar is internal to live unevaluation and is discussed in §3.5.

$$
\begin{array}{rrcl}
\textbf{Types} & T & ::= & T_1 \to T_2 \mid () \mid (T_1, T_2) \mid D \qquad\qquad \textbf{Datatypes}\ D \\
\textbf{Expressions} & e & ::= & \texttt{fix}\ f\ (\lambda x.e) \mid e_1\ e_2 \mid x \qquad\qquad \textbf{Variables}\ f, x \\
 & & \mid & () \mid (e_1, e_2) \mid \texttt{prj}_{\,i\in[2]}\ e \\
 & & \mid & C\ e \mid \texttt{case}\ e\ \texttt{of}\ \{C_i\ x_i \to e_i\}^{\,i\in[n]} \qquad\qquad \textbf{Constructors}\ C \\
 & & \mid & ??_h \qquad\qquad \textbf{Hole Names}\ h \\
\textbf{Results} & r & ::= & [E]\,\texttt{fix}\ f\ (\lambda x.e) \mid () \mid (r_1, r_2) \mid C\ r \\
 & & \mid & [E]\,??_h \mid r_1\ r_2 \mid \texttt{prj}_{\,i\in[2]}\ r \mid [E]\,\texttt{case}\ r\ \texttt{of}\ \{C_i\ x_i \to e_i\}^{\,i\in[n]} \\
 & & \mid & C^{-1}\ r \\
\textbf{Environments} & E & ::= & - \mid E, x \mapsto r \\
\textbf{Hole Fillings} & F & ::= & - \mid F, h \mapsto e \\
\textbf{Type Contexts} & \Gamma & ::= & - \mid \Gamma, x{:}T \\
\textbf{Datatype Contexts} & \Sigma & ::= & - \mid \Sigma, \texttt{type}\ D = \{C_i\ T_i\}^{\,i\in[n]} \\
\textbf{Hole Type Contexts} & \Delta & ::= & - \mid \Delta, h \mapsto (\Gamma \vdash \bullet : T) \\
\textbf{Synthesis Goals} & G & ::= & - \mid G, (\Gamma \vdash \bullet_h : T \models X) \\
\textbf{Example Constraints} & X & ::= & - \mid X, (E \vdash \bullet \models ex) \\
\textbf{Simple Values} & v & ::= & () \mid (v_1, v_2) \mid C\ v \\
\textbf{Examples} & ex & ::= & () \mid (ex_1, ex_2) \mid C\ ex \mid \{v \to ex\} \mid \top \\
\textbf{Unevaluation Constraints} & K & ::= & (U\,;F) \\
\textbf{Unfilled Holes} & U & ::= & - \mid U, h \mapsto X
\end{array}
$$

Fig. 3. Syntax of Core Smyth.

Examples. A *synthesis goal* $(\Gamma \vdash \bullet_h : T \models X)$ describes a hole $??_h$ to be filled according to the contextual type $(\Gamma \vdash \bullet : T)$ and *example constraints* X. Each example constraint $(E \vdash \bullet \models ex)$ requires that an expression to fill the hole must, in the environment E, satisfy example ex.

Examples include *simple values* v, which are first-order product values or constructor applications; *input-output* examples $\{v \to ex\}$, which constrain function-typed holes; and *top* $\top$, which imposes no constraints. We sometimes refer to example constraints simply as "examples" when the meaning is clear from context. The coercion $\lfloor v \rfloor$ "upcasts" a simple value to a result. The coercion $\lceil r \rceil = v$ "downcasts" a result to a simple value, if possible.

Examples are essentially the same as described by Osera and Zdancewic [2015]. Smyth additionally includes top examples. For simplicity Core Smyth includes only first-order function examples, though our implementation (§5) supports higher-order function examples like Myth.

3.2 Type Checking

Type checking $\Sigma\,;\Delta\,;\Gamma \vdash e : T$ (Figure 4) takes a hole type context Δ as input, used by the T-Hole rule to decide valid typings for a hole $??_h$. The remaining rules are standard (§A.2).

3.3 Live Evaluation

Figure 4 defines *live evaluation* $E\,;F \vdash e \Rightarrow r$, which first uses *expression evaluation* $E \vdash e \Rightarrow r$ to produce a final result r, and then *resumes* evaluation $F \vdash r \Rightarrow r'$ of the result r in positions that were paused because of holes now filled by F.

Expression Evaluation. Compared to a conventional natural semantics, there are four new rules—E-Hole, E-App-Indet, E-Prj-Indet, and E-Case-Indet—one for each indeterminate result form. The E-Hole rule creates a hole closure $[E]\,??_h$ that captures the evaluation environment.

The other three rules, suffixed "-Indet," are counterparts to rules E-App, E-Prj, and E-Case for determinate forms. For example, when a function evaluates to a result r_1 that is not a function closure, the E-App-Indet rule creates the indeterminate application result $r_1\ \ r_2$. The remaining

Type Checking (excerpt from §A.2) **and Live Eval.** $\boxed{\Sigma\,;\Delta\,;\Gamma \vdash e : T}$ $\boxed{E\,;F \vdash e \Rightarrow r}$

$$\text{[T-Hole]}\ \frac{\Delta(??_h) = (\Gamma \vdash \bullet : T)}{\Sigma\,;\Delta\,;\Gamma \vdash ??_h : T} \qquad \frac{E \vdash e \Rightarrow r \quad F \vdash r \Rightarrow r'}{E\,;F \vdash e \Rightarrow r'}$$

Expression Evaluation (excerpt from §A.3) $\boxed{E \vdash e \Rightarrow r}$

$$\text{[E-Hole]}\ \frac{}{E \vdash ??_h \Rightarrow [E]\,??_h} \qquad \text{[E-App]}\ \frac{E \vdash e_1 \Rightarrow r_1 \quad E \vdash e_2 \Rightarrow r_2 \quad r_1 = [E_f]\,\texttt{fix}\ f\ (\lambda x.e_f) \quad E_f,\ f \mapsto r_1,\ x \mapsto r_2 \vdash e_f \Rightarrow r}{E \vdash e_1\ \ e_2 \Rightarrow r}$$

$$\text{[E-App-Indet]}\ \frac{E \vdash e_1 \Rightarrow r_1 \quad E \vdash e_2 \Rightarrow r_2 \quad r_1 \neq [E_f]\,\texttt{fix}\ f\ (\lambda x.e_f)}{E \vdash e_1\ \ e_2 \Rightarrow r_1\ \ r_2}$$

Resumption (excerpt from §A.4) $\boxed{F \vdash r \Rightarrow r'}$

$$\text{[R-Hole-Resume]}\ \frac{F(h) = e_h \quad E \vdash e_h \Rightarrow r \quad F \vdash r \Rightarrow r'}{F \vdash [E]\,??_h \Rightarrow r'} \qquad \text{[R-Hole-Indet]}\ \frac{h \notin dom(F) \quad F \vdash E \Rightarrow E'}{F \vdash [E]\,??_h \Rightarrow [E']\,??_h}$$

Fig. 4. Type Checking, Evaluation, and Resumption.

rules are similar (§A.3). Evaluation is deterministic and produces final results; the appendix (§A.3) formally establishes these propositions, as well as a suitable notion of type safety.

Resumption. Result resumption resembles expression evaluation. For closures $[E]\,??_h$ over holes that F fill with an expression e_h, R-Hole-Resume evaluates e_h in the closure environment, producing a result r. Because e_h may refer to other holes now filled by F, r is recursively resumed to r'.

3.4 Example Satisfaction

Live evaluation partially evaluates a sketch to a result, and Figure 5 defines what it means for a result to satisfy an example. To decide whether expression e satisfies example constraint ($E \vdash \bullet \models ex$), the Sat rule evaluates the expression to a result r and then checks whether r satisfies ex. The XS-Top rule accepts all results. The remaining rules break down input-output examples (XS-Input-Output) into equality checks for products and constructors (XS-Unit, XS-Pair, and XS-Ctor).

Hole closures may appear in a satisfying result, but they may *not* be directly checked against product, constructor, or input-output examples. The purpose of *live unevaluation* is to provide a notion of example *consistency* to accompany this "ground-truth" notion of example satisfaction.

3.5 Live Unevaluation

Figure 6 defines *live unevaluation* $F \vdash r \Leftarrow ex \dashv K$, which produces constraints K over holes that are sufficient to ensure example satisfaction $F \vdash r \models ex$. The *live bidirectional example checking* judgement $F \vdash e \rightleftharpoons X \dashv K$ lifts this notion to example constraints: Live-Check appeals to evaluation followed by unevaluation to check each constraint in X.

Theorem (Soundness of Live Unevaluation).
If $F \vdash r \Leftarrow ex \dashv K$ and $F \oplus F' \models K$ and $F \oplus F' \vdash r \Rightarrow r'$, then $F \oplus F' \vdash r' \models ex$.

Theorem (Soundness of Live Bidirectional Example Checking).
If $F \vdash e \rightleftharpoons X \dashv K$ and $F \oplus F' \models K$, then $F \oplus F' \vdash e \models X$.

Example Constraint Satisfaction $\boxed{F \vdash e \models X}$

$$\frac{\{\, E_i ; F \vdash e \Rightarrow r_i \quad F \vdash r_i \models ex_i \,\}^{\,i \in [n]}}{F \vdash e \models \{\, (E_i \vdash \bullet \models ex_i) \,\}^{\,i \in [n]}}\ [\textsc{Sat}]$$

Example Satisfaction $\boxed{F \vdash r \models ex}$

$$\frac{}{F \vdash r \models \top}\ [\textsc{XS-Top}] \qquad \frac{}{F \vdash () \models ()}\ [\textsc{XS-Unit}] \qquad \frac{\{\, F \vdash r_i \models ex_i \,\}^{\,i \in [2]}}{F \vdash (r_1, r_2) \models (ex_1, ex_2)}\ [\textsc{XS-Pair}]$$

$$\frac{F \vdash r \models ex}{F \vdash C\ r \models C\ ex}\ [\textsc{XS-Ctor}] \qquad \frac{F \vdash r_1\ \lfloor v_2 \rfloor \Rightarrow r \quad F \vdash r \models ex}{F \vdash r_1 \models \{v_2 \to ex\}}\ [\textsc{XS-Input-Output}]$$

Unevaluation Constraint Satisfaction $\boxed{F \models K}$

$$\frac{F \supseteq F_0 \quad \{\, F \vdash ??_{h_i} \models X_i \,\}^{\,i \in [n]}}{F \models ((h_1 \mapsto X_1, \ldots, h_n \mapsto X_n) ; F_0)}$$

Fig. 5. Example and Constraint Satisfaction.

Unevaluation Constraints. Two kinds of constraints K are generated by unevaluation (cf. Figure 3). The first is a context U of bindings $h \mapsto X$ that maps unfilled holes $??_h$ to sets X of example constraints ($E \vdash \bullet \models ex$). The second is a hole-filling F which, as discussed below, is used to optimize unevaluation of case expressions. The former are "hole example contexts," analogous to hole type contexts Δ; the metavariable U serves as a mnemonic for holes left unfilled by a hole-filling F. (In the simpler presentation of §2, only example constraints were generated, and each was annotated with a hole name.)

To define what it means for a filling F to constitute a valid solution for a set of constraints $K = (U\,; F_0)$, Figure 5 defines constraint satisfaction $F \models K$ by checking that (i) F subsumes any fillings F_0 in K and (ii) F satisfies the examples X_i for each hole $??_{h_i}$ constrained by K.

When analyzing multiple subexpressions, several unevaluation rules—discussed below—generate multiple sets of constraints that must be combined. Figure 6 shows the signature of two constraint

Unevaluation Constraint Merging (in §A.5) $\boxed{K_1 \oplus K_2 = K}$ $\boxed{\Sigma\,;\Delta\,;\mathit{Merge}(K) \triangleright K'}$

Live Bidirectional Example Checking $\boxed{\Sigma\,;\Delta\,;F \vdash e \rightleftharpoons X \dashv K}$

[Live-Check]
$$\frac{\{\, E_i\,; F \vdash e \Rightarrow r_i \quad F \vdash r_i \Leftarrow ex_i \dashv K_i \,\}^{\,i\in[n]}}{F \vdash e \rightleftharpoons (E_1 \vdash \bullet \models ex_1),\ \ldots,\ (E_n \vdash \bullet \models ex_n) \dashv K_1 \oplus \cdots \oplus K_n}$$

Live Unevaluation $\boxed{\Sigma\,;\Delta\,;F \vdash r \Leftarrow ex \dashv K}$

[U-Top] $\dfrac{}{F \vdash r \Leftarrow \top \dashv -}$ [U-Unit] $\dfrac{}{F \vdash () \Leftarrow () \dashv -}$

[U-Pair] $\dfrac{F \vdash r_1 \Leftarrow ex_1 \dashv K_1 \quad F \vdash r_2 \Leftarrow ex_2 \dashv K_2}{F \vdash (r_1\,, r_2) \Leftarrow (ex_1\,, ex_2) \dashv K_1 \oplus K_2}$ [U-Ctor] $\dfrac{F \vdash r \Leftarrow ex \dashv K}{F \vdash C\ r \Leftarrow C\ ex \dashv K}$

[U-Fix] $\dfrac{F \vdash e \rightleftharpoons (E,\ f \mapsto [E]\,\mathtt{fix}\ f\ (\lambda x.e),\ x \mapsto \lfloor v \rfloor \vdash \bullet \models ex) \dashv K}{F \vdash [E]\,\mathtt{fix}\ f\ (\lambda x.e) \Leftarrow \{v \to ex\} \dashv K}$ [U-Hole] $\dfrac{U = h \mapsto (E \vdash \bullet \models ex)}{F \vdash [E]\,??_h \Leftarrow ex \dashv (U\,; -)}$

[U-App] $\dfrac{\lceil r_2 \rceil = v_2 \quad F \vdash r_1 \Leftarrow \{v_2 \to ex\} \dashv K}{F \vdash r_1\ r_2 \Leftarrow ex \dashv K}$ [U-Prj-1] $\dfrac{F \vdash r \Leftarrow (ex\,, \top) \dashv K}{F \vdash \mathtt{prj}_1\ r \Leftarrow ex \dashv K}$ [U-Prj-2] $\dfrac{F \vdash r \Leftarrow (\top\,, ex) \dashv K}{F \vdash \mathtt{prj}_2\ r \Leftarrow ex \dashv K}$

[U-Case]
$$\frac{\begin{array}{c} j \in [1,n] \quad F \vdash r \Leftarrow C_j\ \top \dashv K_1 \\ F \vdash e_j \rightleftharpoons (E,\ x_j \mapsto {C_j}^{-1}\ r \vdash \bullet \models ex) \dashv K_2 \end{array}}{F \vdash [E]\,\mathtt{case}\ r\ \mathtt{of}\ \{C_i\ x_i \to e_i\}^{\,i\in[n]} \Leftarrow ex \dashv K_1 \oplus K_2}$$

[U-Inverse-Ctor] $\dfrac{F \vdash r \Leftarrow C\ ex \dashv K}{F \vdash C^{-1}\ r \Leftarrow ex \dashv K}$

[U-Case-Guess]
$$\frac{\begin{array}{c} j \in [1,n] \quad F' = \mathit{Guesses}(\Delta, \Sigma, r) \quad F \oplus F' \vdash r \Rightarrow C_j\ r' \\ F \oplus F' \vdash e_j \rightleftharpoons (E,\ x_j \mapsto r' \vdash \bullet \models ex) \dashv K \end{array}}{F \vdash [E]\,\mathtt{case}\ r\ \mathtt{of}\ \{C_i\ x_i \to e_i\}^{\,i\in[n]} \Leftarrow ex \dashv (-\,; F') \oplus K}$$

Fig. 6. Live Bidirectional Example Checking via Live Unevaluation.

merge operators. The "syntactic" merge operation $K_1 \oplus K_2$ pairwise combines example contexts U and fillings F in a straightforward way. Syntactically merged constraints may describe holes $??_h$ both with example constraints X in U and fillings in F; the "semantic" operation $\mathit{Merge}(K)$ uses live bidirectional example checking to check consistency in such situations. The full definitions can be found in the appendix (§A.5).

Simple Unevaluation Rules. Analogous to the five example satisfaction rules (prefixed "XS-" in Figure 5) are the U-Top rule to unevaluate any result with $\top$ and the U-Unit, U-Pair, U-Ctor, and U-Fix rules to unevaluate determinate results. The base case in which unevaluation generates example constraints is for hole closures $[E]\,??_h$—the U-Hole rule generates the (named) example constraint $h \mapsto (E \vdash \bullet \models ex)$.

The U-Fix rule refers to bidirectional example checking—evaluation followed by unevaluation—to "test" that a function is consistent with an input-output example. For instance, to unevaluate the function closure $[\texttt{zero} \mapsto \texttt{0}]\,\lambda x.\,??_h$ with $\{\texttt{1} \to \texttt{2}\}$, first, the function application is evaluated: the closure environment is extended to bind the input example $x \mapsto \lfloor \texttt{1} \rfloor$, and the function body is evaluated to result $[\texttt{zero} \mapsto \texttt{0},\ x \mapsto \lfloor \texttt{1} \rfloor]\,??_h$. Second, the output example $\texttt{2}$ is unevaluated to this result, for which U-Hole generates the constraint $h \mapsto (\texttt{zero} \mapsto \texttt{0},\ x \mapsto \lfloor \texttt{1} \rfloor \vdash \bullet \models \texttt{2})$. (Valid fillings for $??_h$ include $\texttt{S}\ (\texttt{S}\ \texttt{Z})$, $\texttt{S}\ \texttt{x}$, and $\texttt{S}\ (\texttt{S}\ \texttt{zero})$.)

The remaining rules, discussed below, transform "indirect" unevaluation goals for more complex indeterminate results into "direct" examples on holes.

Indeterminate Function Applications. Consider an indeterminate function application $r_1\ r_2$, with the goal to satisfy ex. For results r_2 that are simple (first-order) values v_2, the U-App rule unevaluates the indeterminate function r_1 with the input-output example $\{v_2 \to ex\}$.

In general, the argument r_2 may include holes that would later appear in elimination position when r_1 is filled and the application resumes. For results r_2 that are not simple values, it is not possible to generate sufficient constraints locally to ensure that $r_1\ r_2$ satisfies ex. For instance, if r_2 is of the form $[E]\,??_h$, the hypothetical constraint "$\{([E]\,??_h) \to ex\}$" would not provide any information about which input values the function r_1 must map to results that satisfy ex. As such, there is no unevaluation rule for arbitrary indeterminate application forms.

Indeterminate Projections. The U-Prj-1 and U-Prj-2 rules use $\top$ for the component to be left unconstrained. For example, unevaluating $\texttt{prj}_1\ [E]\,??_h$ with $\texttt{1}$ generates $h \mapsto (E \vdash \bullet \models (\texttt{1}, \top))$.

Indeterminate Case Expressions. Recall from §2.3 the goal to unevaluate an indeterminate case expression with the number $\texttt{1}$: $\texttt{case}\ [-]\,??_h\ \texttt{of}\ \{\texttt{Nothing}\ \_ \to \texttt{0};\ \texttt{Just}\ x \to x\} \Leftarrow \texttt{1}$. Intuitively, this should require $h \mapsto (- \vdash \bullet \models \texttt{Just 1})$.

To compute this constraint, the U-Case rule considers each branch j. The first premise unevaluates the scrutinee r with $C_j\ \top$ to the scrutinee r, generating constraints K_1 required for r to produce an application of constructor C_j. If successful, the next step is to evaluate the corresponding branch expression e_j and check that it is consistent with the goal ex. However, the argument to the constructor will only be available after all constraints are solved and evaluation resumes.

We introduce the *inverse constructor application* ${C_j}^{-1}\ r$ (Figure 3) to bridge this gap between constraint generation and constraint solving. To proceed down the branch expression, we bind the pattern variable x_j to ${C_j}^{-1}\ r$. Locally, this allows the third premise of U-Case to check whether the branch expression e_j satisfies ex. For the example above, the result of evaluating the second branch expression, x, is $\texttt{Just}^{-1}\ ([-]\,??_h)$. Unevaluating $\texttt{Just}^{-1}\ ([-]\,??_h)$ with $\texttt{1}$ generates the constraint $h \mapsto (- \vdash \bullet \models \texttt{Just}^{-1}\ \texttt{1})$. Finally, the U-Inverse-Ctor rule transfers the example from the inverse constructor application to a constructor application, producing $h \mapsto (- \vdash \bullet \models \texttt{Just 1})$.

Indeterminate Case Expressions: Guessing Scrutinees. The interplay between U-Case and U-Inverse-Ctor allows unevaluation to resolve branching decisions by generating constraints without the obligation to synthesize expressions that satisfy them. A downside of this "lazy" approach is the significant degree of non-determinism; indeed, many of the generated sets of constraints may be unsatisfiable.

As a more efficient approach in situations where the full expressiveness of U-Case is not needed, the U-Case-Guess rule "eagerly" resolves the direction of the branch by guessing a hole-filling F' via a non-deterministic uninterpreted function $\mathit{Guesses}(\Delta, \Sigma, r)$, and checking whether this filling resumes the scrutinee r to an application of a constructor C_j, where C_j is one of the n data constructors for the datatype D of the scrutinee. If so, the direction of the branch has been determined, so the last step is to unevaluate the jth branch expression e_j with the goal example ex, in an appropriately extended environment.

For instance, consider again the goal $\texttt{case}\ [E]\,??_h\ \texttt{of}\ \{\texttt{Nothing}\ \_ \to \texttt{0}\,;\ \texttt{Just}\ x \to x\} \Leftarrow \texttt{1}$ but here with the environment $E = \texttt{nothing} \mapsto \texttt{Nothing}, \texttt{just0} \mapsto \texttt{Just 0}, \texttt{just1} \mapsto \texttt{Just 1}$. The *Guesses* function might choose the filling $F' = h \mapsto \texttt{just1}$, which resumes the scrutinee $[E]\,??_h$ to `Just 1`. In the environment extended with $x \mapsto \texttt{1}$, the corresponding branch expression x evaluates to the result `1`. Unevaluating this result with the example `1` succeeds via U-Ctor and U-Unit without generating additional constraints. (If guessing fills $??_h$ with `nothing` or `just0`, the result, `0`, of the branch expression would fail to unevaluate to `1`.)

Whereas the U-Hole rule is the source of example constraints U produced by unevaluation, the U-Case-Guess rule is the source of hole-filling constraints F. We describe our concrete implementation of *Guesses* in §5.

4 SYNTHESIS PIPELINE

Live bidirectional evaluation addresses the challenge of checking example satisfaction for programs with holes. In this section, we define a synthesis pipeline that uses live bidirectional evaluation to (1) derive example constraints from `assert`s and (2) solve the resulting constraints.

$$\overbrace{\underbrace{p \Rightarrow r\,;A}_{\text{Constraint Collection (§4.1)}} \quad \mathit{Simplify}(A) \triangleright K}^{\text{Constraint Solving (§4.2)}} \qquad \overbrace{\mathit{Solve}(K) \rightsquigarrow F}$$

Overview Program: Plus. Before describing each of these components formally, we summarize how they will fit together to synthesize the `plus` function in §2.1:

```
let plus = ??0 in assert ([plus 0 1, plus 2 0, plus 1 2] == [1, 2, 3])
```

First, when evaluating the program, the left-hand side of the `assert` produces three nested, indeterminate function calls: `[([−]??0 0) 1, ([−]??0 2) 0, ([−]??0 1) 2]`. Structurally comparing this list of indeterminate results with the list of values `[1, 2, 3]` yields three *assertion* predicates A as a side-effect (via rules Eval-and-Assert, RC-Ctor, and RC-Assert-1, discussed below):

$$A = (([-]\,??_0\ \texttt{0})\ \texttt{1}) \Rightarrow \texttt{1},\ (([-]\,??_0\ \texttt{2})\ \texttt{0}) \Rightarrow \texttt{2},\ (([-]\,??_0\ \texttt{1})\ \texttt{2}) \Rightarrow \texttt{3}$$

Second, we use live bidirectional example checking (Live-Check) to convert—i.e. *Simplify*—the assertions A into example constraints U (via U-App and U-Hole):

$$U = \texttt{0} \mapsto ((- \vdash \bullet \models \{\texttt{0} \to \{\texttt{1} \to \texttt{1}\}\}),\ (- \vdash \bullet \models \{\texttt{2} \to \{\texttt{0} \to \texttt{2}\}\}),\ (- \vdash \bullet \models \{\texttt{1} \to \{\texttt{2} \to \texttt{3}\}\}))$$

The simplified constraints $K = (U\,; -)$ contain an empty hole-filling because U-Case-Guess is not invoked to resolve any indeterminate case expressions.

Finally, the holes in U are solved one at a time; here there is only $??_0$. Solving one hole may generate new subgoals (Refine and Branch) or new constraints on existing goals (Guess-and-Check).

The search path sketched in § 2.1 produces the solution F below that solves the constraints $K = (U\,;-)$. Each step is annotated with the rules used to conclude the subderivation.

0	↦	`fix` f_1 `(λm.fix` f_2 `(λn.??`$_1$`))`	Solve-One, Refine, Refine-Fix (twice)
1	↦	`case m {Z →` $??_2$ `; S m' →` $??_3$ `}`	Solve-One, Branch, Branch-Case
3	↦	`S` $??_4$	Solve-One, Refine, Refine-Ctor
4	↦	`plus m' n`	Solve-One, Guess-and-Check, Live-Check
2	↦	`n`	Solve-One, Guess-and-Check, Live-Check

4.1 Constraint Collection

Figure 7 defines a *program* to be an expression followed by an `assert` $(e_1 = e_2)$ statement. Changes to allow `assert`s in arbitrary expressions are discussed in §7.

Assertions via Result Consistency. A typical semantics for `assert` would require the expression results r_1 and r_2 to be equal, otherwise raising an exception. Instead, rather than equality, the Eval-and-Assert rule in Figure 7 checks *result consistency*, $r_1 \equiv_A r_2$, a notion of equality modulo assumptions A about indeterminate results. Determinate results are consistent if structurally equal, as checked by the RC-Refl, RC-Pair, and RC-Ctor rules. Indeterminate results r are consistent with simple values v—the RC-Assert-1 and RC-Assert-2 rules generate *assertion* predicates $r \Rightarrow v$ in such cases. Figure 7 also defines assertion satisfaction $F \models A$: for each assertion $r_i \Rightarrow v_i$ in A, the indeterminate result r_i should resume under filling F and produce the value v_i.

Assertion Simplification. For each assertion $r_i \Rightarrow v_i$, the *Simplify* procedure in Figure 7 converts the simple value into an example $\lfloor v_i \rfloor$ and unevaluates it to r_i to generate example constraints.

Theorem (Soundness of Assertion Simplification).
If $Simplify(A) \triangleright K$ *and* $F \models K$, *then* $F \models A$.

Program Evaluation $\boxed{p \Rightarrow r\,;A}$

$$\begin{array}{rrcl} \underline{\textbf{Programs}} & p & ::= & \texttt{let main} = e \texttt{ in assert } (e_1 = e_2) \\ \underline{\textbf{Assertions}} & A & ::= & \{\, r_i \Rightarrow v_i \,\}^{\,i\in[n]} \end{array}$$

[Eval-and-Assert]
$$\frac{- \vdash e \Rightarrow r \qquad \{\, \texttt{main} \mapsto r \vdash e_i \Rightarrow r_i \,\}^{\,i\in[2]} \qquad r_1 \equiv_A r_2}{\texttt{let main} = e \texttt{ in assert } (e_1 = e_2) \Rightarrow r\,;A}$$

Result Consistency $\boxed{r \equiv_A r'}$

$$\frac{}{r \equiv_- r}\ [\text{RC-Refl}] \qquad \frac{r_1 \equiv_{A_1} r_1' \qquad r_2 \equiv_{A_2} r_2'}{(r_1\,,\,r_2) \equiv_{A_1 + A_2} (r_1'\,,\,r_2')}\ [\text{RC-Pair}] \qquad \frac{r \equiv_A r'}{C\ r \equiv_A C\ r'}\ [\text{RC-Ctor}]$$

$$\frac{\lceil r_2 \rceil = v_2 \qquad A = r_1 \Rightarrow v_2}{r_1 \equiv_A r_2}\ [\text{RC-Assert-1}] \qquad \frac{\lceil r_1 \rceil = v_1 \qquad A = r_2 \Rightarrow v_1}{r_1 \equiv_A r_2}\ [\text{RC-Assert-2}]$$

Assertion Satisfaction and Simplification $\boxed{F \models A}$ $\boxed{Simplify(A) \triangleright K}$

$$\frac{\{\, F \vdash r_i \Rightarrow r_i' \qquad \lceil r_i' \rceil = v_i \,\}^{\,i\in[n]}}{F \models \{\, r_i \Rightarrow v_i \,\}^{\,i\in[n]}} \qquad \frac{\{\, r_i\ \texttt{final} \qquad - \vdash r_i \Leftarrow \lfloor v_i \rfloor \dashv K_i \,\}^{\,i\in[n]}}{Simplify(\{\, r_i \Rightarrow v_i \,\}^{\,i\in[n]}) \triangleright K_1 \oplus \cdots \oplus K_n}$$

Fig. 7. Constraint Collection.

4.2 Constraint Solving

The constraints K, of the form $(U\,;F_0)$, include filled holes F_0 from constraint simplification (cf. U-Case-Guess) and a set U of unfilled holes constrained by examples. Figure 8 and Figure 9 define an algorithm to synthesize expressions for unfilled holes, generalizing Myth to use live bidirectional evaluation and to fill interdependent holes.

The $Solve(U\,;F)$ procedure in Figure 8 is the entry point for filling the holes in U. The Solve-Done rule handles the terminal case, when no unfilled holes remain. Otherwise, the Solve-One rule chooses an unfilled hole $??_h$ and forms the synthesis goal $(\Gamma \vdash \bullet_h : T \models X)$ from the hole type and example contexts Δ and U. The *hole synthesis* procedure—discussed next—completes the task, which, in Smyth, may assume constraints K over other holes. Any such constraints K are combined with the existing ones using the semantic *Merge* operation (cf. §3.5), and the resulting constraints K' are recursively solved.

Hole Synthesis. For each unfilled hole, the hole synthesis procedure $F\,;(\Gamma \vdash \bullet_h : T \models X) \rightsquigarrow_{\text{fill}} K\,;\Delta'$ augments guessing-and-checking (Guess-and-Check) with example-directed refinement (Refine) and branching (Branch); these rules are discussed in turn below.

The structure of hole synthesis in Core Smyth closely follows Myth [Osera and Zdancewic 2015], which presents a novel approach to synthesis by analogy to proof search for *bidirectional type checking* [Pierce and Turner 2000]. We refer the reader to their paper for a comprehensive account of their ideas; we limit our discussion to the most important technical differences.

Constraint Solving $\boxed{\Sigma\,;\Delta\,;Solve(K) \rightsquigarrow F\,;\Delta'}$

[Solve-Done]

$$\frac{}{\Sigma\,;\Delta\,;Solve(-\,;F) \rightsquigarrow F\,;\Delta}$$

[Solve-One]

$$\frac{h \in dom(U) \quad \Delta(h) = (\Gamma \vdash \bullet : T) \quad U(h) = X \quad F\,;(\Gamma \vdash \bullet_h : T \models X) \rightsquigarrow_{\text{fill}} K\,;\Delta' \quad \Sigma\,;\Delta +\!\!+ \Delta'\,;Merge((U \setminus h\,;F) \oplus K) \rhd K' \quad \Sigma\,;\Delta +\!\!+ \Delta'\,;Solve(K') \rightsquigarrow F'\,;\Delta''}{\Sigma\,;\Delta\,;Solve(U\,;F) \rightsquigarrow F'\,;\Delta''}$$

Type-and-Example-Directed Hole Synthesis $\boxed{\Sigma\,;\Delta\,;F\,;(\Gamma \vdash \bullet_h : T \models X) \rightsquigarrow_{\text{fill}} K\,;\Delta'}$

[Guess-and-Check]

$$\frac{(\Gamma \vdash \bullet : T) \rightsquigarrow_{\text{guess}} e \quad (F,\, h \mapsto e) \vdash e \rightleftharpoons X \dashv K}{F\,;(\Gamma \vdash \bullet_h : T \models X) \rightsquigarrow_{\text{fill}} (-\,;h \mapsto e) \oplus K\,;-}$$

[Defer]

$$\frac{X = (E_1 \vdash \bullet \models \top), \ldots, (E_n \vdash \bullet \models \top) \quad n > 0}{F\,;(\Gamma \vdash \bullet_h : T \models X) \rightsquigarrow_{\text{fill}} (-\,;h \mapsto ??_h)\,;-}$$

[Refine, Branch]

$$\frac{F\,;(\Gamma \vdash \bullet : X \models T) \rightsquigarrow_{\{\text{refine},\text{branch}\}} e \dashv \{\,(\Gamma_i \vdash \bullet_{h_i} : T_i \models X_i)\,\}^{\,i\in[n]}\,;K}{F\,;(\Gamma \vdash \bullet_h : T \models X) \rightsquigarrow_{\text{fill}} ((h_1 \mapsto X_1, \ldots, h_n \mapsto X_n)\,;h \mapsto e) \oplus K\,;\{\,h_i \mapsto (\Gamma_i \vdash \bullet : T_i)\,\}^{\,i\in[n]}}$$

Fig. 8. Constraint Solving with Guessing, Refinement, and Branching. *We at once define Refine and Branch by differentiating the two by color; the signature of the branching judgement extends that of the refinement judgement with an additional input F and an additional output K.*

Besides modifications to notation and organization, the primary differences of our formulation are that hole synthesis: (i) refers to the filling F from previous synthesis tasks completed by *Solve*; (ii) may generate example constraints over other holes in the program; (iii) may fill other holes in the program besides the goal $??_h$; and (iv) includes a rule, Defer, to "fill" the hole with $??_h$ when all examples are top—these constraints are not imposed directly from program assertions, but are created internally by unevaluation.

Type-Directed Guessing (in §A.6)

$$\Sigma\,;\,(\Gamma \vdash \bullet : T) \rightsquigarrow_{\text{guess}} e$$

Type-and-Example-Directed Refinement

$$\Sigma\,;\,\Delta\,;\,(\Gamma \vdash \bullet : T \models X) \rightsquigarrow_{\text{refine}} e \dashv G$$

$$Filter(X) = \{\, (E \vdash \bullet \models ex) \in X \mid ex \neq \top \,\}$$

[Refine-Unit]

$$\frac{Filter(X) = (E_1 \vdash \bullet \models ())\,, \ldots, (E_n \vdash \bullet \models ())}{(\Gamma \vdash \bullet : () \models X) \rightsquigarrow_{\text{refine}} () \dashv -}$$

[Refine-Pair]

$$\frac{\begin{array}{c}\textit{New Goals, } i = 1, 2 \qquad Filter(X) = \{\, (E_j \vdash \bullet \models (ex_{j1}\,, ex_{j2})) \,\}^{j \in [m]} \\ h_i \text{ fresh} \quad G_i = (\Gamma \vdash \bullet_{h_i} : T_i \models X_i) \quad X_i = (E_1 \vdash \bullet \models ex_{1i})\,, \ldots, (E_m \vdash \bullet \models ex_{mi})\end{array}}{(\Gamma \vdash \bullet : (T_1\,, T_2) \models X) \rightsquigarrow_{\text{refine}} (??_{h_1}\,, ??_{h_2}) \dashv G_1, G_2}$$

[Refine-Ctor]

$$\frac{\begin{array}{c}\textit{New Goal} \qquad Filter(X) = \{\, (E_j \vdash \bullet \models C\ ex_j) \,\}^{j \in [m]} \qquad \Sigma(D)(C) = T \\ h_1 \text{ fresh} \quad G_1 = (\Gamma \vdash \bullet_{h_1} : T \models X_1) \quad X_1 = (E_1 \vdash \bullet \models ex_1)\,, \ldots, (E_m \vdash \bullet \models ex_m)\end{array}}{(\Gamma \vdash \bullet : D \models X) \rightsquigarrow_{\text{refine}} C\ ??_{h_1} \dashv G_1}$$

[Refine-Fix]

$$\frac{\begin{array}{c}\textit{New Goal} \qquad Filter(X) = (E_1 \vdash \bullet \models \{v_1 \to ex_1\}), \ldots, (E_m \vdash \bullet \models \{v_m \to ex_m\}) \\ h_1 \text{ fresh} \quad e = \texttt{fix}\ f\ (\lambda x.??_{h_1}) \quad G_1 = (\Gamma, f : T_1 \to T_2, x : T_1 \vdash \bullet_{h_1} : T_2 \models X_1) \\ X_1 = (E_1, f \mapsto [E_1]\,e, x \mapsto \lfloor v_1 \rfloor \vdash \bullet \models ex_1), \ldots, (E_m, f \mapsto [E_m]\,e, x \mapsto \lfloor v_m \rfloor \vdash \bullet \models ex_m)\end{array}}{(\Gamma \vdash \bullet : T_1 \to T_2 \models X) \rightsquigarrow_{\text{refine}} e \dashv G_1}$$

Type-and-Example-Directed Branching

$$\Sigma\,;\,\Delta\,;\,F\,;\,(\Gamma \vdash \bullet : T \models X) \rightsquigarrow_{\text{branch}} e \dashv G\,;K$$

[Branch-Case]

$$\frac{\begin{array}{c}\Sigma(D) = \{C_i\ T_i\}^{i \in [n]} \qquad (\Gamma \vdash \bullet : D) \rightsquigarrow_{\text{guess}} e \qquad Filter(X) = \{\, (E_j \vdash \bullet \models ex_j) \,\}^{j \in [m]} \\ \{\, E_j \vdash e \Rightarrow r_j \qquad C_{\alpha_j} \in \{C_1, \ldots, C_n\} \qquad F \vdash e \rightleftharpoons (E_j \vdash \bullet \models C_{\alpha_j}\ \top) \dashv K_j \,\}^{j \in [m]} \\ \textit{New Goals, } i = 1, 2, \ldots, n \\ h_i \text{ fresh} \quad G_i = (\Gamma, x_i : T_i \vdash \bullet_{h_i} : T \models X_i) \\ X_i = \{\, (E_j, x_i \mapsto [\![C_i^{-1}\ r_j]\!] \vdash \bullet \models ex_j) \mid j \in [m] \wedge C_{\alpha_j} = C_i \,\}\end{array}}{F\,;\,(\Gamma \vdash \bullet : T \models X) \rightsquigarrow_{\text{branch}} \texttt{case}\ e\ \texttt{of}\ \{C_i\ x_i \to ??_{h_i}\}^{i \in [n]} \dashv G_1, \ldots, G_n\,;K_1 \oplus \ldots \oplus K_m}$$

Fig. 9. Guessing, Refinement, and Branching.

Guessing-and-Checking. The Guess-and-Check rule uses the procedure $(\Gamma \vdash \bullet : T) \rightsquigarrow_{\text{guess}} e$ in Figure 9 to *guess* a well-typed expression without holes. Guessing amounts to straightforward inversion of expression type checking rules; the appendix (§A.6) provides the full definition.

The candidate expression e is checked for consistency against the examples X using live bidirectional example checking (cf. §3.5 and Figure 6). Whereas example checking in Myth produces a Boolean outcome, example checking in Core Smyth may assume constraints K over other holes. The constraints that arise from (live bidirectional) example checking are the source of the aforementioned differences (i), (ii), and (iii) compared to the Myth hole synthesis procedure.

Refinement. The Refine rule refers to the *refinement* procedure $(\Gamma \vdash \bullet : T \models X) \rightsquigarrow_{\text{refine}} e \dashv G$ in Figure 9 to quickly synthesize a partial solution e which refers to freshly created holes $??_{h_1}$ through $??_{h_n}$ described by subgoals G. Using these results, Refine generates output constraints comprising the partial solution $h \mapsto e$ and the new unfilled holes $h_1 \mapsto X_1$ through $h_n \mapsto X_n$. For the purposes of metatheory, the typings for fresh holes are recorded in the hole type context Δ'.

Each refinement rule first uses $Filter(X)$ to remove top examples and then inspects the structure of the remaining examples. For unit-type goals, Refine-Unit simply synthesizes the unit expression $()$. For pair-type goals, Refine-Pair synthesizes the partial solution $(??_{h_1}, ??_{h_2})$, creating two subgoals from the type and examples of each component. The Refine-Ctor rule for datatype goals D works similarly when all of the examples share the same constructor C.

The refinement rules described so far are essentially the same as proposed by Osera and Zdancewic [2015]. But rather than explicitly naming subgoals G and "sending" them to a top-level *Solve* procedure, the refinement rules in Myth recursively call hole synthesis to solve subgoals immediately. In Core Smyth, we separate the creation of subgoals from solving in order to facilitate the "global" reasoning necessary to synthesize recursive function literals without trace-complete examples, discussed next.

For function-type goals, the Refine-Fix rule synthesizes the function sketch $\texttt{fix}\ f\ (\lambda x.\, ??_{h_1})$. The environments inside example constraints X_1 for the function body $??_{h_1}$ bind f to this function sketch (closed by the appropriate environments E_i). As a result, any recursive calls to f will evaluate to closures of $??_{h_1}$ (to be constrained by live bidirectional example checking), thus avoiding the need for trace-complete examples.[1]

Branching. Lastly, the Branch rule refers to the procedure F; $(\Gamma \vdash \bullet : T \models X) \rightsquigarrow_{\text{branch}} e \dashv G ; K$ in Figure 9 to guess an expression on which to *branch*. (As mentioned, the signature of the branching procedure extends refinement with the additional input F and additional output K.)

The single rule, Branch-Case, chooses an arbitrary expression e (of arbitrary datatype D) to scrutinize, synthesizing the sketch $\texttt{case}\ e\ \texttt{of}\ \{C_i\ x_i \to ??_{h_i}\}^{i \in [n]}$ with subgoals h_i for each of the the constructors C_1 through C_n for the datatype D. The main task is to *distribute* the examples X onto appropriate subgoals. To determine which subgoal should be responsible for the jth example, the guessed scrutinee e is evaluated under the example constraint environment E_j to a result r_j.

Consider the particular scenario in which r_j has determinate form $C_i\ r'_j$, for some constructor C_i. The C_i branch will surely be taken under environment E_j, so the constraint $(E_j,\ x_i \mapsto r'_j \vdash \bullet \models ex_j)$ is added to the examples X_i for the subgoal of that branch. If r_j is *indeterminate*, however, we cannot be sure "which way" the scrutinee will evaluate and thus which branch to "assign" the subgoal.

[1] For the constraint environments in $K_{4.1}$ and $K_{4.2}$ in §2.1, the refinement rule for recursive functions in Myth would bind `plus` to trace-complete examples $\{\texttt{0 1} \to \texttt{1},\ \texttt{2 0} \to \texttt{2},\ \texttt{1 2} \to \texttt{3},\ \ldots\}$. In addition to usability obstacles of trace-completeness, their theory is complicated by a non-standard *value compatibility* notion [Osera and Zdancewic 2015, §3.3] to approximate value equality because input-output examples serve as a "lookup table" to resolve recursive calls.

Therefore, in general, Branch-Case non-deterministically chooses a branch $\alpha_j \in [n]$ for each example j and relies on unevaluation to determine whether r_j can satisfy $C_{\alpha_j}\ \top$ (assuming some constraints K_j). The *constructor simplification* operation $[\![r]\!] = (\text{if } r = C_i{}^{-1}\ (C_i\ r') \text{ then } r' \text{ else } r)$ helps streamline the determinate and indeterminate scenarios in the definition of Branch-Case. This flexibility—analogous to U-Case (cf. Figure 6)—is needed to synthesize several *inside-out* recursive functions (without trace-complete examples), as described in the next section.

Theorem (Soundness of Synthesis).
If $\Sigma\,;\Delta \vdash p : T$ *and* $p \Rightarrow r\,; A$ *and* $\mathit{Simplify}(A) \triangleright K$ *and* $\Sigma\,;\Delta\,;\mathit{Solve}(K) \rightsquigarrow F\,;\Delta'$,
then $\Sigma \vdash F : \Delta'$ *and* $F \models A$.

5 IMPLEMENTATION

We implemented Smyth (https://github.com/UChicago-PL/smyth) in approximately 6,500 lines of OCaml code, not including the front-end to Smyth nor the experimental setup. Compared to the core language in Figure 3, our implementation supports Haskell/Elm-like syntax, n-ary tuples, `let`-bindings, `let`-bound recursive function definitions, and user-defined datatypes. Our implementation also supports higher-order function examples (used in the experiments below) and polymorphism (not used below, but described in §C) following Osera and Zdancewic [2015] and Osera [2015], respectively; these features are orthogonal to our contributions.

Our prototype lacks many of the syntactic conveniences used in code listings in §1 and §2 such as nested pattern matching, infix list operators `(::)` and `(++)`, and type inference for holes. Following Myth, we synthesize only structurally decreasing recursive functions, and we further require that the first argument to a recursive call be structurally decreasing. These are not fundamental challenges, but they result in slightly different code than shown in the paper.

Optimizations. We adopt two primary optimizations from Myth. The first is to guess and cache only *proof relevant* [Anderson et al. 1992] elimination forms—variables x or calls $f\ e_1 \cdots e_n$ to variable-bound functions. The second is a *staging* approach to incrementally increase the maximum branching depth, the size of terms to guess as scrutinees, and the size of terms to guess in other goal positions. We generally adopt the same parameters used by Osera [2015], but with additional intermediate stages to favor small solutions. Furthermore, our parameters are "sketch-sensitive": `case` expressions in the sketch, if any, count against the branching depth budget.

To rein in the non-determinism of case unevaluation, our implementation is configured, first, to guess only variables and projections for the $\mathit{Guesses}(\Delta, \Sigma, r)$ procedure in the "eager" U-Case-Guess rule and, second, to bound the number of nested uses of the "lazy" U-Case rule.

6 EXPERIMENTS

We consider several questions regarding how our techniques—which address Limitations A and B of prior evaluator-based synthesis (§1)—translate into practical gains for users of synthesis tools.

- Compared to prior evaluator-based synthesizers, does Smyth reduce the number of examples required to synthesize top-level, single-hole tasks?
- Unlike prior evaluator-based synthesizers, does Smyth support sketching tasks? Is the total specification burden less than when using examples alone?
- Can state-of-the-art logic-based synthesizers complete all tasks that Smyth can?

To shed light on these questions, we designed four experiments based on the benchmarks used to evaluate Myth. Expert examples are the de facto method for evaluating the *raw expressiveness* of synthesis techniques (e.g. [Albarghouthi et al. 2013; Feser et al. 2015; Frankle et al. 2016; Osera and Zdancewic 2015]). A notable exception is how Feser et al. [2015] evaluate the *robustness* of λ^2 using

randomly-generated examples as a "'lower bound' on a human user ... who has no prior exposure to program synthesis tools." Inspired by these approaches, our experiments consider both "expert" and "random" users to investigate Smyth's expressiveness and robustness.

We ran each of the Smyth experiments on a Mid 2012 MacBook Pro with a 2.5 GHz Intel Core i5 CPU and 16 GB of RAM. We describe each experimental setup and summarize the results (Figure 10) in turn, followed by a discussion including limitations.

6.1 Experiment 1: No Sketches + Trace-Complete Examples

As a baseline experiment, we first run Smyth on each Myth benchmark—a top-level, single-hole task specified with the "full" set of trace-complete expert examples reported by Osera [2015]. Figure 10 (column 1) indicates that Smyth passes 38 of the same 43 benchmarks (without sketches) in a similar amount of time (cf. [Osera 2015]).

Of the five Myth benchmarks that failed in Experiment 1, Smyth produced an over-specialized solution for one (`list_even_parity`) and did not terminate within 120 seconds for the remaining four (`list_compress`, `tree_binsert`, `tree_nodes_at_level`, and `tree_postorder`). The over-specialized term Smyth synthesized for `list_even_parity` was smaller (AST size 14) than the desired term (size 16), which was correctly synthesized by Myth. (Smyth synthesizes and ranks the desired term second.) It is unclear why Myth did not find and return the smaller solution, which is consistent with the examples provided; nevertheless, we classify this task as a failure. The four benchmarks for which Smyth did not terminate are discussed further in §6.5.

Our validation process—which checks synthesized terms against a random set of examples from a reference implementation—revealed that the solution for `list_filter` reported by Osera [2015, p.171] is incorrect. As a workaround, we added one more (trace-complete) example to the reported set of 8 examples and observed that Smyth synthesized a correct solution. We treat these 9 examples (marked with an asterisk in Figure 10) as the set of Myth expert examples for this task.

6.2 Experiment 2: No Sketches + Non-Trace-Complete Examples

Second, we measured how many examples—both expert and random—Smyth requires to synthesize the Myth tasks when not limited to the trace-complete examples from Experiment 1.

Experiment 2a: No Sketches + Expert Examples. To construct expert examples for Smyth on each of the 38 benchmarks it can synthesize, we manually removed sets of examples from the full test suite until Smyth no longer synthesized a correct solution, i.e. a solution that conforms to a reference implementation of the desired solution. As such, there are no corresponding tasks for the five benchmarks that failed Experiment 1, as indicated by "$\bullet^1$" in Figure 10.

Of the 38 benchmarks, Figure 10 (column 2a) shows that Smyth required fewer examples to synthesize all but four benchmarks (`bool_neg`, `bool_xor`, `list_length`, and `nat_max`), requiring on average 61% of the number of expert examples required by Myth, with similar running times as in the baseline configuration (timing data not shown). To account for the 5 missing benchmarks, if we were to assume that Smyth were extended with the Myth-style trace-complete approach to synthesizing recursive functions as a backup synthesis procedure and that the remaining benchmarks would require all of the expert examples, then Smyth would require on average 66% of the number of examples for the entire benchmark suite.

Experiment 2b: No Sketches + Random Examples. To evaluate the robustness of Myth, we implemented a random example generator. For simplicity, our random generator does not support function types; therefore, we did not consider the 4 higher-order function benchmarks (`list_filter`, `list_fold`, `list_map`, and `tree_map`; these are marked "$\bullet^2$" in Figure 10). We also did not consider the four benchmarks that timed out in Experiment 1.

	Smyth						Leon		Synquid	
Experiment	**1**		**2a**	**2b**	**3a**	**3b**	**4**		**4**	
Sketch / Objective	*None / Top-1*		*None / Top-1*		*Base Case / Top-1-R*					
Name	**Expert**	**Time**	**Expert**	**Random** (50%, 90%)	**Expert**	**Random** (50%, 90%)	**1**	**2a**	**1**	**2a**
bool_band	4	0.004	3 (75%)	(4,4)	•[3]	•[3]	✓	✓	✓	✓
bool_bor	4	0.003	3 (75%)	(4,4)	•[3]	•[3]	✓	✓	✓	✓
bool_impl	4	0.004	3 (75%)	(4,4)	•[3]	•[3]	✓	✓	✓	✓
bool_neg	2	0.001	2 (100%)	(2,2)	•[3]	•[3]	✓	•[4]	✓	•[4]
bool_xor	4	0.009	4 (100%)	(4,4)	•[3]	•[3]	✓	•[4]	✓	•[4]
list_append	6	0.008	4 (67%)	(3,4)	1+1 (33%)	(1+3,1+4)	✓	✗[1]	✓	✗[1]
list_compress	13	timeout	•[1]	•[1]	•[1]	•[1]	•[1]	•[1]	•[1]	•[1]
list_concat	6	0.010	3 (50%)	(2,4)	incorrect	(1+3,1+5)	✓	✗[1]	✗[1]	✗[1]
list_drop	11	0.092	5 (45%)	(6,9)	1+2 (27%)	(1+7,↓)	✓	✓	✓	✗[0]
list_even_parity	7	overspec	•[1]	(–,–)	•[1]	(–,–)	•[1]	•[1]	•[1]	•[1]
list_filter	9*	0.144	5 (56%)	•[2]	1+4 (56%)	•[2]	✗[2]	✗[2]	✗[2]	✗[2]
list_fold	9	0.838	3 (33%)	•[2]	1+3 (44%)	•[2]	✗[2]	✗[2]	✗[2]	✗[2]
list_hd	3	0.003	2 (67%)	(2,3)	•[3]	•[3]	✓	✓	✓	✓
list_inc	4	0.018	2 (50%)	(2,2)	•[3]	•[3]	✓	✓	✗[0]	✗[1]
list_last	6	0.007	4 (67%)	(5,9)	1+2 (50%)	(1+5,1+10)	✓	✓	✓	✗[0]
list_length	3	0.002	3 (100%)	(3,4)	1+1 (67%)	(1+2,1+2)	✓	•[4]	✓	•[4]
list_map	8	0.049	4 (50%)	•[2]	1+2 (38%)	•[2]	✗[2]	✗[2]	✗[2]	✗[2]
list_nth	13	0.124	5 (38%)	(7,14)	1+2 (23%)	(1+7,1+15)	✓	✓	✓	✗[0]
list_pairwise_swap	7	0.634	5 (71%)	timeout	overspec	timeout	✓	✓	✗[0]	✗[0]
list_rev_append	5	0.107	3 (60%)	(5,8)	1+2 (60%)	(1+3,1+4)	✓	✓	✗[0]	✗[0]
list_rev_fold	5	0.035	2 (40%)	(2,4)	•[3]	•[3]	✓	✓	✗[0]	✗[0]
list_rev_snoc	5	0.010	3 (60%)	(3,6)	1+1 (40%)	(1+2,1+4)	✓	✓	✗[1]	✗[0]
list_rev_tailcall	8	0.008	3 (38%)	(3,4)	1+1 (25%)	(1+3,1+5)	✗[1]	✓	✓	✗[1]
list_snoc	8	0.012	3 (38%)	(3,4)	1+1 (25%)	(1+3,1+4)	✓	✓	✓	✗[0]
list_sort_sorted_insert	7	0.015	3 (43%)	(3,6)	1+1 (29%)	(1+2,1+4)	✓	✓	✗[0]	✗[1]
list_sorted_insert	12	2.902	7 (58%)	timeout	1+7 (67%)	timeout	✗[0]	✗[0]	✗[0]	✗[0]
list_stutter	3	0.003	2 (67%)	(3,3)	1+1 (67%)	(1+2,1+3)	✓	✓	✓	✗[1]
list_sum	3	0.029	2 (67%)	(2,2)	•[3]	•[3]	✓	✗[1]	✗[0]	✗[0]
list_take	12	0.065	5 (42%)	(6,9)	1+3 (33%)	(1+7,1+16)	✓	✓	✓	✗[0]
list_tl	3	0.002	2 (67%)	(2,3)	•[3]	•[3]	✓	✓	✓	✓
nat_add	9	0.006	4 (44%)	(5,6)	1+1 (22%)	(1+3,1+4)	✓	✓	✓	✗[1]
nat_iseven	4	0.003	3 (75%)	(4,4)	1+2 (75%)	(1+3,1+4)	✓	✓	✓	✗[0]
nat_max	9	0.041	9 (100%)	(8,12)	1+4 (56%)	(1+8,1+12)	✗[1]	•[4]	✓	•[4]
nat_pred	3	0.001	2 (67%)	(2,3)	•[3]	•[3]	✓	✓	✓	✓
tree_binsert	20	timeout	•[1]	•[1]	•[1]	•[1]	•[1]	•[1]	•[1]	•[1]
tree_collect_leaves	6	0.074	3 (50%)	$(3,4)^{t=3}$	1+2 (50%)	(1+3,1+3)	✓	✓	✗[1]	✗[1]
tree_count_leaves	7	2.660	3 (43%)	timeout	1+1 (29%)	timeout	✓	✓	✗[0]	✗[0]
tree_count_nodes	6	0.351	3 (50%)	$(4,\downarrow)^{t=10}$	1+2 (50%)	$(1+3,1+5)^{t=3}$	✓	✓	✗[1]	✗[0]
tree_inorder	5	0.123	4 (80%)	(3,4)	1+2 (60%)	(1+3,1+4)	✓	✓	✗[1]	✗[0]
tree_map	7	0.061	4 (57%)	•[2]	1+3 (57%)	•[2]	✗[2]	✗[2]	✗[2]	✗[2]
tree_nodes_at_level	11	timeout	•[1]	•[1]	•[1]	•[1]	•[1]	•[1]	•[1]	•[1]
tree_postorder	20	timeout	•[1]	•[1]	•[1]	•[1]	•[1]	•[1]	•[1]	•[1]
tree_preorder	5	0.153	3 (60%)	$(3,4)^{t=3}$	1+2 (60%)	(1+3,1+3)	✓	✓	✗[1]	✗[1]
Averages			61%*		46%					

Fig. 10. Experiments.
Top-1(-R): 1st (recursive) solution valid. **Time**: Average of 10 runs, in seconds.
2a Average: 61% for 38 non-blank rows. (*Upper bound: 66% for all 43 rows.)
3a Average: 46% for 25 non-blank, non-error rows.

For each of the remaining 35 tasks, we generated $N = 50$ sets of k random input examples (where k ranges from 1 to a reasonable upper bound depending on the benchmark) and used a task reference implementation to compute the corresponding outputs, thus producing N sets of input-output example sets of size k for each k. We fixed relatively small upper bounds on the AST sizes of the input examples generated to ensure the examples could reasonably be provided by a human, and,

rather than sampling inputs uniformly at random—in which case, e.g., a list of length 3 would be twice as likely as a list of length 2—we first sampled different *shapes* for the data structures (`List`s and `Tree`s) uniformly at random, then filled in base values at the AST leaves uniformly at random. Furthermore, we required that each set of examples (regardless of size) contains the unique "minimal input" to the function, that is, the input that consists of the minimal value for each type of each argument of the function, where, for `Nat`s, the minimal value is `0`, for `List`s, it is the empty list, and for `Tree`s, it is a leaf.

Entries in Figure 10 (column 2b) show two values: the minimum k for which Smyth synthesized the desired solution within a $t = 1$ second timeout for 50% of the N sets of examples, and the minimum such k to achieve 90% success; the appendix (§B) includes graphs for each benchmark. Several entries require explanation. Two benchmarks are marked with a superscript "$t = 3$" (`tree_collect_leaves` and `tree_preorder`) and one benchmark is marked with a superscript "$t = 10$" (`tree_count_nodes`) to indicate they they required a longer timeout. For `tree_count_nodes`, we do not report the minimum k value for 90% (marked "↓"), because the percentage dips below for subsequent values of k. One benchmark is marked "(—,—)" (`list_even_parity`) and did not achieve a 50% success for reasonably-small values of k. For this benchmark, we hypothesize that our simply-typed approach cannot glean enough information from its input type, `BooleanList`.

To analyze these k-values, we consider the difference $k'_p := k_p - k_{\text{expert}}$ for each benchmark that was successfully synthesized in this experiment, where p is the required success rate (either 50% or 90%) and k_{expert} is the number of Smyth expert examples for Experiment 2a. The value k'_p thus represents how many more examples are needed, compared to the expert set, to achieve success p% of the time. The adjacent table summarizes the distribution of k'_p for Experiment 2b; additional statistics and corresponding histograms can be found in the appendix (§B).

	median k'_p	max k'_p
p = 50%	0	2
p = 90%	1	9

6.3 Experiment 3: Base Case Sketching Strategy

Experiments 1 and 2 considered tasks without sketches from the user. As a third experiment, we systematically converted the Myth benchmarks into a suite of small sketching tasks by employing a simple *base case sketch strategy*—performing case analysis on the correct argument of the function, filling in the base case properly, and leaving a hole in the recursive branch. Of the 38 tasks, 27 are recursive and thus subject to this strategy. The remaining, non-recursive tasks are marked "^{3}".

In Figure 10 and the following, we write $1 + n$ to denote a specification with n examples in addition to the base case sketch; our accounting treats the specification burden of the base case sketching strategy as equivalent to 1 example. (We could report AST sizes of sketches and examples, but even these would be just a rough proxy for the "complexity" of a specification.)

Experiment 3a: Base Case Sketches + Expert Examples. Analogous to Experiment 2a, we manually removed sets of examples from the full trace-complete expert examples until Smyth no longer successfully completed the task. For this experiment, however, because the base case strategy pertains to recursive functions, we considered a task successful if the smallest *recursive* solution was correct, rather than simply the smallest solution overall. Figure 10 (column 3a) shows the results of this experiment.

For 25 of these 27 tasks that succeeded, Smyth on average required smaller total specifications with base case sketches than with no sketches. On average, specifications were 46% the size of the full trace-complete examples—compared to 57% without a sketch (average, not shown, of 25 rows in the Experiment 2a column). Given the sketches, the average number of examples required was 2.12; `list_sorted_insert` required 7, while the rest required between 1 and 4.

Three tasks that succeeded (`list_filter`, `list_pairwise_swap`, and `list_sorted_insert`) required sketch-sensitive staging parameters (§ 5). This is because Smyth's staging parameters increase branching depth before scrutinee size, and a relatively large scrutinee is needed for the desired solution; compared to when no sketch is provided, sketch-insensitive staging parameters effectively "penalize" the sketch for having introduced a case. Before we accounted for branching depth in the user-provided sketch, Smyth synthesized overspecialized solutions for these three tasks even with the full set of Myth expert examples.

Two of the 27 tasks failed this experiment. For `list_even_parity`, Smyth synthesized an over-specialized solution (even with sketch-sensitive staging parameters). For `list_concat`, Smyth actually synthesized "`list_rev_concat`," which appends together a list of lists in *reverse* order. The Myth expert examples are not sufficient to distinguish these two functions; Smyth returns both, but they have the same AST size and the desired solution is arbitrarily ranked second.

Experiment 3b: Base Case Sketches + Random Examples. Analogous to Experiment 2b, we generated random input-output examples for the benchmarks, this time in addition to providing the base case sketches. We again consider the difference $k'_p := k_p - k_{\text{expert}}$ for each benchmark that was successfully synthesized in this experiment, where k_{expert} is now the number of Smyth expert examples for Experiment 3a rather than for Experiment 2a. The adjacent table summarizes the distribution of k'_p for Experiment 3b; additional data can be found in the appendix (§B).

	median k'_p	max k'_p
$p = 50\%$	2	6
$p = 90\%$	4	14

6.4 Experiment 4: Programming-by-Example in Leon and Synquid

The previous experiments evaluate the improvements in Smyth compared to prior evaluator-based techniques. In our final experiment, we run several of our "programming-by-example" tasks on Leon and Synquid. The goal is to understand whether—from the perspective of a user who wishes to specify tasks through examples—Leon or Synquid are strictly more powerful than Smyth. That is, can Leon or Synquid solve every task that Smyth can?

We systematically generated Scala and Haskell versions of our benchmarks to test Leon and Synquid, respectively. Because this experiment is designed to answer a very simple question, we did not develop a thorough experimental environment with random examples or multiple trials. Instead, we used web interfaces to Leon and Synquid to test benchmarks.[2]

First, we tested the small sketching tasks from § 1 and § 2. As described in § 1, both tools fail to complete the `stutter_n` task. We also found that Synquid fails to complete the four sketching tasks from Figure 2 and that Leon successfully completes `max` and `odd` but fails on `minus` and `mult`.

We then tested the tools for the top-level, single-hole tasks used in Experiments 1 and 2a with trace-complete and non-trace-complete expert examples, respectively. Besides the function to synthesize, we used simple types (without examples or precise logical predicates) for all functions in the context. Four benchmarks had the same number of expert examples in Experiment 2a as they did in Experiment 1 and thus do not have corresponding tasks in Experiment 4 (marked "$\bullet^4$").

Figure 10 (columns 4) show the results. Leon and Synquid successfully completed many tasks (marked ✓), but failed several tasks for a variety of reasons: terminating without producing solutions or not terminating within a timeout (✗^{0}); returning over-specialized solutions (✗^{1}); and not being able to directly express higher-order function examples (✗^{2}). As expected, Synquid failed to synthesize recursive functions without inductive (i.e. trace-complete) specifications (column 4, 2a).[3]

[2] https://leon.epfl.ch/ and http://comcom.csail.mit.edu/comcom/#Synquid. Accessed February 2020 and May 2020.

[3] Earlier results from this experiment revealed an implementation issue in Synquid involving the axiomatization of recursive datatypes in the underlying logic. This issue—which prevented the desired solutions for many benchmarks from typechecking, even when given trace-complete examples—has since been fixed [Polikarpova 2020].

These results are not entirely surprising, as the underlying techniques are not necessarily tailored to the structure of examples encoded as conjunctions-of-implications. This suggests opportunities for further improvements to both evaluator- and logic-based techniques, for instance, by integrating live bidirectional evaluation into more fine-grained logic-based techniques.

As a final note, this experiment was *not* intended to evaluate whether Smyth is "better" than the logic-based tools. Indeed, many tasks involving complex invariants are beyond the reach of evaluator-based techniques, Smyth included. Polikarpova et al. [2016, §4.3] provide some empirical comparison between example-based and logic-based specifications on several common benchmarks.

6.5 Limitations and Discussion

Failing Benchmarks. One major optimization in Myth that we have not implemented is to cache solutions F—which correspond to Myth's "refinement trees"—across branches of the search. This optimization does not directly carry over to our setting because, unlike in Myth, synthesized terms in Smyth may introduce different, conflicting assumptions across different branches of search. Thus, our first hypothesis is that suitably extending caching to our setting could help synthesize the remaining tasks (although the difficulty of this task is unclear).

Of the five benchmarks not successfully synthesized in our implementation, Myth finds four solutions with *inside-out recursion* [Osera 2015], which pattern match on a recursive call to the function being synthesized. Inside-out solutions are smaller than more "natural" ones, and sometimes they are the only solutions to tasks in Myth and Smyth because only elimination forms are enumerated and `let`-bindings are not synthesized [Osera 2015]. Although Smyth does synthesize an inside-out solution for one benchmark (`list_pairwise_swap`), inside-out recursion relies heavily on the non-determinism of Branch-Case and U-Case. Accordingly, our second hypothesis is that additional tuning for these sources of non-determinism could help synthesize the necessary inside-out recursion.

Scalability. Each benchmark in our experiments included the minimal context—as defined in the Myth benchmarks—required to synthesize the desired solution. In addition to minimal contexts, the Myth paper also reported results in the presence of a slightly larger context and ran into scalability issues on some benchmarks. Though we did not run these versions of the benchmarks, we inherit any scalability issues of the prior techniques.

Moreover, our approach introduces new sources of non-determinism. To scale to much larger programs with complex control flow, static reasoning (interleaved with concrete evaluation) could be used to prune unsatisfiable or heuristically "difficult" sets of example constraints. Orthogonal techniques for scaling to large contexts with additional components [Feng et al. 2017b; Guo et al. 2020; Gvero et al. 2013] might also be incorporated into our approach in future work.

Assertions. Our formulation and thus our benchmarks support only top-level `asserts`. To allow `asserts` in arbitrary expressions (as needed for larger and more realistic sketching tasks), evaluation and resumption could be extended to generate assertions A as a side-effect, to be translated by *Simplify* into constraints for synthesis. We expect the algorithmic changes to be straightforward, but the extended definition of assertion satisfaction along with the corresponding correctness properties and proofs are more delicate; we leave this task for future work.

Polymorphism. Of the 38 tasks that Smyth successfully synthesized in Experiment 1, 23 can be specified with a polymorphic type signature rather than a monomorphic one. We re-ran Experiments 2 and 3 with polymorphic type signatures, which are supported in our implementation but are not included in our formal development. As described in the appendix (§C), polymorphic type signatures lead to a modest reduction in the number of examples needed for synthesis.

7 RELATED WORK

Our work generalizes the theory of evaluator-based synthesis techniques to (a) eliminate the need for trace-complete examples and (b) to support sketching—addressing Limitations A and B from §1. We build directly on the work of Osera and Zdancewic [2015], so we discussed Myth throughout the paper. To conclude, we discuss several additional directions of related work.

7.1 Live Evaluation and Bidirectional Evaluation

The key technical mechanism underlying our approach is live bidirectional evaluation, the combination of live evaluation and live unevaluation. We choose the term "live" to describe partial evaluation of sketches, following terminology of Omar et al. [2019]. Future work must address important usability and scalability questions to further develop and deploy our techniques in interactive, *live programming* environments [Kubelka et al. 2018; Tanimoto 2013].

Live Evaluation (Hazelnut Live). We adapt the technique for partially evaluating sketches from Hazelnut Live [Omar et al. 2019]. In contrast to solver-based and symbolic execution techniques for partially evaluating programs with holes (e.g. [Bornholt and Torlak 2018; Feng et al. 2017a; Wang et al. 2020]), live evaluation is a form of *concrete* evaluation, adapting ideas from *contextual modal type theory* [Nanevski et al. 2008]. Omar et al. [2019, §5] detail the relationship to related work on *partial evaluation*. Hazelnut Live does not offer any form of synthesis; their "fill-and-resume" feature refers to ordinary program edits by the user.

We note some technical differences in our formulation. We choose a natural semantics presentation [Kahn 1987] for Core Smyth rather than one based on substitution. Whereas their fill-and-resume mechanism is defined using contextual substitution, our formulation instead defines evaluation resumption. Hazelnut Live also includes hole types to support gradual typing [Siek and Taha 2006; Siek et al. 2015], a language feature orthogonal to the (expression) synthesis motivations for our work. Finally, Omar et al. [2019] present a bidirectional type system [Chlipala et al. 2005; Pierce and Turner 2000] that, given type-annotated functions, computes hole environments Δ; the same approach can be employed in our setting without complication.

Bidirectional Evaluation (Sketch-n-Sketch). Several proposals define *unevaluators*, or *backward evaluators*, that allow changes to the output value of an expression (without holes) to affect changes to the expression [Matsuda and Wang 2018; Mayer et al. 2018; Perera et al. 2012]. Though related by analogy and terminology, our novel live unevaluation mechanism shares essentially no technical overlap with the above techniques. The prior backward evaluators essentially only modify constant literals of base type—which can be thought of as "non-empty" holes that are subject to replacement—at the leaves of an existing program, whereas our live unevaluator propagates example constraints to holes of arbitrary type and in arbitrary position.

An environment-style semantics is purposely chosen for each of the above unevaluators, because value environments provide a sufficient mechanism for tracing value provenance during evaluation. In contrast, our unevaluator could just as easily be formulated with substitution; in either style, hole expressions are labeled with unique identifiers, which provide the necessary information to generate example constraints.

7.2 Program Synthesis

We conclude with a broader discussion of the evaluator- and logic-based synthesis techniques that we introduced in §1. We use the term "functional programming"—in contrast to "domain-specific"—to describe languages in which users (and synthesizers) write unrestricted programs in a richly-typed functional language (i.e. with directly recursive functions on algebraic datatypes).

7.2.1 Evaluator-Based Synthesis Techniques. We chose this term in § 1 to describe synthesis algorithms in which the core search strategy uses *concrete* evaluation to "check" candidate terms, typically against input-output example specifications.

Programming-by-Example (PBE) for Domain-Specific Languages. Programming-by-example techniques have been developed for numerous domain-specific applications, including string transformations [Gulwani 2011] (including bidirectional ones [Miltner et al. 2019]), shell scripting [Gulwani et al. 2015], web scraping [Chasins et al. 2018], parallel data processing [Smith and Albarghouthi 2016], and generating vector graphics [Hempel et al. 2019]. See Gulwani et al. [2017] for a recent survey of developments. These approaches generally synthesize entire programs. To allow experts to provide partial implementations, it should be possible to formulate notions of live bidirectional evaluation of these domain-specific techniques.

λ^2 [Feser et al. 2015] synthesizes functions in a (first-order) functional programming language (with higher-order components). λ^2 enumerates *open hypotheses* (i.e. sketches) involving calls to a fixed set of primitive `List` and `Tree` combinators (e.g. `filter` and `map`), and relies on axioms for *deductive reasoning* to convert examples for a goal into examples for the subgoals. This process is akin to refinement in Myth, and also helps prune unsatisfiable example constraints (e.g. if a `map` hypothesis requires input and output lists of different lengths).

However, function examples are not used used to "refine" the search; their deduction rule for general recursion essentially falls back on raw term enumeration, and their checking routine operates only on *closed hypotheses* (without holes). In other words, examples need not be trace-complete because they are not used to help synthesize recursive function literals. Although the language supported by λ^2 nominally includes direct recursive function literals [Feser et al. 2015, §3], in practice, their implementation synthesizes solutions *only* by composing the primitive data structure combinators [Feser 2016, 2020], and furthermore does not introduce non-trivial matches on inductive data [Feser 2020]. λ^2 can synthesize a variety of functional programming tasks, similar to the Myth and Smyth benchmarks, including with randomly-generated examples (cf. § 6) and with significantly larger contexts than used in the Myth and Smyth experiments. But because λ^2 does not search for directly recursive functions, it is fundamentally a more domain-specific technique than Myth and Smyth.

For the domain of table transformations, Morpheus extends the approach of λ^2 with (i) SMT-based reasoning to perform more powerful deduction and (ii) partial evaluation of sketches. Viser [Wang et al. 2020] further improves upon the techniques in Morpheus, by providing *backward* reasoning about program sketches using symbolic reasoning over logical and subset constraints. (Viser also integrates a domain-specific language for visualization, resulting in a visualization-by-example tool.) Sketches in both Morpheus and Viser are drawn from a first-order, domain-specific language of table transformations. In contrast, Smyth performs bidirectional reasoning about program sketches (a) in a *general-purpose* richly-typed functional programming language (as opposed to domain-specific table transformation languages), (b) using techniques based on concrete evaluation (rather than SMT solving and other symbolic reasoning techniques).

PBE for Functional Programming. Two prior evaluator-based systems synthesize recursive functions. Escher [Albarghouthi et al. 2013] does so for an untyped, first-order functional language (with base types rather than inductive datatypes), relying on run-time type errors to help rule out candidate terms. Myth [Osera and Zdancewic 2015] pioneered the idea to synthesize recursive functions over algebraic datatypes using search techniques inspired by *bidirectional typing* [Pierce and Turner 2000] and *relevant proof search* [Anderson et al. 1992; Byrnes 1999]. Both Escher and Myth require trace-complete examples. As discussed next, the bidirectional typing approach of Myth has influenced several logic-based approaches to synthesis.

7.2.2 Logic-Based Synthesis Techniques. We chose this term in § 1 to describe synthesis algorithms that use *symbolic*, rather than concrete, evaluation to enumerate terms, and which operate on more fine-grained, precise logical specifications than examples.

PBE for Functional Programming via Refinement Types. Frankle et al. [2016] reformulate Myth by recasting concrete examples in a type language of intersection and singleton types. Rather than employing concrete evaluation, they perform (symbolic) proof search within their rich type language. Their formal development includes union and negation types, which allows more than just examples (with concrete input and output values) to be specified. Their implementation further supports type polymorphism, with symbolic values as examples. The combination of negation and polymorphism admit what Polikarpova et al. [2016] dub "generalized examples," which facilitate smaller specifications for several Myth benchmarks. (Generalized examples resemble the *symbolic input-output examples* supported by Leon for program repair [Kneuss et al. 2015].) This reformulation of (generalized) examples suffers the same Limitations A and B as Escher and Myth. It would be valuable to extend Smyth in future work with similar typing constructs.

Program Sketching. Sketch [Solar-Lezama 2008; Solar-Lezama 2009; Solar-Lezama et al. 2005, 2006] is an imperative, C-like language that pioneered the approach of program synthesis by sketching. Rosette [Torlak and Bodik 2013, 2014] further develops this approach within the untyped functional language Racket. Holes in Sketch and Rosette range only over integers and booleans, but these can be used to define richer types of expressions. The mechanisms for such *syntax-guided synthesis* [Alur et al. 2013] are particularly powerful in Rosette, which leverages the metaprogramming facilities in Racket. As Inala et al. [2017, § 6] suggest, one could embed the syntax and semantics of a richly-typed, general-purpose functional programming language in Rosette. There is no obvious reason to expect recursive functions over user-defined algebraic datatypes embedded in this way to be readily synthesized, but this approach would be an interesting experiment.

Solver-Based Techniques for Functional Programming. Synquid [Polikarpova et al. 2016] and Leon [Kneuss et al. 2013] directly support sketching in richly-typed functional languages using solver-based techniques driven by logical specifications. Synquid employs bidirectional typing (like Myth) in a setting with SMT-based refinement types [Rondon et al. 2008; Vazou et al. 2013]. Synquid furthermore introduces *round-trip type checking*, which propagates goal types "through" elimination forms, allowing errors to be localized (i.e. found sooner) during type checking. In a synthesis context, failing sooner means avoiding costly search paths.

Example-based and logic-based specifications are complementary. Combining support for such specifications is another interesting direction for future work. It would be interesting to consider whether live bidirectional evaluation could help eliminate the inductive (i.e. trace-complete) requirement of partial specifications in Synquid, so that its powerful logic-based reasoning could better operate when given examples as partial specifications.

ACKNOWLEDGMENTS

The authors would like to thank Ian Voysey for guidance regarding proof strategies; Nadia Polikarpova, Brian Hempel, Michael Adams, Youyou Cong, and anonymous reviewers for many helpful suggestions; Aws Albarghouthi, John Feser, Viktor Kunčak, and Nadia Polikarpova for answering questions about Escher, λ^2, Leon, and Synquid; and Robert Rand—who coined the name Myth—for suggesting the name Smyth, thus further entangling our work with its predecessor. This work was supported by NSF grants *Semantic Foundations for Hole-Driven Development* (CCF-1814900 and CCF-1817145) and *Direct Manipulation Programming Systems* (CCF-1651794).

REFERENCES

Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. 2013. Recursive Program Synthesis. In *Computer Aided Verification (CAV)*.

Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-Guided Synthesis. In *Formal Methods in Computer-Aided Design (FMCAD)*.

Alan Ross Anderson, Nuel D. Belnap Jr., and J. Michael Dunn. 1992. *Entailment, Vol. II: The Logic of Relevance and Necessity*. Princeton University Press.

James Bornholt and Emina Torlak. 2018. Finding Code That Explodes under Symbolic Evaluation. *Proceedings of the ACM on Programming Languages (PACMPL), Issue OOPSLA* (2018).

John Byrnes. 1999. *Proof Search and Normal Forms in Natural Deduction*. Ph.D. Dissertation. Carnegie Mellon University.

Sarah E. Chasins, Maria Mueller, and Rastislav Bodik. 2018. Rousillon: Scraping Distributed Hierarchical Web Data. In *Symposium on User Interface Software and Technology (UIST)*.

Adam Chlipala, Leaf Petersen, and Robert Harper. 2005. Strict Bidirectional Type Checking. In *Workshop on Types in Languages Design and Implementation (TLDI)*.

Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017a. Component-Based Synthesis of Table Consolidation and Transformation Tasks from Examples. In *Conference on Programming Language Design and Implementation (PLDI)*.

Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W. Reps. 2017b. Component-Based Synthesis for Complex APIs. In *Symposium on Principles of Programming Languages (POPL)*.

John Feser. 2016. Inductive Program Synthesis from Input-Output Examples. Master's Thesis, Rice University.

John Feser. 2020. Personal communication, February 2020.

John K. Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing Data Structure Transformations from Input-Output Examples. In *Conference on Programming Language Design and Implementation (PLDI)*.

Jonathan Frankle, Peter-Michael Osera, David Walker, and Steve Zdancewic. 2016. Example-Directed Synthesis: A Type-Theoretic Interpretation. In *Symposium on Principles of Programming Languages (POPL)*.

Sumit Gulwani. 2011. Automating String Processing in Spreadsheets Using Input-Output Examples. In *Symposium on Principles of Programming Languages (POPL)*.

Sumit Gulwani, Mikaël Mayer, Filip Niksic, and Ruzica Piskac. 2015. StriSynth: Synthesis for Live Programming. In *International Conference on Software Engineering (ICSE)*.

Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. 2017. Program Synthesis. *Foundations and Trends in Programming Languages* 4, 1-2 (2017), 1–119. https://doi.org/10.1561/2500000010

Zheng Guo, David Justo, Michael James, Jiaxiao Zhou, Ziteng Wang, Ranjit Jhala, and Nadia Polikarpova. 2020. Program Synthesis by Type-Guided Abstraction Refinement. *Proceedings of the ACM on Programming Languages (PACMPL), Issue POPL* (2020).

Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. 2013. Complete Completion Using Types and Weights. In *Conference on Programming Language Design and Implementation (PLDI)*.

Brian Hempel, Justin Lubin, and Ravi Chugh. 2019. Output-Directed Programming for SVG. In *Symposium on User Interface Software and Technology (UIST)*.

Jeevana Priya Inala, Nadia Polikarpova, Xiaokang Qiu, Benjamin S. Lerner, and Armando Solar-Lezama. 2017. Synthesis of Recursive ADT Transformations from Reusable Templates. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*.

Gilles Kahn. 1987. Natural Semantics. In *Symposium on Theoretical Aspects of Computer Sciences (STACS)*.

Etienne Kneuss, Manos Koukoutos, and Viktor Kuncak. 2015. Deductive Program Repair. In *Computer Aided Verification (CAV)*.

Etienne Kneuss, Ivan Kuraj, Viktor Kuncak, and Philippe Suter. 2013. Synthesis Modulo Recursive Functions. In *Conference on Object-Oriented Programming Languages, Systems, and Applications (OOPSLA)*.

Juraj Kubelka, Romain Robbes, and Alexandre Bergel. 2018. The Road to Live Programming: Insights from the Practice. In *International Conference on Software Engineering (ICSE)*.

Justin Lubin, Nick Collins, Cyrus Omar, and Ravi Chugh. 2020. Program Sketching with Live Bidirectional Evaluation. Extended version of this ICFP 2020 paper available as *CoRR abs/1911.00583* (https://arxiv.org/abs/1911.00583).

Kazutaka Matsuda and Meng Wang. 2018. HOBiT: Programming Lenses Without Using Lens Combinators. In *European Symposium on Programming (ESOP)*.

Mikaël Mayer, Viktor Kunčak, and Ravi Chugh. 2018. Bidirectional Evaluation with Direct Manipulation. *Proceedings of the ACM on Programming Languages (PACMPL), Issue OOPSLA* (2018).

Anders Miltner, Solomon Maina, Kathleen Fisher, Benjamin C. Pierce, David Walker, and Steve Zdancewic. 2019. Synthesizing Symmetric Lenses. *Proceedings of the ACM on Programming Languages (PACMPL), Issue ICFP* (2019).

Anders Miltner, Saswat Padhi, Todd D. Millstein, and David Walker. 2020. Data-Driven Inference of Representation Invariants. In *Conference on Programming Language Design and Implementation (PLDI)*.

Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. 2008. Contextual Modal Type Theory. *ACM Transactions on Computational Logic (TOCL)* (2008).

Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A. Hammer. 2019. Live Functional Programming with Typed Holes. *Proceedings of the ACM on Programming Languages (PACMPL), Issue POPL* (2019).

Peter-Michael Osera. 2015. *Program Synthesis with Types.* Ph.D. Dissertation. University of Pennsylvania.

Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-Example-Directed Program Synthesis. In *Conference on Programming Language Design and Implementation (PLDI)*.

Roly Perera, Umut A. Acar, James Cheney, and Paul Blain Levy. 2012. Functional Programs That Explain Their Work. In *International Conference on Functional Programming (ICFP)*.

Benjamin C. Pierce and David N. Turner. 2000. Local Type Inference. *ACM Transactions on Programming Languages and Systems (TOPLAS)* (2000).

Nadia Polikarpova. 2020. Personal communication, February and May 2020.

Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program Synthesis from Polymorphic Refinement Types. In *Conference on Programming Language Design and Implementation (PLDI)*.

Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. 2008. Liquid Types. In *Conference on Programming Language Design and Implementation (PLDI)*.

Jeremy G. Siek and Walid Taha. 2006. Gradual Typing for Functional Languages. In *Scheme and Functional Programming Workshop*.

Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. 2015. Refined Criteria for Gradual Typing. In *Summit on Advances in Programming Languages (SNAPL)*.

Calvin Smith and Aws Albarghouthi. 2016. MapReduce Program Synthesis. In *Conference on Programming Language Design and Implementation (PLDI)*.

Armando Solar-Lezama. 2008. *Program Synthesis by Sketching.* Ph.D. Dissertation. UC Berkeley.

Armando Solar-Lezama. 2009. The Sketching Approach to Program Synthesis. In *Asian Symposium on Programming Languages and Systems (APLAS)*.

Armando Solar-Lezama, Rodric Rabbah, Rastislav Bodík, and Kemal Ebcioğlu. 2005. Programming by Sketching for Bit-Streaming Programs. In *Conference on Programming Language Design and Implementation (PLDI)*.

Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial Sketching for Finite Programs. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

Steven L. Tanimoto. 2013. A Perspective on the Evolution of Live Programming. In *Workshop on Live Programming (LIVE)*.

Emina Torlak and Rastislav Bodik. 2013. Growing Solver-Aided Languages with Rosette. In *Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward!)*.

Emina Torlak and Rastislav Bodik. 2014. A Lightweight Symbolic Virtual Machine for Solver-Aided Host Languages. In *Conference on Programming Language Design and Implementation (PLDI)*.

Niki Vazou, Patrick M. Rondon, and Ranjit Jhala. 2013. Abstract rRefinement Types. In *European Conference on Programming Languages and Systems (ESOP)*.

Chenglong Wang, Yu Feng, Rastislav Bodik, Alvin Cheung, and Isil Dillig. 2020. Visualization by Example. *Proceedings of the ACM on Programming Languages (PACMPL), Issue POPL* (2020).