Program Synthesis with Live Bidirectional Evaluation

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

We present an algorithm for completing program sketches (partial programs, with holes), in which evaluation and example-based synthesis are interleaved until the program is complete and produces a value. Our approach combines and extends recent advances in live programming with holes and type-and-example-directed synthesis of recursive functions over algebraic data types. Novel to our formulation is the ability to simultaneously solve interdependent synthesis goals—the key technique, called live bidirectional evaluation, iteratively solves constraints that arise during “forward” evaluation of candidate completions and propagates examples “backward” through partial results.

We implement our approach in a prototype system, called Sketch-N-Myth, and develop several examples that demonstrate how live bidirectional evaluation enables a novel workflow for programming with synthesis. On benchmarks used to evaluate a state-of-the-art example-based synthesis technique, Sketch-N-Myth requires on average 55% of the number of examples (even without sketches) by overcoming the example trace-completeness requirement of previous work. Our techniques thus contribute to ongoing efforts to develop synthesis algorithms that can be deployed in future programming environments.

1 INTRODUCTION

Recent advances have enabled synthesizing complex recursive functions over datatypes in richly-typed functional programming languages. One approach—employed by Leon [Kneuss et al. 2013] and SYNQUID [Polikarpova et al. 2016]—uses fine-grained logical predicates to specify synthesis tasks and solver-based algorithms to complete them. Another approach—employed by Escher [Albarghouthi et al. 2013], λ² [Feser et al. 2015], and Myth [Osera and Zdancewic 2015], and its successor [Frankle et al. 2016]—uses input-output example specifications in various ways. These techniques are expressive, enabling a variety of complex functions to be synthesized entirely. However, these techniques have not yet generally been incorporated into systems that allow the user to provide partial implementations, with missing pieces to be filled in.

The “sketching” approach to program synthesis offers a compelling such workflow: the programmer writes a partial program, called a sketch, that defines the structure of a desired implementation with holes—constrained by ordinary assert statements in the program—to be completed by the synthesizer [Solar-Lezama 2008; Torlak and Bodik 2013]. Holes in the sketch may appear in arbitrary positions and may depend on one another. Program sketching has been thoroughly developed to support completion of integer-typed holes—at compile-time in the imperative C-like language Sketch [Solar-Lezama 2008], and at run-time in the untyped functional language Rosette [Torlak and Bodik 2013, 2014]—but not yet for richly-typed functional programming languages.

As a step towards blending these approaches, we focus our work in this paper on the question: Can example-based program synthesis support sketching for functional programs?

Authors’ addresses: Justin Lubin, University of Chicago, justinlubin@uchicago.edu; Nick Collins, University of Chicago, nickmc@uchicago.edu; Cyrus Omar, University of Michigan, comar@umich.edu; Ravi Chugh, University of Chicago, rchugh@cs.uchicago.edu.
1.1 Example-Directed Synthesis in MYTH

Among the example-based systems, the techniques of Osera and Zdancewic [2015] represent the state-of-the-art with respect to our goals. Given a context of datatypes and value definitions, the user provides MYTH with the name and type of a function to synthesize, along with a set of input-output examples \( \{ i_{m_1} \rightarrow o_{out_1}, \ldots \} \) to specify its behavior.

The MYTH algorithm augments type-based enumerative search—guessing type-correct expressions and then checking example-compatibility—with example-directed refinement to create independent subgoals, according to the structure of the goal type and the examples. When these subgoals cannot be filled by guessing-and-checking literals, variables, or function applications of a reasonable size, MYTH guesses an expression on which to branch and then distributes the examples to subgoals for the branches. Compared to naïve type-based enumeration, example refinement and distribution drastically reduce the search space. As a result, together with several optimizations, MYTH can synthesize a variety of challenging data structure manipulation tasks.

Limitations. Nevertheless, the following limitations preclude even broader utility.

(A) To synthesize a recursive function, MYTH requires that input-output examples include those for any recursive calls internal to the solution. As Osera and Zdancewic [2015, §5.3] discuss, providing such trace complete examples [Polikarpova et al. 2016, §5] “proved to be difficult initially” and “discovering ways to get around this restriction ... would greatly help in converting this type-directed synthesis style into a usable tool.”

(B) The user cannot provide MYTH a sketch, neither to encode domain-specific insight to guide the search, nor to specify multiple, interdependent synthesis tasks. Instead, examples for each task must be specified separately, and each task is solved separately and completely.

Technically, these limitations originate from how, in the guess-and-check subroutine, MYTH checks whether a candidate expression \( e \) satisfies example constraints of the form \( E \vdash v \vdash ex \). The essence of the approach is shown below. In the environment \( E \), the CHECK rule first evaluates the expression to a value \( v \) and then checks whether \( v \) satisfies the example \( ex \). To check whether a function \( \text{fix } f (\lambda x. e) \) satisfies input-output examples \( \{ v_i \rightarrow ex_j \}_{i \in [n]} \), the CHECK-INPUT-OUTPUT rule evaluates the function on each input \( v_i \) and recursively checks satisfaction of the output \( v'_i \). This approach presents two obstacles. First, to test a function, CHECK-INPUT-OUTPUT replaces \( f \)

in the function body with the examples, to serve as a “lookup table” to evaluate recursive calls. Furthermore, the CHECK approach works only for complete programs, which produce complete values \( v \) to be checked. To eliminate the trace-completeness requirement (Limitation A) and to support sketching (Limitation B), an alternative approach is needed.

1.2 SKETCH-n-MYTH: Sketching with Example-Directed Synthesis

We present an approach for program synthesis by sketching, in which ordinary assert statements give rise to example-based constraints, which are solved iteratively using MYTH-style techniques to fill the holes.
**Live Bidirectional Evaluation.** The key innovation to combine sketching and Myth is a technique for checking example satisfaction, called *live bidirectional evaluation*, that comprises two parts:

1. A **live evaluator** \( e \Rightarrow r \) that partially evaluates a sketch by proceeding around holes, producing a result \( r \) which is either a value or an indeterminate expression that, when the necessary holes are filled, will continue evaluating safely (an approach borrowed from Omar et al. [2019]); and

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

Live bidirectional evaluation addresses both Limitations A and B. As shown below in [Live-Check], live bidirectional evaluation serves as a replacement for the simpler evaluate-and-check approach. Furthermore, because holes can appear in arbitrary expression positions (e.g. within the body of a recursive function being synthesized), our approach in [Live-Check-Input-Output] is simply to test each input-output pair via (live bidirectional) evaluation—recursive calls will generate additional constraints, rather than requiring them to be part of trace-complete examples from the user.

### Contributions

This paper presents new example-directed synthesis techniques that support sketching for richly-typed, functional programming languages.

Formally, we present a calculus of recursive functions, algebraic datatypes, and holes—called **CORE SKETCH-N-MYTH**—which includes the following technical contributions.

- We present **live unevaluation**, a technique that—together with **live evaluation** [Omar et al. 2019]—checks example satisfaction in the presence of sketches. Live bidirectional evaluation is, thus, core to the guess-and-check subroutine of example-based synthesis. (§3.5)

- We observe that live bidirectional evaluation can also be used to simplify ordinary program assertions into the kind of example constraints required by Myth. This allows examples to be provided indirectly by the flow of holes throughout evaluation, rather than directly (i.e. syntactically) on holes in the source code. (§4.1)

- We re-formulate Myth-style techniques to make appropriate use of sketches and live bidirectional evaluation, resulting in a synthesis algorithm that (a) alleviates the trace-completeness requirement of Myth and (b) iteratively solves multiple interdependent tasks. (§4.2)

We implement our calculus in a prototype system, called **SKETCH-N-MYTH**, and we evaluate the system with two empirical methods.

- We synthesize 37 benchmarks from the Myth benchmark suite, on average using 55% of the number of examples (without sketches) in **SKETCH-N-MYTH**—because examples need not be trace-complete. (§5.1)

- We identify three simple and common sketching strategies, and we apply them systematically to the Myth benchmarks. The results show that, when a strategy applies, it can reduce the number of examples required and/or the running time needed to synthesize a task. (§5.1)
Paper Outline. Next, in §2, we work through several simple examples in detail to provide an overview of our approach. We formally define live bidirectional evaluation in §3 and the synthesis pipeline in §4. We describe our implementation and experiments in §5, before concluding with a discussion of related and future work in §6 and §7. Appendix A contains additional definitions.

2 OVERVIEW

In this section, we walk through several simple programming tasks in Sketch-N-Myth. The specific user interface in our implementation is not a technical contribution or focus of this paper. Nevertheless, we describe user interactions in Sketch-N-Myth to provide a glimpse of how future “live” programming environments [Kubelka et al. 2018; McDirmid 2013; Tanimoto 2013] might employ the new synthesis techniques.

Our examples involve a small library of natural numbers, outlined below. In the exposition, we employ several syntactic conveniences not currently implemented in our prototype—differences are described in §5. Hole expressions ??_i are labeled with unique identifiers; in our implementation, these are automatically generated when omitted by the user. The literals 0, 1, 2, etc. in examples are syntactic sugar for the corresponding Nats.

\[
\text{type Nat} = \mathbb{Z} | S \text{Nat} \\
\text{plus, minus, mult, max : Nat} \to \text{Nat} \to \text{Nat}
\]

2.1 Example 1: Plus

We begin with a type signature and single input-output example for plus.

\[
\text{plus} : \text{Nat} \to \text{Nat} \to \text{Nat} \\
\text{plus m n} = ??_0 \\
\text{assert (plus 2 0 == 2)}
\]

When the program is evaluated, the assertion generates the example \((m \mapsto 2, n \mapsto 0) \vdash 0 \models 2\) for hole ??_0. “Direct” example constraints like this can be resolved using the synthesis techniques from Myth [Osera and Zdancewic 2015].

Type-Directed Guessing (à la Myth). We review the main ideas of Myth to provide a self-contained overview of their approach and our extensions. Next to the code listings below, we write “current” sets of constraints and solutions, if any, in comments adjacent to holes. We also show solutions (i.e. hole fillings) inside boxes, such as \([2 + n]\). Constraints and solutions are shown next to code only for exposition, to aid the discussion; they are not part of the user interface in Sketch-N-Myth. During the walkthrough, new constraints are labeled with asterisks and existing constraints are shown in gray (even when just giving existing constraints new names).

In the first “stage” of synthesis, Sketch-N-Myth searches for “small” expressions—literals, variables, and applications of data constructors to small expressions. For the single constraint on hole ??_0, three such solutions are found: \(m\) and \(S S n\) (i.e. \(2 + n\)) and \(S S Z\) (i.e. \(2\)).

\[
\text{plus m n} = \\
??_0 -- (m \mapsto 2, n \mapsto 0) \vdash 0 \models 2 \\
-- [m | S S n | S S Z]
\]

Displaying Solutions. Sketch-N-Myth displays these solutions in a pop-up menu (not shown); the user hovers solutions in the menu to preview the resulting changes in the code editor (also not shown) and clicks to choose one.

By default, Sketch-N-Myth shows (up to) three non-recursive solutions and (up to) three recursive solutions, ranked according to simple heuristics. These (up to) six options are referred to as
the solution window. The user can also ask to see more solutions, if any, found during the current synthesis stage.

**Examples via Partially Evaluated Assertions.** Not surprisingly—given that we provided only one input-output example—none of these expressions implement the desired functionality of `plus`. So, we decide to add two more input-output examples, aiming to "cover" the corner cases: `assert (plus 0 0 = 0)` and `assert (plus 0 1 = 1)`.

Rather than with three separate `assert` statements, we decide to specify the examples by calling the library function `specifyFunction2` (lines 9-10), which takes a binary function `f` and list of triples `(x,y,z)—each a test case with two inputs and the expected output—and generates the list of assertions `assert (f x y == z)`. Notice that these input-output examples are no longer syntactically apparent in the source program, but rather emerge by partially evaluating the sketch.

Using the live programming with holes technique proposed by Omar et al. [2019], SKETCH-N-MYTH evaluates sketches by proceeding "around" hole expressions that reach evaluation position, allowing other expressions not dependent on them to evaluate further. Each time a hole `??_h` reaches evaluation position, the resulting hole closure of the form `[E] ??_h` records the environment `E`. (In contrast, in languages without direct support for holes, the typical workaround `raise /quotedbl.VarNot implemented/quotedbl.Var` as a placeholder for the body of `plus` would terminate evaluation as soon as the function is called for the first time.)

The result of the call to `specifyFunction2` in this case produces two new example constraints, shown on lines 14-15. Small expressions are not sufficient to satisfy all three constraints. Therefore, SKETCH-N-MYTH begins the next stage of synthesis, guessing (small) expressions on which to branch.

```
9 specifyFunction2 plus
10 [ (2, 0, 2), (0, 0, 0), (0, 1, 1) ]
11
12 plus m n =
13 ??_0 -- (m -> 2, n -> 0) |- _0 | 2
14     -- (m -> 0, n -> 0) |- _0 | 0
15     -- (m -> 0, n -> 1) |- _0 | 0
16
17 Example-Directed Branching (à la MYTH). Consider the search path when the argument `m` is guessed as the scrutinee. The new working sketch, below, contains two subgoals—hole expressions `??_1` and `??_2` on the Z and S branches, respectively.

18 plus m n =
19     case m of
20       Z ->
21         ??_1 -- (m -> 0, n -> 0) |- _1 | 0
22         -- (m -> 0, n -> 1) |- _1 | 1
23         -- _1
24       S m' ->
25         ??_2 -- (m -> 2, n -> 0, m' -> 1) |- _2 | 2
26         -- _2 | m S m' | S S n | S S Z
```

To constrain the two new subgoals, the three constraints (0.1) through (0.3) for the previous goal, `??_0`, are distributed across the subgoals, depending on how the scrutinee `m` evaluates in each world environment. The first subgoal is defined by constraints (0.2) and (0.3), because those world environments bind `m` to 0. The variable `n` constitutes the only small solution to these two constraints.
The second subgoal is defined by constraint (0.1), where the environment binds \( m \) to \( 2 \)—the environment for the new constraint (2.1) subgoal is extended to bind the pattern variable \( m' \) to \( 1 \). The three expressions from before—\( m \) \( S \) \( n \) and \( S \) \( S \) \( Z \)—are still valid solutions; now \( S \) \( m' \) also evaluates to \( 2 \).

In addition to the four solutions above—the one solution for \( ??_1 \) paired with one of the four for \( ??_2 \)—there are analogous solutions for a search path that chooses to branch on \( n \), rather than \( m \). These solutions do not implement the desired functionality of \( \texttt{plus} \), however; none is even recursive.

To further specify the task, we add a fourth input-output example (\( \texttt{plus 1 2 == 3} \)) which gets evaluated and distributed to the \( S \) branch.

\[
\begin{align*}
\text{specifyFunction2 plus} & \\
[ (2, 0, 2), (0, 0, 0), (0, 1, 1), (1, 2, 3) ] & \\
\text{plus m n =} & \\
\text{case m of} & \\
S m' \rightarrow & \\
?_2 & (m \mapsto 2, n \mapsto 0, m' \mapsto 1) + \bullet_2 \models 2 \quad \text{(Constraint 2.1)} \tag{2a.1} \\
& (m \mapsto 1, n \mapsto 2, m' \mapsto 0) + \bullet_2 \models 3 \quad \text{(Constraint 2a.2)} \\
S ?_2b & (m \mapsto 2, n \mapsto 0, m' \mapsto 1) + \bullet_{2b} \models 1 \quad \text{(Constraint 2b.1)*} \tag{2b.1} \\
& (m \mapsto 1, n \mapsto 2, m' \mapsto 0) + \bullet_{2b} \models 2 \quad \text{(Constraint 2b.2)*} \\
S (S ?_2c) & (m \mapsto 2, n \mapsto 0, m' \mapsto 1) + \bullet_{2c} \models 0 \quad \text{(Constraint 2c.1)*} \tag{2c.1} \\
& (m \mapsto 1, n \mapsto 2, m' \mapsto 0) + \bullet_{2c} \models 1 \quad \text{(Constraint 2c.2)*} \\
\end{align*}
\]

These \textit{refinement trees} are a central optimization in \textit{Myth}. The commonalities between examples—shared constructors for data type goals, lambda-abstraction for function type goals—are eagerly synthesized in the program. Type-directed guessing—which does not use example constraints to prune the search space of well-typed terms—can thus begin at deeper levels of the tree, where desired completions would be smaller than at nodes higher in the refinement tree.

None of the prior four solutions (line 24) satisfy the new constraint (2a.2) constraint at the \( ?_2a \) node in the tree, nor are there any other small expressions to consider. Furthermore, there are no small expressions that satisfy the constraints for nodes \( S ?_2b \) and \( S (S ?_2c) \) in...
the tree. It is time for the next stage of synthesis, where Sketch-N-Myth starts guessing larger (non-constructor) application expressions.

**Synthesizing Recursive Functions Without Trace Complete Examples.** When describing the environments above, we did not pay attention to the fact that plus—the function Sketch-N-Myth is working to synthesize—is recursive and part of the environments in example constraints. Thus, recursive calls to plus may be guessed.

Consider the branch of search where Sketch-N-Myth is working on the second node in the refinement tree (lines 41-42). In this case, the name plus binds the following:

\[
\text{plus} \mapsto \text{fix plus } (\lambda \ m \ n \to \text{case } m \text{ of } \{ Z \to ??_1; S m' \to S ??_2b \})
\]

Suppose that subgoal ??_2b (as opposed to ??_1) is chosen next. Sketch-N-Myth starts guessing well-typed, structurally-decreasing, and non-nested applications to plus—namely, plus m' n, plus m n', and plus m' n'.

Consider the guess plus m' n, in which case plus binds the following; notice how the “current guess” fills the second branch:

\[
\text{plus} \mapsto \text{fix plus } (\lambda \ m \ n \to \text{case } m \text{ of } \{ Z \to ??_1; S m' \to S \text{ plus } m' n \})
\]

Simply (live) evaluating and (live bidirectionally) checking the four test cases on line 26 produces the constraints on the base case, ??_1, below. This goal can be completed by n, thus providing the following complete solution for plus.

\[
\text{plus } m \ n = \begin{cases} 
& \text{case } m \text{ of } \\
& Z \to \\
& \begin{cases} 
& ??_1 \quad \text{(Constraint 1.1)} \\
& \begin{cases} 
& (m \mapsto 0, n \mapsto 0) \vdash \bullet_1 \models 0 \\
& (m \mapsto 0, n \mapsto 1) \vdash \bullet_1 \models 1 \\
& (m \mapsto 0, n \mapsto 2) \vdash \bullet_1 \models 2. \\
& n
\end{cases} \\
& \begin{cases} 
& \text{plus } m' n
\end{cases}
\end{cases}
\]

Notice that the four tests exercise the main four cases of plus—two zero arguments, two non-zero arguments, and one of each—but they are not trace-complete—we did not provide examples for plus 1 0 or plus 0 2. Instead, when evaluating the first test, plus 2 0 == 2, internally plus 1 0 was evaluated internally and led to constraint (1.1)—also generated by the second test. And when evaluating the fourth test, plus 1 2 == 3, internally plus 0 2 was evaluated and led to the new constraint (1.3). Thus, live bidirectional evaluation allows Sketch-N-Myth to generate additional constraints that would otherwise have to be provided by the user.

(When considering other guesses for the subgoal ??_2b—and when considering the nodes ??_2a and ??_2c in the refinement tree—the resulting constraints do not lead to a solution. Thus, the solution above is the only recursive solution offered in the solution window by Sketch-N-Myth.)

### 2.2 Example 2: Max

The previous example demonstrated how, given a goal type and examples, Sketch-N-Myth (i) partially evaluates a program, using (partially evaluated) assertions to derive example constraints for synthesis, and (ii) uses live bidirectional evaluation when guessing-and-checking to avoid requiring trace-complete examples. Next, we describe an example where the user provides a sketch, rather asking Sketch-N-Myth to synthesize a task entirely.
User-Defined Sketches. The max function, to return the maximum of two naturals, happens to be a task apparently difficult for the example-based search techniques, requiring 8 examples in Sketch-n-Myth. 5 of these 8 examples include zero as one or both arguments.

Instead, if the user sketches the easy, zero cases for max, as shown below, just a few input-output examples are sufficient to complete the task (i.e. the recursive case).

\[
\begin{align*}
\text{max} & : \text{Nat} \to \text{Nat} \\
\text{max} \; m \; n = & \\
\text{case} \; (m, \; n) \; \text{of} & \\
\quad (Z, \; _) & \to \; n \\
\quad (_, \; Z) & \to \; m \\
\quad (S \; m', \; S \; n') & \to \; ?? \quad -- \; S \; (\text{max} \; m' \; n')
\end{align*}
\]

specifyFunction2 max
\[
\begin{bmatrix}
(1, 1, 1),
(1, 2, 2),
(3, 1, 3)
\end{bmatrix}
\]

User-provided sketches are handled in the same way as the internally-created sketches described above; they can be partially evaluated and used to check satisfaction of still-incomplete solutions. The max sketch demonstrates a programming task in which domain knowledge from the user can be split naturally across a partial implementation and input-output examples.

2.3 Example 3: Minus

Next, we describe a task where the structure of partially evaluated assertions are more complex than in the previous examples.

To implement minus, we have the idea to iteratively subtract from a until b becomes 0. We provide an initial sketch and a couple input-output examples. For readability, we choose hole names that correspond to the desired completions.

\[
\begin{align*}
\text{minus} & : \text{Nat} \to \text{Nat} \\
\text{minus} \; a \; b = & \\
\text{case} \; (a, \; b) \; \text{of} & \\
\quad (S \; a', \; S \; b') & \to \; \text{minus} \; ??_a' \; ??_b' \\
\quad _ & \to \; ??_a \\
\quad & \quad -- \; (\text{minus} \mapsto \ldots, \; a \mapsto 2, \; b \mapsto 0) \vdash \bullet_1 \models 2 \\
\quad & \quad -- \; a \quad S \quad S \quad b \quad S \quad S \quad Z
\end{align*}
\]

specifyFunction2 minus
\[
\begin{bmatrix}
(2, 0, 2),
(3, 2, 1)
\end{bmatrix}
\]

The assertion \( \text{minus} \; 2 \; 0 \; \models 2 \) produces a constraint and corresponding solution (lines 52 and 53) much like we have already discussed. Evaluating the assertion \( \text{minus} \; 3 \; 2 \; \models 1 \), however, presents a new challenge: the partially evaluated expression to be constrained is not directly a hole closure.

Indeterminate Results. Figure 1 shows several evaluation steps that stem from the function call \( \text{minus} \; 3 \; 2 \). Starting in the initial environment \( E \) that binds the recursive function itself, the first several steps proceed as usual.

First, the arguments 3 and 2 are evaluated, bound to \( a \) and \( b \) in \( E_{32} \), the environment used to evaluate the body of \( \text{minus} \) (the subscript "32" is chosen to indicate the value bindings of \( a \) and \( b \)). Second, within the function body, the variables \( a \) and \( b \) are resolved to 3 and 2. Third, these values are scrutinized and determined to match the pattern for the first branch; the bindings 2 and 1 for \( a' \) and \( b' \), respectively, appear in the extended environment \( E'_{32} \).


\[
E \overset{\text{def}}{=} \text{minus} \mapsto \ldots\\
E_{32} \overset{\text{def}}{=} E, a \mapsto 3, b \mapsto 2\\
E'_32 \overset{\text{def}}{=} E_{32}, a' \mapsto 2, b' \mapsto 1
\]

1: Call function
2: Lookup variables
3: Evaluate first branch
4: Close over holes
5: Call function
6: Lookup variables
7: Close over "indeterminate" case

\[
E''_{32} \overset{\text{def}}{=} E, a \mapsto [E'_{32}] ??_a', b \mapsto [E'_{32}] ??_b'
\]

\[
r_{32} \overset{\text{def}}{=} [E''_{32}] \text{case } ([E'_{32}] ??_a', [E'_{32}] ??_b') \text{ of } \ldots
\]

Fig. 1. Evaluation sequence for `minus 3 2` that produces an *indeterminate* result.

Fourth is to evaluate the recursive call in the first branch (line 50). Here we reach two hole expressions `??_a'` and `??_b'`, each closed by the current environment \(E'_{32}\). Fifth, inside the function body again, these hole closures are bound to \(a\) and \(b\) in the environment \(E''_{32}\). The sixth step immediately retrieves these two values and discriminates them. But hole closures are *indeterminate*—their outer value structure has not yet been determined—so evaluate choose an appropriate branch. Thus, the seventh step closes over the case expression producing an indeterminate case expression \(r_{32}\), which can be resumed when the hole closures resolve to *determinate* forms—in this case, applications of `Nat` constructors.

**Indirect Example Constraints.** The result \(r_{32}\) cannot proceed, yet the run-time assertion declares that it produce the value 1. Our key observation is such evaluation requirements `\(r \Rightarrow \nu.alt\)` can be used to "indirectly" constrain holes, specifically those that have blocked the result from evaluating to a determinate form. In particular, we can reuse the notion of live bidirectional evaluation—so far used to define a notion of guessing-and-checking for sketches—to turn the assertion `\(r_{32} \Rightarrow 1\)` into ordinary example constraints—over possibly many other holes in the program—that would ensure that \(r_{32}\) would indeed resume evaluating to the value 1.

For the indeterminate case like this, first we need to evaluate the case to determine which branch to take. The task, to find an appropriate well-typed expression to scrutinize, is the same as type-directed guessing above. Once a scrutinee has been chosen, live evaluation reveals the kind of data constructor it produces, at which point we recursively live unevaluate the branch expression such that it produces the desired value \(\nu\).

For this sketch and these two examples, `SKETCH-N-MYTH` produces two solutions:

\[
\text{minus } a \ b = \text{case } (a, b) \text{ of } \{(S \ a', S \ b') \rightarrow \text{minus } b' \ 0; _ \rightarrow \ a\}
\]

\[
\text{minus } a \ b = \text{case } (a, b) \text{ of } \{(S \ a', S \ b') \rightarrow \text{minus } 1 \ 0; _ \rightarrow \ a\}
\]

These solutions are over-specialized, so we add a third input-output example below. Given this, `SKETCH-N-MYTH` produces the desired implementation as the only option.

```
specifyFunction2 minus
  [ (2, 0, 2), (3, 2, 1), (3, 1, 2) ]
```

\[
\text{minus } a \ b = \text{case } (a, b) \text{ of } \{(S \ a', S \ b') \rightarrow \text{minus } a' \ b'; _ \rightarrow \ a\}
\]
2.4 Example 4: Mult
To round out our small library of naturals, we sketch a recursive solution to mult in terms of plus (using three holes as arguments). Given this specification, SKETCH-N-MYTH synthesizes the appropriate missing arguments to plus and mult. As with minus, live bidirectional evaluation untangles the interplay between indeterminate branching in plus and mult.

\[
\text{mult} : \text{Nat} \to \text{Nat} \to \text{Nat}
\]
\[
\text{mult} \ p \ q =
\text{case} \ p \ \text{of}
\quad \text{Z} \to \text{Z}
\quad \text{S} \ p' \to \text{plus} \ ?? \text{ ?} \text{ ?} \text{ ?} \to \text{plus} \ q \text{ ?} \text{ ?} \text{ ?} \text{ ?}
\]

\[
\text{specifyFunction2} \ \text{mult}
[ \ (1, 1, 1), (1, 2, 2), (2, 1, 2) ]
\]

2.5 Example 5: Stutter N
As a final example, consider a stutter function that duplicates every element in a list [Osera and Zdancewic 2015, §1], built either manually or using synthesis.

\[
\text{stutter} : \text{List} \to \text{List}
\]
\[
\text{stutter} \ x =
\text{case} \ x \ \text{of}
\quad [] \to []
\quad x::x' \to x::x::\text{stutter} \ x'
\]

Suppose we want to generalize this into a function stutter\textsubscript{n}, that will stutter its arguments n times—relying on an unimplemented helper function replicate to process each element. Given three input-output examples for stutter\textsubscript{n}, SKETCH-N-MYTH synthesizes a correct implementation for the replicate function.

\[
\text{replicate} : \text{Nat} \to \text{Nat} \to \text{List}
\]
\[
\text{replicate} \ n \ x =
?? \to \quad \text{case} \ n \ \text{of}
\quad \text{Z} \to []
\quad \text{S} \ n' \to x::\text{replicate} \ n' \ x
\]

\[
\text{stutter\textsubscript{n}} : \text{Nat} \to \text{List} \to \text{List}
\]
\[
\text{stutter\textsubscript{n}} \ n \ x =
\text{case} \ x \ \text{of}
\quad [[] \to []
\quad x::x' \to \text{replicate} \ n \ x ++ \text{stutter\textsubscript{n}} \ n \ x'
\]

\[
\text{specifyFunction2} \ \text{stutter\textsubscript{n}}
[ \ (0, [1], []),
  (1, [1], [1]),
  (2, [1], [1, 1]) ]
\]

Recap. To recap, the live programming with synthesis workflow in SKETCH-N-MYTH consists of (1) partially evaluating a sketch, and then using live bidirectional evaluation to convert (partially evaluated) assertions into example constraints that can be handled by MYTH-style synthesis; and (2) MYTH-style synthesis extended with live bidirectional evaluation for guessing-and-checking candidate expressions.
Program Synthesis with Live Bidirectional Evaluation

| Types   | $T ::= T_1 \to T_2 \mid (\) \mid (T_1, T_2) \mid D$ |
|---------|--------------------------------------------------|
| Expressions | $e ::= \text{fix } (\lambda x. e) \mid e_1 \cdot e_2 \mid x$ |
|          | $\mid (\) \mid (e_1, e_2) \mid \text{proj } i \subseteq 2 \cdot e$ |
|          | $\mid C \cdot e \mid \text{case } e \text{ of } \{ C_i x_i \to e_i \}_{i \in [n]}$ |
|          | $\mid ???_h$ |

| Datatypes | $D$ |
| Variables | $f, x$ |
| Constructors | $C$ |
| Hole Names | $h$ |

“Determinate” Results

$\text{Results } r ::= [E] \text{fix } (\lambda x. e) \mid (\) \mid (r_1, r_2) \mid C \cdot r$

“Indeterminate” Results

$\mid [E] ???_h \mid r_1 r_2 \mid \text{proj } i \subseteq 2 \cdot r \mid [E] \text{case } r \text{ of } \{ C_i x_i \to e_i \}_{i \in [n]}$

| Environments | $E ::= - \mid E, x \mapsto r$ |
| Type Contexts | $\Gamma ::= - \mid \Gamma, x : T$ |
| Datatype Contexts | $\Sigma ::= - \mid \Sigma, \text{type } D = \{ C_i : T_i \to D \}_{i \in [n]}$ |
| Hole Type Contexts | $\Delta ::= - \mid \Delta, h \mapsto (\Gamma \mapsto \bullet : T)$ |
| Synthesis Goals | $G ::= \{ (\Gamma_i \mapsto \bullet_i, : T_i \equiv X_i) \}_{i \in [n]}$ |

| Example Constraints | $X ::= \{ (E_i \mapsto \bullet \equiv e) \}_{i \in [n]}$ |
| Examples | $ex ::= (\) \mid (ex_1, ex_2) \mid C \cdot ex \mid \{ v \mapsto ex \} \mid T$ |
| Simple Values | $v ::= (\) \mid (v_1, v_2) \mid C \cdot v$ |
| Uneval Constraints | $K ::= (U ; F)$ |
| Unfilled Holes | $U ::= - \mid U, h \mapsto X$ |

Fig. 2. Syntax of Core Sketch-N-Myth.

3 LIVE BIDIRECTIONAL EVALUATION WITH EXAMPLES

Having worked through an overview of the Sketch-N-Myth system, next we formally define

\[
\text{live evaluation } E ; F \vdash e \Rightarrow r \quad \text{and} \quad \text{live unevaluation } F \vdash r \Leftarrow ex + K
\]

for a calculus called Core Sketch-N-Myth. 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 the notation for live bidirectional evaluation introduced in §1, here we refer to environments $E$ and $F$—often typeset in light gray font to emphasize that environments would “fade away” in a substitution-style presentation.

Our formulation proceeds in several steps. First, in §3.1 and §3.2, we define the syntax and type checking judgements of Core Sketch-N-Myth. Next, in §3.3, we present live evaluation, which borrows the live programming with holes technique from Hazelnut Live [Omar et al. 2019]; to keep the presentation self-contained, we defer comparisons to Hazelnut Live to §6. Lastly, novel to our work, 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 2 defines the syntax of Core Sketch-N-Myth, a calculus of recursive functions, unit, pairs, and (named, recursive) algebraic datatypes. We say “products” to refer collectively to unit and pairs.

Datatypes. We assume a fixed datatype environment $\Sigma$, where each datatype $D$ is defined with some number $n$ of constructors $C_i$, each of which carries a single argument of type $T_i$—that is,
Final Results and Environments

\[ \begin{array}{c|c|c|c|c} \hline r_{\text{det}} & r_{\text{indet}} & E_{\text{final}} & r_{\text{final}} \\ \hline r_{\text{final}} & r_{\text{final}} & \text{---final} & E, x \mapsto r_{\text{final}} \\ \hline \end{array} \]

Determinate Results

\[ \begin{array}{c|c|c} \hline E_{\text{final}} & \{r_i \text{ final}\}^{i \in [2]} & r_{\text{final}} \\ \hline [E] \text{fix } f(\lambda x. e)_{\text{det}} & ()_{\text{det}} & (r_1, r_2)_{\text{det}} \\ \hline \end{array} \]

Indeterminate Results

\[ \begin{array}{c|c|c|c} \hline E_{\text{final}} & r_1_{\text{indet}} & r_2_{\text{indet}} & E_{\text{final}} \\ \hline [E] ??_h_{\text{indet}} & r_1_{\text{indet}} & r_2_{\text{indet}} & \text{prj}_{i \in [2]}_{\text{indet}} \\ \hline \end{array} \]

Fig. 3. Result Classification. Final results are determinate or indeterminate.

the type of \( C_i \) is \( T_i \rightarrow D \). Rather than supporting arbitrary-arity constructors—as in the technical formulation of Osera and Zdancewic [2015]—we choose single-arity constructors and products—following the formulation by Frankle [2015]—to lighten the presentation of synthesis in §4.

Expressions and Holes. The expression forms on the first three lines are standard functions, product, and constructor forms, respectively. The expressions \( \text{prj}_{i \in [2]} e \) and \( \text{prj}_{i \in [2]} e \) project the first and second components of a pair. We require that each case expression has one branch for each of the \( n \) constructors \( C_i \) corresponding to the type of the scrutinee \( e \). Our formulation does not support nested patterns for simplicity.

Hole expressions ?? can appear arbitrarily within expressions—i.e. expressions are sketches [Solar-Lezama 2008]. We assume that each hole in a sketch has a unique name \( h \). We sometimes write ?? when the name is not referred to. For each hole ?? in a sketch, hole environments \( \Delta \) defines the contextual type \((\Gamma \vdash \bullet : T)\) to describes what expressions can “fill” the hole [Nanevski et al. 2008].

Results. CORE SKETCH-N-MYTH includes a separate grammar of evaluation 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—one on their own—correspond to what would be the values in a conventional natural semantics (without holes). In CORE SKETCH-N-MYTH, these are determinate results that can be eliminated in a type-appropriate position—Figure 3 defines the predicate \( r_{\text{det}} \) to identify such results, and type checking is discussed below. Note that a recursive function closure \([E] \text{fix } f(\lambda x. e)\) 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 result forms on the second line are unique to the presence of holes. These results are indeterminate, because a hole has reached elimination position. We sometimes refer to indeterminate results as “partially evaluated expressions.” Figure 3 defines the predicate \( r_{\text{indet}} \) to identify such results. 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. \( r_1_{\text{indet}} \)); we require
Expression Typing

\[ \Sigma; \Delta; \Gamma \vdash e : T \]

[T-Fix]
\[ \Sigma; \Delta; \Gamma, f : T_1 \to T_2, x : T_1 \vdash e : T_2 \]
\[ \Sigma; \Delta; \Gamma \vdash f \ast f (\lambda x. e) : T_1 \to T_2 \]

[T-Var]
\[ \Gamma(x) = T \]
\[ \Sigma; \Delta; \Gamma \vdash x : T \]

[T-Hole]
\[ \Delta(??_h) = (\Gamma \vdash \bullet : T) \]
\[ \Sigma; \Delta; \Gamma \vdash ??_h : T \]

[T-Unit]
\[ \Sigma; \Delta; \Gamma \vdash () : () \]
\[ \{ \Sigma; \Delta; \Gamma \vdash e_i : T_i \} \ i \in [2] \]
\[ \Sigma; \Delta; \Gamma \vdash (e_1, e_2) : (T_1, T_2) \]

[T-Pair]
\[ \Sigma; \Delta; \Gamma \vdash e_1 : T_2 \to T \]
\[ \Sigma; \Delta; \Gamma \vdash e_2 : T_2 \]
\[ \Sigma; \Delta; \Gamma \vdash e_1 \cdot e_2 : T \]

[T-App]
\[ \Sigma; \Delta; \Gamma \vdash e : T \]
\[ \xi \in [2] \]
\[ \Sigma; \Delta; \Gamma \vdash \text{prj}_{i \in [2]} e : T_i \]
\[ \Sigma; \Delta; \Gamma \vdash \text{case } e \text{ of } \{ C_i x_i \to e_i \} \ i \in [n] : T \]

Result and Example Typing (Figure 15 of Appendix A)

\[ \Sigma; \Delta \vdash r : T \]
\[ \Sigma; \Delta \vdash \text{ex} : T \]

Fig. 4. Type Checking.

that \( r_2 \) be final in accordance with our eager evaluation semantics, discussed below. An indeterminate projection \( \text{pr}_j \ i \in [2] \) \( r \) appears when the argument has not yet evaluated to a pair (i.e. \( r \ \text{indet} \)). An indeterminate case closure \( E \) case \( r \) 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. \( r \ \text{indet} \))—like with function and hole closures, the environment \( E \) is used when evaluation resumes with the appropriate branch.

The inverse constructor application form \( C \ \text{r} \) on the third line is used internally by live unevaluation and is discussed in §3.5.

Examples. A synthesis goal \( (\Gamma \vdash \bullet : T \vdash X) \) describes a hole \( ??_h \) to be filled in accordance with the contextual type \( (\Gamma \vdash \bullet : T) \) and example constraints \( X \). As described in §1, each example constraint \( (E \vdash \bullet \vdash \text{ex}) \) requires that any expression to fill the hole must, in environment \( E \), satisfy example \( \text{ex} \). Examples include simple values \( v \), which are first-order product values or constructor applications; input-output examples \( \{ v \to \text{ex} \} \), 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.

We define three simple functions below. The coercion \( [r] = v \) (the three rules below on the top-left) “downcasts” a result to a simple value. The coercion \( [v] = r \) (top-right) “upcasts” a simple value to a result. The \( \text{Filter}(X) \) function (bottom) removes constraints involving the top example.

\[
\begin{align*}
\text{Filter}(X) &= \{ (E \vdash \bullet \vdash \text{ex}) \in X \mid \text{ex} \neq \top \}
\end{align*}
\]

3.2 Type Checking

Figure 4 defines type checking for CORE SKETCH-N-MYTH expressions. Type checking takes as input a hole environment \( \Delta \), used by the T-Hole rule to decide valid typings for a hole \( ??_h \). The remaining typing rules are standard.
**Live Evaluation**

\[ E \vdash e \Rightarrow r \quad F \vdash r \Rightarrow r' \]

\[ E ; F \vdash e \Rightarrow r' \]

**Expression Evaluation**

\[ E \vdash e \Rightarrow r \]

\[ [E/hyphen.scP/r.sc/j.sc/hyphen.scI/n.sc/d.sc/e.sc/t.sc] \]

\[ \frac{\} }{E/hyphen.scA/p.sc/p.sc/hyphen.scI/n.sc/d.sc/e.sc/t.sc} \]

\[ \frac{\} }{E/hyphen.scC/a.sc/s.sc/e.sc/hyphen.scI/n.sc/d.sc/e.sc/t.sc} \]

\[ \frac{\} }{E/hyphen.scC/t.sc/o.sc/r.sc} \]

\[ \frac{\} }{E/hyphen.scV/a.sc/r.sc} \]

\[ \frac{\} }{E/hyphen.scH/o.sc/l.sc/e.sc/hyphen.scI/n.sc/d.sc/e.sc/t.sc} \]

\[ \frac{\} }{E/hyphen.scP/a.sc/i.sc/r.sc} \]

\[ \frac{\} }{E/hyphen.scP/r.sc/j.sc/hyphen.scI/n.sc/d.sc/e.sc/t.sc} \]

\[ \frac{\} }{E/hyphen.scC/a.sc/s.sc/e.sc/hyphen.scI/n.sc/d.sc/e.sc/t.sc} \]

\[ \frac{\} }{E/hyphen.scC/t.sc/o.sc/r.sc} \]

\[ \frac{\} }{E/hyphen.scV/a.sc/r.sc} \]

\[ \frac{\} }{E/hyphen.scH/o.sc/l.sc/e.sc/hyphen.scI/n.sc/d.sc/e.sc/t.sc} \]

**Resumption (excerpt from Figure 18 of Appendix A)**

\[ F \vdash r \Rightarrow r' \quad F \vdash E \Rightarrow E' \]

\[ F \vdash [E] ??_h \Rightarrow r' \]

\[ F \vdash [E] ??_h \Rightarrow [E'] ??_h \]

Fig. 5. Evaluation and Resumption.

Figure 4 also shows corresponding type checking judgements for results and examples. These do not require a type environment \( \Gamma \), because results and expressions do not contain free variables. Result typing refers to expression typing because function closures and case closures contain expressions and evaluation environments. Appendix A provides the complete definitions.

### 3.3 Live Evaluation

Figure 5 defines live evaluation \( E ; F \vdash e \Rightarrow r \) for Core Sketch-N-Myth. Live evaluation first uses expression evaluation \( E \vdash e \Rightarrow r \) to produce a final result \( r \), and then uses a hole-filling \( F \) to resume 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, E-CASE-INDET—one for each of the indeterminate result forms,
Example Satisfaction (of Expressions)

\[
\begin{align*}
\text{Sat} & : \{ E_i : F \vdash e \Rightarrow r_i \quad F \vdash r_i \models e x_i \} \mapsto \{ x \} \\
F \vdash e \models \{ (E_i \vdash \bullet \models e x_i) \} \mapsto \{ x \}
\end{align*}
\]

Example Satisfaction (of Results)

\[
\begin{align*}
\text{XS-Top} & : F \vdash r \models \top \\
\text{XS-Unt} & : F \vdash () \models () \\
\text{XS-Pair} & : \{ F \vdash r_i \models e x_i \} \mapsto \{ e x \} \\
\text{XS-CTop} & : F \vdash r \models e x \\
\text{XS-Input-Output} & : F \vdash r_1 \{ v_2 \} \Rightarrow r \\
\end{align*}
\]

Fig. 6. Example Syntax, Typing, and Satisfaction.

described above. 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 only type-compatible determinate result form—the E-APP-INDET rule creates the indeterminate application result \( r_1 \; r_2 \). The E-PRJ-INDET and E-CASE-INDET rules are similar. Evaluation is deterministic and produces final results.

**Proposition 3.1 (Determinism of Evaluation).** If \( E \vdash e \Rightarrow r \) and \( E \vdash e \Rightarrow r’ \), then \( r = r’ \).

**Proposition 3.2 (Finality of Evaluation).** If \( E \vdash e \Rightarrow r \), then \( r \text{ final} \).

**Resumption.** Figure 5 defines \( F \vdash r \Rightarrow r’ \) to resume evaluation of indeterminate results \( r \), if and when one or more expression holes in the program are filled by the hole-filling \( F \). Resumption does not require an evaluation environment \( E \), because results do not contain free variables.

Resumption recursively evaluates results much like expression evaluation. The hole-filling \( F \) is used to evaluate hole closures \([E] ??_h \) that reached evaluation position during expression evaluation. If \( F \) does not fill the hole \( ??_h \), the R-HOLE-INDET rule returns the hole closure unchanged. If \( F \) does fill the hole, the R-HOLE-RESUME rule evaluates the filled expression \( e_i \) in the closure environment, producing a result \( r \). A filled expression \( e_i \) may refer to new hole expressions—filled by \( F \)—so R-HOLE-RESUME recursively resumes \( r \). Note that a filled expression \( e_i \) will be evaluated for each of the corresponding hole closures in the result.

Like evaluation, resumption is deterministic and produces final results. Resumption evaluates results “as far as possible” given a hole-filling.

**Proposition 3.3 (Determinism of Resumption).** If \( F \vdash r \Rightarrow r \) and \( F \vdash r \Rightarrow r’ \), then \( r = r’ \).

**Proposition 3.4 (Finality of Resumption).** If \( F \vdash r \Rightarrow r’ \), then \( r’ \text{ final} \).

**Proposition 3.5 (Idempotency of Resumption).** If \( F \vdash r_0 \Rightarrow r \), then \( F \vdash r \Rightarrow r \).

**Soundness.** Type checking and evaluation are related by type soundness properties defined in Appendix A. Formulating a progress property in a big-step semantics is complicated by the fact that non-terminating computations are not necessarily distinguished from stuck ones [Leroy and Grall 2009]. Using a technique similar to that described by Ancona [2014], we augment evaluation with a natural number \( k \) that limits the depth of an evaluation derivation.
3.4 Example Satisfaction
Having defined how to partially evaluate a sketch to a result in Core Sketch-N-Myth, in Figure 6 we define 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 (live) 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 at the leaves (XS-UNIT, XS-PAIR, and XS-CTOR). Although hole closures may appear in a satisfying result, they may not be directly checked, i.e., discriminated against a product, constructor, or input-output example. The purpose of live unevaluation is to provide an algorithmic notion of example consistency to accompany this “ground-truth” notion of example satisfaction.

3.5 Live Unevaluation

Live unevaluation $F \vdash r \iff ex \vdash K$ (Figure 7) produces constraints $K$ over holes in the program that are sufficient to ensure example satisfaction $F \vdash r \models ex$. Live bidirectional example checking $F \vdash e \iff X \vdash K$ (Figure 7) lifts this notion to expressions and sets of examples: the Live-Check rule appeals to live evaluation followed by live unevaluation. The following properties characterize the relationship between live bidirectional evaluation and example satisfaction.

**Proposition 3.6 (Soundness of Live Unevaluation).**

If $F \vdash r \iff ex \vdash K$ and $F \models K$, then $F \vdash r \models ex$.

**Proposition 3.7 (Soundness of Live Bidirectional Example Checking).**

If $F \vdash e \iff X \vdash K$ and $F \models K$, then $F \vdash e \models X$.

**Unevaluation Constraints.** Two kinds of constraints $K$ are generated by unevaluation (cf. Figure 2). 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 as an optimization for implementation. The former are “hole example contexts,” analogous to hole type contexts $\Delta$; the metavariable $U$ serves as a mnemonic for holes left unfilled by a hole-filling $F$.

Figure 7 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 $??_{hi}$ constrained by $K$.

Figure 7 also shows the signature of two constraint merge operators. The “syntactic” merge operation $K_1 \oplus K_2$ pairwise combines fillings $F$ and example contexts $U$ in a straightforward way. Syntactically merged constraints may describe holes $??_h$ both with fillings in $F$ and example constraints $X$ in $U$; the “semantic” operation $\text{Merge}(F)$ uses live bidirectional example checking to check consistency in such situations. The full definitions can be found in Appendix A.

**Simple Unevaluation Rules.** Analogous to the five example satisfaction rules (prefixed “XS-” in Figure 6) are the U-Top rule to unevaluate top and the U-UNIT, U-PAIR, U-CTOR, and U-Fix rules to unevaluate examples against determinate results. (U-Fix is the complete version of the Live-Check-INPUT-OUTPUT rule sketched in §1.)

The base case in which unevaluation generates constraints is when the result $r$ is a hole closure $[E] ??_{hi}$, for which the U-Hole rule generates the (named) example constraint $h \mapsto (E \vdash \bullet \models ex)$. What remains is to break down the “indirect” unevaluation goals for more complex indeterminate result forms into “direct” example constraints on holes.

**Indeterminate Function Applications.** Consider an indeterminate function application $r_1 \ r_2$, with the goal that it produce some example $ex$. In general, $r_2$ can be an arbitrarily complicated
Constraint Satisfaction

\[ F \supseteq F_0 \quad \{ F \vdash ? h_i \triangleright X_i \} \quad i \in [n] \]
\[ F \mid\mid ((h_1 \mapsto X_1, \ldots, h_n \mapsto X_n) ; F_0) \]

Constraint Merging (Figure 20 of Appendix A)

\[ K_1 \circ K_2 = K \]
\[ \Sigma ; \Delta ; \text{Merge}(K) \rhd K' \]

Live Bidirectional Example Checking

\[ \Sigma ; \Delta ; F \vdash e \triangleleft X + K \]

Live Unevaluation

\[ \Sigma ; \Delta ; F \vdash r \triangleleft ex + K \]

Fig. 7. Live Bidirectional Example Checking via Live Unevaluation.

(partially evaluated) expression, so it is impossible to locally generate sufficient constraints to ensure that the function application will eventually produce the desired result. We can, however, if the input-output example \{v_2 \rightarrow ex\} to the indeterminate function r_1. If and when this example reaches a hole, then U-Hole will generate an example constraint, to be filled by synthesis (with
a function value that satisfies the input-output example). If this example reaches a function, then the U-Fix rule will use this as a “test” to make sure the receiving function is indeed consistent with this input-output example—the first premise calls the function on \(v_2\), and the second premise recursively unevaluates the intended output example \(ex\) to the result of the function call.

**Indeterminate Projections.** The U-Prj-1 and U-Prj-2 rules unevaluate examples to an indeterminate projection by using the top example \(\top\) as a placeholder for the component to be left unconstrained. For example, unevaluating 1 to \(prj_1 [E] \top h\) results in \(h \mapsto (E \vdash \bullet \models (1, \top))\).

**Indeterminate Case Expressions.** Consider the following indeterminate case expression and the goal to unevaluate the number 1:

\[
\text{case } [] \top_h \text{ of } \{\text{Nothing } \_ \rightarrow \emptyset; \text{Just } x \rightarrow x\} \Leftarrow 1.
\]

Intuitively, this requires that the scrutinee evaluates to \(\text{Just } 1\), that is, \(h \mapsto (- \vdash \bullet \models \text{Just } 1)\).

To compute this constraint, the U-Case rule attempts to unevaluate \(ex\) back along each branch \(j\). The first premise unevaluates \(C_j \top\) to the scrutinee \(r\), generating any constraints \(K_1\) required to guarantee that \(r\) 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 2) as a way to bridge this gap between constraint generation and constraint solving. To proceed down the branch expression, we bind the pattern variable \(x_j\) to \(C_j^{-1} r\); later, when evaluation resumes and the scrutinee does produce a \(C_j r'\) result, the inverse constructor retrieves the argument \(r'\). Locally, this allows U-Case to evaluate the branch to some result \(r_j\), which the third premise can use to unevaluate the goal \(ex\), generating constraints \(K_2\). For the example above, the result of evaluating the second branch expression, \(x\), is \(\text{Just }^{-1} [[] (?) h]\). Unevaluating 1 to \(\text{Just }^{-1} [[]] (?) h\) generates the constraint \(h \mapsto (- \vdash \bullet \models \text{Just }^{-1} 1)\). Finally, the U-Inverse-CTOR rule transfers the example from the inverse constructor application to a (forward) constructor application, producing \(h \mapsto (- \vdash \bullet \models \text{Just } 1)\).

This interplay between U-Case and U-Inverse-CTOR allows unevaluation to resolve branching decisions without making explicit choices as a hole-filling. The downside of this “lazy” approach is the significant degree of non-determinism. To facilitate a more efficient approach for an implementation, the U-Case-Guess rule refers to an uninterpreted predicate Guesses(\(A, \Sigma, r\)) that “eagerly” chooses a hole-filling \(F'\) that determines the scrutinee \(r\) (i.e., resumes \(r\) to some constructor application \(C_j^{-1}\)). In §4 and §5, we describe our concrete implementation of Guesses in Sketch-n-Myth. The U-Case-Guess rule is the source of hole-filling constraints produced by unevaluation; recall that U-Hole is the source of example constraints.

4 SYNTHESIS PIPELINE

Live bidirectional evaluation addresses the key technical challenge for combining program sketching and example-based synthesis, namely, how to check example satisfaction for sketches. To complete the story, what remains is to (1) derive example constraints from sketches and (2) solve the resulting constraints. In Sketch-n-Myth, the following synthesis pipeline addresses these concerns.

\[
\begin{align*}
\text{(§4.1) Constraint Collection} & \quad - \vdash e \Rightarrow r \vdash A & \quad \text{(§4.2) Constraint Solving} & \quad \text{Simplify}(A) \triangleright K
\end{align*}
\]

\[\text{Solve}(K) \rightsquigarrow F\]
Expression and Assertion Syntax (extends Figure 2)

Expressions \( e \) ::= \( \cdots \mid \) assert \( (e_1 = e_2) \)

Assertions \( A \) ::= \( \{ r_i \Rightarrow v_i \}^i_{i=1} \)

Typing and Evaluation (extends Figure 4 and Figure 5)

\[
\begin{align*}
\Sigma; \Delta; \Gamma \vdash e : T & \quad E \vdash e \Rightarrow r + A \\
\Sigma; \Delta; \Gamma \vdash e_1 : T & \quad \Sigma; \Delta; \Gamma \vdash e_2 : T \\
\Sigma; \Delta; \Gamma \vdash \text{assert} \ (e_1 = e_2) : () & \\
E \vdash e_1 \Rightarrow r_1 + A_1 & \quad E \vdash e_2 \Rightarrow r_2 + A_2 \\
& \quad r_1 \equiv_A r_2 \\
& \quad E \vdash \text{assert} \ (e_1 = e_2) \Rightarrow () + A_1 + A_2 + A_3 \\
& \quad r \equiv_A r'
\end{align*}
\]

Changes to evaluation rules in Figure 5: constraints are propagated from premises to conclusions.

See Figure 16 in Appendix A for more details.

Result Consistency

\[
\begin{align*}
[\text{RC-Refl}] & \quad r \equiv_r r' \\
[\text{RC-Pair}] & \quad r_1 \equiv_A r_1' \quad r_2 \equiv_A r_2' \\
& \quad (r_1, r_2) \equiv_A (r_1', r_2') \\
[\text{RC-Ctor}] & \quad C r \equiv_A C r' \\
\end{align*}
\]

Assertion Satisfaction and Simplification

\[
\begin{align*}
\{ F \vdash r_1 \Rightarrow r_1' + A_1 \\
F \models A_1 \quad [r_1'] = v_i \}^i_{i=1} & \quad \{ r_i \text{ final} \} \quad r_i \Leftarrow [v_i] + K_i \}^i_{i=1} \\
F \models r_i \Rightarrow v_i \}^i_{i=1} & \quad \text{Simplify}(\{ r_i \Rightarrow v_i \}^i_{i=1}) \Rightarrow K_1 \oplus \cdots \oplus K_n
\end{align*}
\]

Ex. Satisfaction and Checking (changes in Appendix A)

\[
\begin{align*}
F \vdash e \models X & \quad \Sigma; \Delta; F \vdash e \models X + K
\end{align*}
\]

Fig. 8. Constraint Collection.

(1) First, we extend the language with assert-statements that give rise to a set \( A \) of assertions as a side-effect of evaluation. Each assertion, of the form \( r_i \Rightarrow v_i \), is simplified into constraints \( K \) over holes in the sketch.

(2) Then, for each synthesis goal \( \Gamma_i \vdash \bullet_{K_i} : T_i \models X_i \) defined by \( K \), we synthesize an appropriate expression with MYTH-style techniques extended to use live bidirectional evaluation. In our formulation, the search to complete one hole may assume constraints over others. Thus, Solve iteratively solves interdependent goals until the set of constraints converges.

4.1 Constraint Collection

To serve as the source of example constraints for synthesis, we extend the expression language with the form assert \( (e_1 = e_2) \), as shown in Figure 8.

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. Rather than equality, in our approach the E-Assert rule in Figure 8 checks result consistency, \( r_1 \equiv_A r_2 \), a notion of equality modulo assumptions \( A \) about indeterminate results (i.e. results with holes in elimination position). Determinate results are consistent if structurally equal, as checked by the RC-Refl, RC-Pair, and RC-Ctor rules. Indeterminate results \( r \) are defined to be consistent with simple values \( v \)—the RC-Assert-1 and RC-Assert-2 rules generate assertions \( r \Rightarrow v \) in such cases.
Constraint Solving

\[ \Sigma; \Delta; \text{Solve}(K) \rightsquigarrow F; \Delta' \]

\textbf{[Solve-One]}

\[ h \in \text{dom}(U) \quad \Delta(h) = (\Gamma \vdash \bullet : T) \quad U(h) = X \]

\[ F; (\Gamma \vdash \bullet_h : T \models X) \rightsquigarrow \text{fill} K; \Delta' \]

\textbf{[Solve-Done]}

\[ \Sigma; \Delta; \text{Solve}(-; F) \rightsquigarrow F; \Delta \]

\[ \Sigma; \Delta; \text{Solve}(U; F) \rightsquigarrow F'; \Delta'' \]

Hole Filling

\[ \Sigma; \Delta; F; (\Gamma \vdash \bullet_h : T \models X) \rightsquigarrow \text{fill} K; \Delta' \]

\textbf{[Guess-and-Check]}

\[ (\Gamma \vdash \bullet : T) \rightsquigarrow \text{guess} e \quad (F, h \mapsto e) \vdash e \Rightarrow X + K \]

\[ F; (\Gamma \vdash \bullet_h : T \models X) \rightsquigarrow \text{fill} (-; h \mapsto e) \oplus K; - \]

\textbf{[Refine, Branch]}

\[ (\Gamma \vdash \bullet : T \models X) \rightsquigarrow \{ \text{refine, branch} \} e + \{ (\Gamma_i \vdash \bullet_{h_i} : T_i \models X_i) \}^{i \in [n]} \]

\[ \Delta' = \{ h_i \mapsto (\Gamma_i \vdash \bullet : T_i) \}^{i \in [n]} \]

\[ F; (\Gamma \vdash \bullet_h : T \models X) \rightsquigarrow \text{fill} ((h_1 \mapsto X_1, \ldots, h_n \mapsto X_n); h \mapsto e); \Delta'' \]

\textbf{[Defer]}

\[ X = (E_1 \vdash \bullet \models \top), \ldots, (E_n \vdash \bullet \models \top) \quad n > 0 \]

\[ F; (\Gamma \vdash \bullet_h : T \models X) \rightsquigarrow \text{fill} (-; h \mapsto ???_h); - \]

Fig. 9. Constraint Solving.

Figure 8 defines what it means for a hole-filling to satisfy assertions (written \( F \models A \)): for each \( r_i \Rightarrow v_i \) in \( A \), the indeterminate result \( r_i \) should resume under \( F \) and produce the value \( v_i \). Like evaluation, resumption now also produces assertions as a side-effect; Figure 18 and Figure 19 in Appendix A define the corresponding changes to resumption and example satisfaction.

**Assertion Simplification.** The \textit{Simplify}(A) procedure in Figure 8 translates assertions \( r_i \Rightarrow v_i \) into example constraints via live unevaluation (every simple value constitutes an example).

**Proposition 4.1 (Soundness of Assertion Simplification).**

If \( \text{Simplify}(A) \gg K \) and \( F \models K \), then \( F \models A \).

4.2 Constraint Solving

Figure 9 and Figure 10 define the \textbf{Core Sketch-n-Myth} procedures to solve constraints \( K \), of the form \( (U; F_0) \), collected from program assertions in a sketch. The filling \( F_0 \) represents any hole completions to which unevaluation has already (non-deterministically) committed—cf. the U-Case-Guess rule in §3.5. The \textit{Solve} procedure is the entry point for generating a filling \( F \) to complete the unfilled holes \( U \).

**Iterative Solving.** \textit{Solve}(U; F) is an iterative procedure that terminates, via \textit{Solve-Done}, when no unfilled holes remain. Otherwise, the \textit{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 \( \Delta \) and \( U \). The hole filling procedure—discussed next—is used to complete the task, generating new constraints \( K \). The new constraints are combined with the existing constraints using the semantic \textit{Merge} operation (cf. §3.5), and the resulting constraints \( K' \) are recursively solved.
Type-Directed Guessing (Figure 21 of Appendix A)

\[ \Sigma : (\Gamma \vdash \cdot : T) \rightsquigarrow \text{guess} \, e \]

Type-and-Example-Directed Refinement

\[ \Sigma ; \Delta : (\Gamma \vdash \cdot : T \models X) \rightsquigarrow \text{refine} \, e + G \]

\[ \text{[Refine-Unit]} \]

\[ \text{Filter}(X) = (E_1 \vdash \cdot \models (\cdot), \ldots, (E_n \vdash \cdot \models (\cdot)) \]

\[ (\Gamma \vdash \cdot : (\cdot) \models X) \rightsquigarrow \text{refine} \, (\cdot) + - \]

\[ \text{[Refine-Pair]} \]

\[ \text{Filter}(X) = (E_1 \vdash \cdot \models (e_{x_{11}}, e_{x_{12}}), \ldots, (E_m \vdash \cdot \models (e_{x_{m1}}, e_{x_{m2}})) \]

\[ \text{New Goals, } i = 1, 2 \]

\[ h_i \text{ fresh } G_i = (\Gamma \vdash \cdot h_i : T \models X_i) \]

\[ X_i = (E_1 \vdash \cdot \models e_{x_{i_{1}}}), \ldots, (E_m \vdash \cdot \models e_{x_{m_i}}) \]

\[ (\Gamma \vdash \cdot : (T_1, T_2) \models X) \rightsquigarrow \text{refine} \, (??_{h_1}, ??_{h_2}) + G_1, G_2 \]

\[ \text{[Refine-CTor]} \]

\[ \text{Filter}(X) = (E_1 \vdash \cdot \models C e_{x_{1}}), \ldots, (E_m \vdash \cdot \models C e_{x_{m_i}}) \]

\[ \Sigma(D)(C) = T \rightarrow D \]

\[ \text{New Goal} \]

\[ h_i \text{ fresh } G_i = (\Gamma \vdash \cdot h_i : T \models X_i) \]

\[ X_i = (E_1 \vdash \cdot \models e_{x_{i_{1}}}), \ldots, (E_m \vdash \cdot \models e_{x_{m_i}}) \]

\[ (\Gamma \vdash \cdot : D \models X) \rightsquigarrow \text{refine} \, C ??_{h_1} + G_1 \]

\[ \text{[Refine-Fix]} \]

\[ \text{Filter}(X) = (E_1 \vdash \cdot \models \{v_1 \rightarrow e_{x_{1}}\}), \ldots, (E_m \vdash \cdot \models \{v_m \rightarrow e_{x_{m}}\}) \]

\[ \text{New Goal} \]

\[ h_i \text{ fresh } e = \text{fix} f (\lambda x. ??_{h_i}) \quad G_i = (\Gamma, f : T \rightarrow T_2, x : T_1 \vdash \cdot h_i : T_2 \models X_i) \]

\[ X_i = (E_1, f \mapsto [E_1] e, x \mapsto [v_1] \vdash \cdot \models e_{x_{i_{1}}}), \ldots, (E_m, f \mapsto [E_m] e, x \mapsto [v_m] \vdash \cdot \models e_{x_{m_i}}) \]

\[ (\Gamma \vdash \cdot : T_1 \rightarrow T_2 \models X) \rightsquigarrow \text{refine} \, e + G_1 \]

Type-and-Example-Directed Branching

\[ \Sigma ; \Delta : (\Gamma \vdash \cdot : T \models X) \rightsquigarrow \text{branch} \, e + G \]

\[ \text{[Branch-Case]} \]

\[ \Sigma(D) = \{C_1 : T_1 \rightarrow D\}^{i \in [n]} \quad (\Gamma \vdash \cdot : D) \rightsquigarrow \text{guess} \, e \]

\[ \text{New Goals, } i = 1, 2, \ldots, n \]

\[ h_i \text{ fresh } G_i = (\Gamma, x_i : T_1 \vdash \cdot h_i : T \models X_i) \]

\[ X_i = \{ (E, x_i \mapsto r \vdash \cdot \models ex) \mid (E \vdash \cdot \models ex) \in \text{Filter}(X) \wedge E \vdash e \Rightarrow C_i \, r - - \} \]

\[ \text{Filter}(X) = X_1 + \cdots + X_n \]

\[ (\Gamma \vdash \cdot : T \models X) \rightsquigarrow \text{branch case} \, e \, \text{of} \, \{C_i \, x_i \rightarrow ??_{h_i}\}^{i \in [n]} + G_1, \ldots, G_n \]

Fig. 10. Guessing, Refinement, and Branching.

Hole Filling à la Myth. Following Osera and Zdancewic [2015], the core hole filling procedure \( F ; (\Gamma \vdash \cdot _h : T \models X) \rightsquigarrow \text{fill} \, K ; \Delta \) (Figure 9) augments type-directed guessing-and-checking (GUESS-AND-CHECK) with example-directed refinement (REFINE) and branching (BRANCH); these rules are discussed in turn below.
In contrast to the formulation in MYTH, however, the CORE SKETCH-N-MYTH procedure (i) refers to the hole-filling $F$ from previous synthesis tasks completed by Solve; (ii) may generate example constraints over other holes in the program; and (iii) may fill other holes in the program, in addition to the goal $h$. We additionally include a rule, Defer, for tasks where all examples are top—these constraints are not imposed directly from program assertions, but are created internally by unevaluation. Defer "fills" such holes with $h$.

**Guessing-and-Checking.** The Guess-and-Check rule uses the type-directed guessing procedure $(\Gamma \vdash \bullet : T) \leadsto guess e$ (Figure 10) to generate a well-typed term $e$. Guessing amounts to straightforward inversion of expression type checking rules; Appendix A provides the full definition. The candidate expression $e$ is then checked for example consistency using live bidirectional example checking (cf. Figure 7). The resulting constraints $K$ are the source of the aforementioned differences (i), (ii), and (iii) compared to the MYTH hole-filling procedure.

Type-directed guessing also serves as one single way to implement the Guesses($\Delta, \Sigma, r$) predicate for unevaluating case expressions, as discussed in §3.5.

**Example-Directed Refinement.** In contrast to simply enumerating well-typed terms, the Refine rule refers to the example-directed refinement procedure $(\Gamma \vdash \bullet : T \mid X) \leadsto refine e + G$ (Figure 10) to quickly synthesize a partial solution $e$, which refers to freshly created holes $h_i$ 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 meta-theory, the typings for fresh holes are recorded in output hole type context $\Delta'$.

Each of the refinement rules first uses Filter$(X)$ to remove top examples and then inspects the structure of the remaining examples. For unit-type goals, the Refine-Unit simply synthesizes the unit expression $\bot$. For pair-type goals, the Refine-Pair synthesizes the partial solution $(h_1, h_2)$, assuming solutions to two fresh subgoals created 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 head $C$.

For function-type goals, the Refine-Fix rule synthesizes the function sketch $\text{fix } f (\lambda x. h_i)$. The environments inside example constraints $X_i$ for the function body $x \mapsto f$ to this function sketch (closed by the appropriate environments $E_i$). As a result, any recursive calls to $f$ will evaluate to closures of $h_i$, to be constrained by live bidirectional example checking and thus avoiding the need for trace-complete examples (cf. Limitation A in §1).²

**Example-Directed Branching.** Lastly, the Branch rule refers to the example-directed branching procedure $(\Gamma \vdash \bullet : T \mid X) \leadsto branch e + G$ (Figure 10) to guess an expression on which to branch. The signature of the branching procedure follows that of refinement. The single rule, Branch-Case, chooses an arbitrary expression $e$ (of arbitrary datatype $D$) to scrutinize, and then distributes the examples $X$ onto the constructors $C_1$ through $C_n$ corresponding to the datatype $D$. The examples $X_i$ for the branches are defined by evaluating the scrutinee $e$ to a determinate result

---

¹ MYTH requires trace-complete sets of input-output examples, called partial functions $(v_i \mapsto e x_i)^i \in [n]$, and includes them in the expression grammar to resolve recursive calls to the function being synthesized (cf. Check-Input-Output in §1). In addition to the usability implications of the trace-completeness requirement, the technical formulation is complicated by a non-standard value compatibility notion [Osera and Zdancewic 2015, §3.3], to approximate value equality when treating a partial function as a "lookup table."

² The Guess-and-Check rule provides a second antidote for trace-completeness: when guessing an expression $\text{fix } f (\lambda x. e)$ to fill $h$, the extended hole-filling $F, h \mapsto \text{fix } f (\lambda x. e)$ "ties the recursive knot" before checking example consistency. But because functions can always be synthesized "eagerly" by Refine-Fix—and because large terms, such as recursive functions, are unlikely to be guessed in their entirety (§5)—this source of expressiveness is not needed in practice.
and gathering those which share the constructor head \( C_i \). The final premise ensures that every example is distributed to the subgoal for some branch.

Note that Branch-Case includes a “knob” that can be turned: the scrutinee \( e \) could be allowed to evaluate to an indeterminate result, subsequently constrained by live unevaluating examples of the form \( C_i \top \). In our current presentation, however, we choose not to introduce this additional source of expressiveness and non-determinism.

**Soundness.** The following defines correctness of the Core Sketch-n-Myth synthesis pipeline.

**Proposition 4.2 (Soundness of Synthesis).**

\[
\text{If } - \vdash e \Rightarrow r \vdash A \text{ and Simplify}(A) \vdash K \text{ and } \Sigma ; \Delta ; \text{Solve}(K) \leadsto F ; \Delta', \text{ then } \Sigma \vdash F : \Delta \cup \Delta' \text{ and } F \models A.
\]

### 5 IMPLEMENTATION AND EVALUATION

We implemented the Core Sketch-n-Myth synthesis algorithm as an OCaml server and added support within the Sketch-n-Sketch programming system [Chugh et al. 2016; Mayer et al. 2018] to interface with the newly-created backend. Our server implementation consists of approximately 3,400 lines of OCaml code and our extension to Sketch-n-Sketch consists of approximately 2,000 lines of Elm code.

Compared to the core language, our implementation supports Haskell/Elm-like syntax, \( n \)-ary tuples, let-binding, and let-bound recursive function definitions. Our implementation also supports higher-order function examples, following the formulation of Osera and Zdancewic [2015]; this feature is orthogonal to the extensions in our work. Our prototype lacks many of the syntactic conveniences used in code listings in the paper such as nested pattern matching, infix list operators (: : ) and (++), and type inference for holes. Moreover, to ensure termination, we do not support recursive functions whose first argument is not structurally decreasing. None of the differences pose fundamental challenges, but they do result in slightly different input and output.

**Optimizations.** We adopt two main optimizations from Myth [Osera 2015; Osera and Zdancewic 2015]. The first is to optimize term guessing by restricting EGUESS rules to proof relevant terms [Anderson et al. 1992] and caching them for use across different paths of the synthesis search. The second is to employ a synthesis 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.

#### 5.1 Experiments

In addition to our qualitative analysis of using Sketch-n-Myth to construct the examples from §2, we evaluated our implementation of Sketch-n-Myth quantitatively by running it against the set of benchmarks used to test Myth.

Below, we describe a baseline configuration to compare Sketch-n-Myth to Myth, followed by two experiments to evaluate the effects of how Sketch-n-Myth (a) eliminates the trace-completeness requirement and (b) supports sketching.

**Experiment 0: Baseline Configuration.** If the user supplies no program sketch to Sketch-n-Myth whatsoever, then Sketch-n-Myth should behave exactly the same as Myth (sans some more advanced optimizations). And, indeed, the first column of Figure 11 indicates that Sketch-n-Myth passes 37 out of 43 of the same benchmarks in a very similar amount of time.

Of the remaining 6 not successfully synthesized in our implementation, Myth finds an “inside-out” solution for several that pattern matches on a call to the recursive function being synthesized. We hypothesize that “turning the Branch-Case knob” (cf. §4.2) would provide the necessary expressiveness to synthesize such solutions, though the additional nondeterminism would need to
| Name         | #Ex | #Sol | Time (s) | #Ex | All | Rec | Non-Rec | Time (s) |
|--------------|-----|------|----------|-----|-----|-----|---------|----------|
| bool_band    | 4   | 4    | 0.0037   | 3   | 4/4 | 0/0 | 3/5     | 0.0032   |
| boolior      | 4   | 4    | 0.0033   | 3   | 4/4 | 0/0 | 3/5     | 0.0031   |
| bool_impl    | 4   | 1    | 0.0031   | 3   | 1/1 | 0/0 | 1/1     | 0.0032   |
| bool_neg     | 2   | 1    | 0.0012   |     |     |     |         |          |
| bool_xor     | 4   | 28   | 0.0069   | 3   | 4/8 | 0/0 | 2/3     | 0.0061   |
| list_append  | 6   | 1    | 0.0060   | 3   | 1/17| 1/3 | 0/3     | 0.0063   |
| list_concat  | 6   | 3    | 0.0074   | 3   | 3/9 | 3/3 | 0/3     | 0.0069   |
| list_drop    | 11  | 30   | 0.0250   | 5   | 9/30| 2/3 | 0/0     | 0.0148   |
| list_filter  | 8   | 30   | 0.0984   | 4   | 15/30| 2/3 | 0/3     | 0.0699   |
| list_fold    | 9   | 1    | 0.7553   | 3   | 1/30| 1/3 | 0/0     | 0.6966   |
| list_hd      | 3   | 1    | 0.0023   | 2   | 1/3 | 0/1 | 1/2     | 0.0023   |
| list_inc     | 4   | 3    | 0.0113   | 2   | 3/3 | 0/0 | 3/3     | 0.0098   |
| list_last    | 6   | 1    | 0.0063   | 4   | 1/1 | 1/1 | 0/0     | 0.0063   |
| list_length  | 3   | 1    | 0.0023   | 2   | 1/3 | 1/1 | 0/2     | 0.0023   |
| list_map     | 8   | 2    | 0.0404   | 3   | 2/18| 1/3 | 0/3     | 0.0428   |
| list_nth     | 13  | 10   | 0.1170   | 6   | 6/30| 3/3 | 0/0     | 0.0212   |
| list_rev_append | 5  | 1    | 0.0990   | 3   | 1/1 | 1/1 | 0/0     | 0.0655   |
| list_rev_fold | 5   | 2    | 0.0275   | 2   | 2/2 | 0/0 | 2/2     | 0.0248   |
| list_rev_snoc | 5   | 1    | 0.0082   | 2   | 1/13| 1/1 | 0/3     | 0.0095   |
| list_rev_tailcall | 8  | 1    | 0.0061   | 3   | 1/1  | 1/1  | 0/0     | 0.0051   |
| list_snoc    | 8   | 6    | 0.0117   | 4   | 6/30| 2/3 | 0/3     | 0.0104   |
| list_sorted_insert | 7 | 1    | 0.0123   | 1   | 2/19| 1/1 | 0/3     | 0.0122   |
| list_sorted_insert | 12 | 30   | 5.6730   | 7   | 20/30| 3/3 | 0/0     | 2.5098   |
| list_stutter | 3   | 2    | 0.0026   | 2   | 2/18| 2/2 | 0/3     | 0.0047   |
| list_sum     | 3   | 4    | 0.0239   | 2   | 4/5 | 0/0 | 3/5     | 0.0214   |
| list_take    | 12  | 30   | 0.0655   | 6   | 2/6 | 2/3 | 0/0     | 0.0415   |
| list_tl      | 3   | 2    | 0.0025   | 2   | 2/6 | 0/0 | 2/3     | 0.0028   |
| nat_add      | 9   | 2    | 0.0051   | 3   | 2/15| 1/3 | 0/3     | 0.0042   |
| nat_iseven   | 4   | 1    | 0.0030   | 3   | 1/2 | 1/1 | 0/1     | 0.0038   |
| nat_max      | 9   | 30   | 0.0374   | 8   | 20/30| 1/3 | 0/0     | 0.0331   |
| nat_pred     | 3   | 2    | 0.0015   | 2   | 2/4 | 0/0 | 2/3     | 0.0015   |
| tree_collect_leaves | 6 | 9    | 0.0667   | 3   | 9/18| 1/3 | 0/0     | 0.0378   |
| tree_count_leaves | 7 | 30   | 3.0997   | 3   | 30/30| 3/3 | 0/0     | 1.1925   |
| tree_count_nodes | 6 | 30   | 0.3096   | 3   | 30/30| 3/3 | 0/0     | 0.1733   |
| tree_inorder | 5   | 9    | 0.1067   | 3   | 9/18| 1/3 | 0/0     | 0.0674   |
| tree_map     | 7   | 12   | 0.0477   | 3   | 9/30| 1/3 | 0/0     | 0.0518   |
| tree_preorder | 5  | 30   | 0.1357   | 3   | 30/30| 3/3 | 0/0     | 0.1352   |

Fig. 11. Experiments 0 and 1: Baseline Configuration and Fewer Examples.

All: all solutions. (Non-)Rec: (non-)recursive window solutions.

Fractions represent valid out of total solutions.

be tamed. We hypothesize that the remaining examples could be synthesized in SKETCH-N-MYTH if extended with additional advanced optimizations described in [Osera 2015].

**Experiment 1: Fewer Examples.** For the second column of Figure 11, we manually removed examples from all but the most basic test suite program (bool_neg requires two examples to be synthesized correctly) and ran the synthesis algorithm with the smaller set of examples.

For this experiment, we marked a solution as correct if it conformed to the input-output example suite from the baseline configuration. We then used a simple heuristic to rank correct solutions: sort them by AST size and show the (at most) top-3 non-recursive solutions and the (at most) top-3 recursive solutions. We call this subset of solutions the solution window. Because of the weaker specification, SKETCH-N-MYTH typically synthesizes many more solutions than in the baseline configuration; however, for each synthesis task, SKETCH-N-MYTH synthesized a correct solution
in the solution window, on average needing 55% of the number of examples and taking less than twice the amount of time as in first column (and sometimes even faster).

**Experiment 2: General Sketching Strategies.** To evaluate the expressiveness of sketching, we devised a small set of sketching strategies, tested them on each of the applicable Myth benchmark programs, and recorded the results in Figure 12, Figure 13, and Figure 14. Specifically, we identified three generally-applicable sketching strategies:

(i) an *outer match sketch* that performs case analysis on the correct argument of the function but leaves holes in the branches of the case expression;

(ii) a *tail-recursive sketch* that performs case analysis on the recursive argument of the function, returns the accumulator variable in the base case, and leaves a hole in the recursive branch; and

(iii) a *base case sketch* that performs case analysis on the correct argument of the function, fills in the base case properly, and leaves a hole in the recursive branch.

The first and third of these strategies can be applied semi-automatically (but require user input for selecting the correct variable to match on and, in the case of the general recursion sketch, what the base case should be) while the second strategy is can be applied fully automatically (but only to tail-recursive functions).

Of the 37 benchmark programs, the outer match strategy was applicable to 31 programs, the tail-recursive strategy was applicable to 2 programs, and the base case strategy was applicable to 22 programs.

The outer match sketch strategy provides very little additional information to the synthesizer (only the first scrutinee to match against), and, accordingly, the results of the experiment indicate that not much is gained in terms of the number of examples that the user must provide to the program. The main benefit of this strategy is helping to narrow the search space for the synthesizer, as well as helping prevent overspecialization when providing very few examples (such as for bool_and, bool_bor, and bool_impl, as shown in Figure 12).

The tail-recursive sketch strategy has the benefit of being completely automatable due to the consistent structure of tail-recursive functions. However, because it only applies to tail-recursive function, it is much more limited in its applicability. When applicable, though, the additional information that it provides to the synthesizer is significant, and can drastically cut down the number of examples the user needs to specify, as shown in Figure 13.

A middle ground between the previous two strategies is the base case sketch strategy: it is applicable to all recursive functions and provides significant additional information to the synthesizer, at the cost of not being fully automatable. Figure 14 reveals the dramatic difference in the number of examples the user needs to specify after having sketched out the base case of the recursive function at hand. On average, only 28% of the examples needed for specifying a function in Myth were needed when applying the base case sketch strategy. No benchmark program for which this strategy was applicable (i.e., recursive functions) required more than 3 examples, with a mean of 1.78 examples being necessary.

### 6 RELATED WORK

Program synthesis is a large and active research area; Gulwani et al. [2017] provide a recent survey of developments. Our work directly extends the example-based techniques of Osera and Zdancewic [2015], so we discussed technical differences between Myth and Sketch−N−Myth throughout the paper. We limit our discussion below to other work closely related to our goals and techniques.
Justin Lubin, Nick Collins, Cyrus Omar, and Ravi Chugh

| Name          | #Ex | #Sol | Time (s) | #Ex | All  | Rec  | Non-Rec | Time (s) |
|---------------|-----|------|----------|-----|------|------|---------|----------|
| bool_band     | 4   | 2    | 0.0035   | 2   | 2/4  | 0/0  | 2/3     | 0.0033   |
| bool_bor      | 4   | 2    | 0.0026   | 2   | 2/4  | 0/0  | 2/3     | 0.0027   |
| bool_impl     | 4   | 1    | 0.0034   | 2   | 1/2  | 0/0  | 1/2     | 0.0031   |
| bool_neg      | 2   | 1    | 0.0008   | —   | —/—  | —/—  | —/—     | —/—      |
| bool_xor      | 4   | 14   | 0.0035   | 3   | 2/4  | 0/0  | 2/3     | 0.0030   |
| list_append   | 6   | 1    | 0.0137   | 3   | 1/23 | 1/3  | 0/3     | 0.0171   |
| list_concat   | 6   | 8    | 0.0143   | 3   | 8/17 | 3/3  | 0/3     | 0.0123   |
| list_drop     | 11  | 30   | 0.0385   | 5   | 9/30 | 2/3  | 0/0     | 0.0149   |
| list_hd       | 3   | 1    | 0.0035   | 2   | 1/3  | 0/1  | 1/2     | 0.0032   |
| list_last     | 6   | 1    | 0.0055   | 4   | 1/1  | 1/1  | 0/0     | 0.0055   |
| list_length   | 3   | 1    | 0.0039   | 2   | 1/3  | 1/1  | 0/2     | 0.0037   |
| list_map      | 8   | 2    | 1.0217   | 3   | 2/18 | 1/3  | 0/3     | 1.3518   |
| list_nth      | 13  | 30   | 0.0502   | 7   | 4/12 | 3/3  | 0/0     | 0.0251   |
| list_rev_append| 5  | 15   | 1.1513   | 3   | 15/15| 3/3  | 0/0     | 0.7198   |
| list_rev_snoc | 5   | 2    | 0.0638   | 2   | 2/18 | 2/2  | 0/3     | 0.0963   |
| list_rev_tailcall | 8  | 1    | 0.0088   | 3   | 1/1  | 1/1  | 0/0     | 0.0113   |
| list_snoc     | 8   | 2    | 0.0109   | 4   | 2/10 | 2/3  | 0/3     | 0.0091   |
| list_sort_sorted_insert | 7  | 4    | 0.1016   | 2   | 2/30 | 2/3  | 0/3     | 0.0780   |
| list_stutter  | 3   | 2    | 0.0036   | 2   | 2/18 | 2/2  | 0/3     | 0.0053   |
| list_take     | 12  | 11   | 0.0365   | 6   | 2/6  | 2/3  | 0/0     | 0.0108   |
| list_tl       | 3   | 2    | 0.0030   | 2   | 2/6  | 0/0  | 2/3     | 0.0033   |
| nat_add       | 9   | 2    | 0.0082   | 3   | 2/24 | 1/3  | 0/3     | 0.0044   |
| nat_iseven    | 4   | 1    | 0.0022   | 3   | 1/2  | 1/1  | 0/1     | 0.0023   |
| nat_max       | 9   | 30   | 0.0594   | 8   | 21/30| 2/3  | 0/0     | 0.0477   |
| nat_pred      | 3   | 2    | 0.0016   | 2   | 2/4  | 0/0  | 2/3     | 0.0016   |
| tree_collect_leaves | 6  | 1    | 0.0189   | 3   | 1/2  | 1/2  | 0/0     | 0.0163   |
| tree_count_leaves | 7  | 30   | 1.9255   | 4   | 0/30 | 0/3  | 0/0     | 0.0874   |
| tree_count_nodes | 6  | 18   | 0.1376   | 3   | 18/18| 3/3  | 0/0     | 0.0801   |
| tree_inorder  | 5   | 1    | 0.0222   | 3   | 1/2  | 1/2  | 0/0     | 0.0187   |
| tree_map      | 7   | 30   | 0.8314   | 3   | 9/30 | 1/3  | 0/0     | 0.6180   |
| tree_preorder | 5   | 30   | 0.1370   | 3   | 30/30| 3/3  | 0/0     | 0.1341   |

**Fig. 12.** Experiment 2(i): *Outer Match Sketch Strategy.*

| Name       | #Ex | #Sol | Time (s) |
|------------|-----|------|----------|
| list_fold  | 9   | 30   | 5.8512   |
| list_rev_tailcall | 8  | 1    | 0.0082   |

**Fig. 13.** Experiment 2(ii): *Tail Recursive Sketch Strategy.*

**Program Sketching.** The sketching approach to synthesis—where holes in the program are constrained by ordinary assert statements—has been most thoroughly studied in two systems. **SKETCH** [Solar-Lezama 2008; Solar-Lezama 2009; Solar-Lezama et al. 2005, 2006] is an imperative, C-like language in which holes ?? are completed at compile-time. **ROSETTE** [Torlak and Bodik 2013, 2014] is an untyped functional language based on Racket [Flatt and PLT 2010] in which holes are generated dynamically (via the define-symbolic operator) and constraints are collected through a combination of concrete and symbolic execution; the program demands hole completions later during evaluation (via the synthesize operator). Holes in both SKETCH and ROSETTE range over integer-typed expressions.

Sketching has not yet been formally investigated in the context of richly-typed functional programming languages. Our work addresses the question of how to extend example-based synthesis techniques for such languages with support for sketching. In contrast, **SYNQUID** [Polikarpova et al. 2016] and **LEON** [Kneuss et al. 2013] synthesize recursive functions using solver-based techniques.
driven by logical specifications (e.g., SMT-based refinement types in SYNQUID)—it would thus be natural to extend their approaches with direct support for sketching. Example-based and logic-based specifications are complementary—examples may fare better (i.e., in terms of specification size) for functions which expose some representation details to clients, while logical specifications fare better for those which do not [Polikarpova et al. 2016, §4.3]. Combining these techniques in a functional sketching system is an interesting direction for future work.

**Live Evaluation.** We borrow the technique for partially evaluating sketches from HAZELNUT LIVE [Omar et al. 2019]. We note several technical differences in our formulation.

We choose a natural semantics presentation [Kahn 1987] for CORE SKETCH-N-MYTH rather than one based on substitution. Because HAZELNUT LIVE results are simply expressions, their fill-and-resume mechanism is defined using substitution and reduction; we formulated a separate notion of results and resumption to evaluate them. However, we expect that CORE SKETCH-N-MYTH expressions could be elaborated to an internal language—akin to our results, extended with variables—for evaluation; we leave the details to future work. Final results in HAZELNUT LIVE are indeterminate expressions or values; determinate results here include non-values, e.g., $(r_1, r_2)$ where either component is indeterminate.

HAZELNUT LIVE supports binary sums and products, and their implementation [Hazel Team 2019] extends the system with recursive functions and primitive lists; we formulate CORE SKETCH-N-MYTH with recursive functions and named, recursive algebraic datatypes because of our goal to extend MYTH. 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. 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 $\Delta$; the same approach can be employed in our setting without complication.
Bidirectional Evaluation. Several proposals define unevaluators, or backward evaluators, that allow changes to the output value of an expression to affect changes to the expression. Perera et al. [2012] propose an unevaluator that, given an output modified with value holes, slices away program expressions that do not contribute to the parts of the output that remain—useful in an interactive debugging session, for example. Matsuda and Wang [2018] propose a bidirectional evaluator—which forms a lens [Foster et al. 2007]—for manipulating first-order values in a language of residual expressions, containing no function applications in elimination positions. Mayer et al. [2018] generalize this approach to arbitrary programs and values in a higher-order functional language, effectively mapping output value changes to program repairs.

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 CONCLUSION

This paper brings together program sketching—previously developed using constraint-based synthesis techniques with static verification—and example-based synthesis techniques with dynamic verification, within a richly-typed functional programming language. Along the way, we addressed a shortcoming of state-of-the-art example-based synthesis, namely, the reliance on trace-complete examples from the user.

We believe that granularly interleaving synthesis with evaluation, “live” throughout the program development process, will contribute to the vision of a truly usable programmer’s assistant [Teitelman 1972]. Several challenges remain to achieve this long-term goal. First, our techniques need to be extended with support for common features, such as type polymorphism, imperative updates, modules, and constructs for parallelism. Second, it will be important to provide additional ways for users to communicate intent, for example, with syntax constraints to define grammars of desired completions [Alur et al. 2013] and feedback to label desirable and undesirable parts of candidate solutions [Peleg et al. 2018]. Heuristics and ranking algorithms—taking into account existing code repositories (e.g. [Feng et al. 2017; Gvero et al. 2013]) and edit histories—must also be developed for situations where a large number of candidate solutions are synthesized. Furthermore, synthesis results must be explained and visualized in more comprehensible ways—because example-based techniques can be prone to subtle biases, because synthesis will not always find a complete solution, and because user intent often evolves during development. To serve as a practical tool for programming, these challenges need to be addressed while delivering good performance, interactivity, and predictability.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 1814900 (“Semantic Foundations for Hole-Driven Development”) and Grant No. 1651794 (“Direct Manipulation Programming Systems”). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

Awst Albarghouthi, Sumit Gulwani, and Zachary Kincaid. 2013. Recursive Program Synthesis. In Computer Aided Verification (CAV).
Program Synthesis with Live Bidirectional Evaluation

Rajeev Alur, Rastislav Bodík, 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).

Davide Ancona. 2014. How to Prove Type Soundness of Java-like Languages Without Forgoing Big-step Semantics. In Workshop on Formal Techniques for Java-like Programs (FTTJP).

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

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

Ravi Chugh, Brian Hempel, Mitchell Spradlin, and Jacob Albers. 2016. Programmatic and Direct Manipulation, Together at Last. In Conference on Programming Language Design and Implementation (PLDI).

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

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).

Matthew Flatt and PLT. 2010. Reference: Racket. Technical Report PLT-TR-2010-1. PLT Design Inc. https://racket-lang.org/tr1/.

Etienne Kneuss, Ivan Kuraj, Viktor Kunčak, 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).

In Conference on Programming Language Design and Implementation (PLDI).

Hazel Team. 2019. Hazel. (2019). https://github.com/hazelgrove/hazel.

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

J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. 2007. Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View-Update Problem. ACM Transactions on Programming Languages and Systems (TOPLAS) 29, 3 (2007).

Jonathan Frankle. 2015. Type-Directed Synthesis of Products. CoRR abs/1510.08121 (2015). http://arxiv.org/abs/1510.08121

Jeremy G. Siek and Walid Taha. 2006. Gradual Typing for Functional Languages. In Workshop on Types in Languages Design and Implementation (TLDI).

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

Eliezer A. Friedman, Grossman, Oded. 2000. Multiple-Step Operational Semantics. In Conference on Programming Language Design and Implementation (PLDI).

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).

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

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

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

Sean McDirmid. 2013. Usable Live Programming. In Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward!).

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).

Hila Peleg, Sharon Shoham, and Eran Yahav. 2018. Programming Not Only by Example. In International Conference on Software Engineering (ICSE).

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)*.

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 Bodik, 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)*.

Warren Teitelman. 1972. Automated Programming: The Programmer’s Assistant. In *AFIPS Fall Joint Computing Conference (2)*, 917–921.

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)*.
A APPENDIX

This section provides additional definitions for §3 and §4, including changes to definitions in §3 required to support the assertions A presented in §4.

A.1 Type Checking

Figure 15 defines type checking for results and examples.

**Result Typing**

\[
\begin{align*}
\text{[RT-Fix]} & : \Sigma; \Delta \vdash r : T & \text{[RT-Hole]} & : \Delta(??_h) = (\Gamma \vdash \bullet : T) & \Sigma; \Delta \vdash E : \Gamma \\
\Sigma; \Delta \vdash E : \Gamma & & \Sigma; \Delta \vdash [E] f i x f (\lambda x. e) : T & & \Sigma; \Delta \vdash [E] ??_h : T \\
\text{[RT-Unit]} & \Sigma; \Delta \vdash () : () & \text{[RT-Pair]} & \{ \Sigma; \Delta \vdash r_i : T_i \}^{i \in [2]} & \Sigma(D)(C) = T \to D & \Sigma; \Delta \vdash r : T \\
\Sigma; \Delta \vdash r_1 : T_2 & & \Sigma; \Delta \vdash (r_1, r_2) : (T_1, T_2) & & \Sigma; \Delta \vdash C r : D \\
\Sigma; \Delta \vdash r_1 r_2 : T & & \Sigma; \Delta \vdash \text{pr}_j r_i : T_i & & \Sigma; \Delta \vdash [E] \text{case } r \text{ of } \{ C_i x_i \to e_i \}^{i \in [n]} : T \\
\end{align*}
\]

**Environment Typing**

\[
\begin{align*}
\Sigma; \Delta \vdash \_ : \_ & & \Sigma; \Delta \vdash E : \Gamma & & \Sigma; \Delta \vdash r : T \\
\Sigma; \Delta \vdash () : () & & \Sigma; \Delta \vdash (E, x \mapsto r) : (\Gamma, x:T) \\
\end{align*}
\]

**Example Typing**

\[
\begin{align*}
\text{[XT-Unit]} & : \Sigma; \Delta \vdash () : () & \text{[XT-Pair]} & \{ \Sigma; \Delta \vdash ex_i : T_i \}^{i \in [2]} & \Sigma(D)(C) = T \to D & \Sigma; \Delta \vdash ex : T \\
\Sigma; \Delta \vdash () : () & & \Sigma; \Delta \vdash (ex_1, ex_2) : (T_1, T_2) & & \Sigma; \Delta \vdash C ex : D \\
\end{align*}
\]

\[
\begin{align*}
\text{[XT-Ctor]} & : \Sigma; \Delta \vdash ex : T & \text{[XT-Top]} & \Sigma; \Delta \vdash \_ : T & \Sigma; \Delta \vdash ex : T \\
\Sigma(D)(C) = T \to D & & \Sigma; \Delta \vdash [v] : T_1 & & \Sigma; \Delta \vdash \{ v \to ex \} : T_1 \to T_2 \\
\end{align*}
\]

Fig. 15. Result and Example Type Checking.
A.2 Type Soundness

Type checking and evaluation are related by the following properties.

**Proposition A.1 (Type Preservation).**
If \( \Sigma ; \Delta ; \Gamma \vdash e : T \) and \( \Sigma ; \Delta \vdash E : \Gamma \) and \( E \vdash e \rightarrow r \), then \( \Sigma ; \Delta \vdash r : T \).

**Proposition A.2 (Progress).**
For all \( k \), if \( \Sigma ; \Delta ; \Gamma \vdash e : T \) and \( \Sigma ; \Delta \vdash E : \Gamma \), there exists \( r \) s.t. \( E \vdash e \Rightarrow r \) and \( \Sigma ; \Delta \vdash r : T \).

The progress property is complicated by the fact that, in a big-step semantics, non-terminating computations are not necessarily distinguished from stuck ones [Leroy and Grall 2009]. Using a technique similar to that described by Ancona [2014], we augment evaluation with a natural \( k \) that limits the beta-reduction depth of an evaluation derivation. The augmented evaluation judgment \( E \vdash e \Rightarrow r , k + A \) (Figure 16) asserts that evaluation produced a particular result or that it reached the specified depth before doing so. (Augmented evaluation also collects assertions \( A \), as described in §4.)

---

**Augmented Evaluation**

\[
E \vdash e \Rightarrow k r + A
\]

---

\[
\begin{align*}
& \text{[AE-Fix]} \quad E \vdash f \fix (\lambda x . e) \Rightarrow k [E] \fix f (\lambda x . e) + \quad \text{[AE-Var]} \quad x \mapsto r \in E \\
& \quad \quad \quad \quad \quad \quad E \vdash x \Rightarrow k r + A \\
& \text{[AE-Untr]} \quad E \vdash () \Rightarrow k () + \quad \text{[AE-Adv]} \quad \{ E \vdash e_i \Rightarrow k r_i + A_i \}^{i \in [2]} \\
& \quad \quad \quad \quad \quad \quad E \vdash (e_1 , e_2) \Rightarrow k (r_1 , r_2) + A + A_2 \\
& \text{[AE-App]} \quad E \vdash e \Rightarrow k (r_1 , r_2) + A \\
& \quad \quad \quad \quad \quad \quad E \vdash \pr j_{i \in [2]} e \Rightarrow k r_i + A \\
\end{align*}
\]

---

Fig. 16. Augmented Evaluation with beta-depth limit and assertions. The only rules that decrease the beta-depth limit are AE-App and AE-Case. The only rule that introduces assertions is AE-Assert.
Evaluation Failure

\[
E \vdash e \Rightarrow_k \emptyset
\]

| Rule | Description |
|------|-------------|
| [EF-Pair] | \( \exists i \in [1, 2] \quad E \vdash e_i \Rightarrow_k \emptyset \Rightarrow (e_1, e_2) \Rightarrow_k \emptyset \) |
| [EF-Ctor] | \( E \vdash e \Rightarrow_k \emptyset \) |
| [EF-App-Eval] | \( E \vdash e_1 \Rightarrow r_1, k \quad r_1 = [E_f] \text{fix} f (\lambda x . e_f) \quad E \vdash e_2 \Rightarrow r_2, k \quad E_f, f \mapsto r_1, x \mapsto r_2 \vdash e_f \Rightarrow_{k-1} \emptyset \quad E \vdash e_1, e_2 \Rightarrow_k \emptyset \) |
| [EF-App-Part] | \( \exists i \in [1, 2] \quad E \vdash e_i \Rightarrow_k \emptyset \Rightarrow e_1, e_2 \Rightarrow_k \emptyset \) |
| [EF-Pj] | \( E \vdash e \Rightarrow_k \emptyset \Rightarrow \text{pr}_j \big|_{i \in [2]} e \Rightarrow_k \emptyset \) |
| [EF-Case-Schut] | \( E \vdash e \Rightarrow_k \emptyset \Rightarrow \text{case } e \text{ of } \{ C_i \ x_i \mapsto e_i \}^{i \in [n]} \Rightarrow_k \emptyset \) |
| [EF-Case] | \( \exists j \in [1, n] \quad E \vdash e \Rightarrow C_j r, k \quad E, x_j \mapsto r \vdash e_j \Rightarrow_{k-1} \emptyset \Rightarrow \text{case } e \text{ of } \{ C_i \ x_i \mapsto e_i \}^{i \in [n]} \Rightarrow_k \emptyset \) |
| [EF-Assert-Part] | \( \exists i \in [1, 2] \quad E \vdash e_i \Rightarrow_k \emptyset \Rightarrow \text{assert } (e_1 = e_2) \Rightarrow_k \emptyset \) |
| [EF-Assert] | \( E \vdash e_1 \Rightarrow r_1 \vdash_A, k \quad E \vdash e_2 \Rightarrow r_2 \vdash_A, k_1 \quad r_1 \neq_A r_2 \quad E \vdash \text{assert } (e_1 = e_2) \Rightarrow_k \emptyset \) |
| [EF-Limit] | \( E \vdash e \Rightarrow_0 \emptyset \) |

Figure 17. Evaluation failure (due to inconsistent assertion or fuel exhaustion)

Figure 16 shows how the evaluation judgment can be augmented to add fuel that limits the depth of beta reductions that can occur during evaluation. Note that for simplicity, the fuel is only depleted in recursive invocations that extend the environment. Also note that this relation is exactly the same as the ordinary evaluation relation, except for the beta-depth-limit \( k \). As such, a progress theorem proven over this relation reflects the properties of the original evaluation relation.

Figure 17 is a new relation that captures the scenarios in which evaluation fails—namely, an assertion over inconsistent results. In non-assert cases, a failure to evaluate a subterm is simply propagated upwards. Because it calls into evaluation for some rules, it might not terminate and is thus forced to have a beta-depth-limit as well. In Section 3, the rule E-LIMIT was described as being part of the evaluation judgment, but here we have moved it to EF-LIMIT in the failure judgment, which permits the intuitive interpretation that failure occurs either in the case of an ordinary assertion failure, or whenever fuel runs out, whereas the evaluation judgment only goes through if it terminates.

A.3 Resumption

Figure 18 defines how to resume partially evaluated expressions.

**Proposition A.3 (Fill-and-Resume).**

If \( E \vdash e \Rightarrow r \) and \( F \vdash r \Rightarrow r' \), then \( E' \vdash [F] e \Rightarrow r' \) where \( F \vdash E \Rightarrow E' \).

If \( - \vdash e \Rightarrow r \) and \( F \vdash r \Rightarrow r' \), then \( - \vdash [F] e \Rightarrow r' \).
Resumption

\[ \begin{align*}
F \vdash r &\Rightarrow r' + A \\
\text{[R-Hole-Resume]} \quad &\quad F(i) = e_i \quad E \vdash e_i \Rightarrow r + A \quad F \vdash r \Rightarrow r' + A' \\
\text{[R-Hole-Indet]} \quad &\quad i \notin \text{dom}(F) \quad F \vdash E \Rightarrow E' + A \\
\text{[R-Pair]} \quad &\quad F \vdash \{r_1, r_2\} \Rightarrow (r_1', r_2') + A_1 + A_2 \\
\text{[R-Unit]} \quad &\quad F \vdash () \Rightarrow () + - \\
\text{[R-App]} \quad &\quad F \vdash r_1 \Rightarrow r'_1 + A_1 \quad F \vdash r_2 \Rightarrow r'_2 + A_2 \\
\text{[R-App-Indet]} \quad &\quad F \vdash r \Rightarrow (r_1, r_2) + A \\
\text{[R-Case]} \quad &\quad \exists j \in [1, n] \quad \exists j \in [1, n] \quad F \vdash e_j = e_i \quad F \vdash (E)\lambda x_j, e_j ((C_j)^{-1} r) \Rightarrow r_j + A' \\
\text{[R-Case-Indet]} \quad &\quad F \vdash E \Rightarrow E' + A' \quad r'' = [E'] \text{ case } r' \text{ of } \{C_i x_i \rightarrow e_i\}_{i \in [n]} \\
\text{[R-Unwrap-Ctor]} \quad &\quad F \vdash r \Rightarrow C_j r_j + A \\
\text{[R-Unwrap-Ctor-Indet]} \quad &\quad F \vdash r \Rightarrow r' + A \quad F \vdash C_i^{-1} r \Rightarrow C_i^{-1} r' + A \\
\text{Environment Resumption} \quad &\quad F \vdash E \Rightarrow E' + A \\
\end{align*} \]

Fig. 18. Resumption.
A.4 Example Satisfaction and Checking

Figure 19 defines extensions to example satisfaction and live bidirectional example checking when evaluation is extended with assertions.

**Example Satisfaction** (changes to Figure 6)

\[
\begin{align*}
F &\vdash e \models X \quad F \vdash r \models ex \\
\text{[Sat]} &
\end{align*}
\]

\[
\begin{align*}
\text{Filter}(X) &= (E_1 \vdash \bullet \models ex_1), \ldots, (E_n \vdash \bullet \models ex_n) \\
\{ E_i \vdash e \Rightarrow r_i \Rightarrow A_i \quad F \vdash r_i \Rightarrow r_i' \Rightarrow A_i' \quad F \vdash r_i' \models ex_i \quad F \models A_i \oplus A_i' \}^i\in[n] \\
F &\vdash e \models X \\
\text{[XS-Input-Output]} &
\end{align*}
\]

\[
\begin{align*}
F \vdash r_1 [v_2] \Rightarrow r \oplus A \\
F \vdash r \models ex \\
F \models A \\
F \vdash r_1 \models \{v_2 \rightarrow ex\} \\
\text{Simplify } (A \oplus A') \supseteq K &
\end{align*}
\]

**Live Bidirectional Example Checking** (changes to Figure 7)

\[
\begin{align*}
\Sigma; \Delta; F \vdash e \models X \oplus K \\
\text{[Check]} &
\end{align*}
\]

\[
\begin{align*}
\{ E_i \vdash e \Rightarrow r_i \Rightarrow A_i \quad F \vdash r_i \Rightarrow r_i' \Rightarrow A_i' \quad F \vdash r_i' \Leftarrow ex_i \Rightarrow K_i \quad \text{Simplify}(A_i \oplus A_i') \supseteq K_i' \}^i\in[n] \\
F &\vdash (E_1 \vdash \bullet \models ex_1), \ldots, (E_n \vdash \bullet \models ex_n) \Leftarrow e \oplus K_1 \oplus K_1' \oplus \cdots \oplus K_n \oplus K_n' \\
\text{[U-Case-Guess]} &
\end{align*}
\]

\[
\begin{align*}
j \in [1, n] \\
F' &= \text{Guesses}(\Delta, \Sigma, r) \quad F \oplus F' \vdash r \Rightarrow C_j r' \Rightarrow A \quad \text{Simplify}(A) \supseteq K_0 \\
F \oplus F' \vdash e_j \Leftarrow (E, x_j \mapsto r' \Rightarrow \bullet \models ex) \Rightarrow K &
\end{align*}
\]

\[
F \vdash [E] \text{ case } r \text{ of } \{ C_i x_i \rightarrow e_i \}^i\in[n] \Leftarrow ex \Rightarrow (-; F') \oplus K_0 \oplus K
\]

Fig. 19. Example Satisfaction and Checking. Extended with evaluation and resumption assertions.
A.5 Unevaluation Constraint Merging

Figure 20 defines the merge operations for constraints.

**(Syntactic) Constraint Merging**

\[
F_1 \oplus F_2 = F \\
U_1 \oplus U_2 = U \\
K_1 \oplus K_2 = K
\]

\[
\forall ??_h \in \text{dom}(F_1) \cap \text{dom}(F_2). \ F_1(??_h) = F_2(??_h)
\]

\[
F_1 \oplus F_2 = F_1 + F_2
\]

\[
U_1' = U_1 \setminus \text{dom}(U_2) \quad U_2' = U_2 \setminus \text{dom}(U_1)
\]

\[
U_{12} = \{ \ h \mapsto U_1(??_h) \+ U_2(??_h) \mid ??_h \in \text{dom}(F_1) \cap \text{dom}(F_2) \}
\]

\[
U_1 \oplus U_2 = U_1' \+ U_{12} \+ U_2'
\]

\[
F_1 \oplus F_2 = F' \quad U_1 \oplus U_2 = U'
\]

\[
(U_1; F_1) \oplus (U_2; F_2) = (U'; F')
\]

**(Semantic) Constraint Merging**

\[
\Sigma \Delta; \text{Merge}(K) \triangleright K
\]

\[
F(??_h) = e \quad \Sigma \Delta; F \vdash e \Rightarrow X \triangleleft K
\]

\[
\Sigma \Delta; \text{Resolve}(F; h \mapsto X) \leadsto K
\]

\[
??_h \notin F
\]

\[
\Sigma \Delta; \text{Resolve}(F; h \mapsto X) \leadsto (h \mapsto X; -)
\]

\[
\{ \Sigma \Delta; \text{Resolve}(F; h_1 \mapsto X) \leadsto K' \}_{i \in\{n\}}
\]

\[
\Sigma \Delta; \text{Step}(h_1 \mapsto X_1, \ldots, h_n \mapsto X_n; F) \leadsto K_1' \+ \cdots \+ K_n'
\]

\[
\Sigma \Delta; \text{Step}(K) \leadsto K' \quad K \neq K'
\]

\[
\Sigma \Delta; \text{Merge}(K') \triangleright K''
\]

\[
\Sigma \Delta; \text{Merge}(K) \triangleright K''
\]

\[
\Sigma \Delta; \text{Step}(K) \leadsto K' \quad K = K'
\]

\[
\Sigma \Delta; \text{Merge}(K) \triangleright K'
\]

Fig. 20. Constraint Merging.
A.6 Type-Directed Guessing

Figure 21 defines type-directed guessing rules analogous to expression type rules (Figure 4).

**Type-Directed Guessing**

\[
\Sigma : (\Gamma \vdash \bullet : T) \rightsquigarrow_{\text{guess}} e
\]

\[\Sigma (D)(C) = T \rightarrow D \quad (\Gamma \vdash \bullet : T) \rightsquigarrow_{\text{guess}} e \quad (\Gamma \vdash \bullet : D) \rightsquigarrow_{\text{guess}} C\ e\]

\[
\Sigma (D) = \{ C_i : T_i \rightarrow D \} \quad (\Gamma \vdash \bullet : D) \rightsquigarrow_{\text{guess}} e \quad \{ (\Gamma, x_i:T_i \vdash \bullet : T) \rightsquigarrow_{\text{guess}} e_i \} \quad i \in [n]
\]

\[
(\Gamma \vdash \bullet : T) \rightsquigarrow_{\text{guess}} \text{case of } \{ C_i \ x_i \rightarrow e_i \} \quad i \in [n]
\]

\[
\Gamma(x) = T \quad (\Gamma \vdash \bullet : T) \rightsquigarrow_{\text{guess}} x
\]

\[
\Gamma \vdash \bullet : T_2 \rightarrow T \quad (\Gamma \vdash \bullet : T_2) \rightsquigarrow_{\text{guess}} e_1
\]

\[
\Gamma \vdash \bullet : T_2 \rightarrow T \quad (\Gamma \vdash \bullet : T_2) \rightsquigarrow_{\text{guess}} e_2
\]

\[
\Gamma \vdash \bullet : (T_1, T_2) \rightarrow T \quad (\Gamma \vdash \bullet : (T_1, T_2)) \rightsquigarrow_{\text{guess}} e
\]

\[
(\Gamma \vdash \bullet : T_1) \rightsquigarrow_{\text{guess}} \text{pr}_i \quad i \in [2]
\]

Fig. 21. Type-Directed Guessing.