Program Sketching with Live Bidirectional Evaluation

Anonymous Author(s)

Abstract

We present Sketch-n-Myth, a technique for completing program sketches whereby the evaluation of ordinary assertions gives rise to input-output examples. The key innovation, called *live bidirectional evaluation*, propagates examples "backward" through partially evaluated sketches. Compared to previous example-based synthesis techniques, live bidirectional evaluation enables Sketch-n-Myth to (a) synthesize recursive functions without trace-complete examples and (b) specify and solve interdependent synthesis goals. On benchmarks used to evaluate Myth, a state-of-the-art example-based synthesis tool, Sketch-n-Myth requires on average 67% of the number of examples, without sketching. For many of these benchmarks, a simple sketching strategy further reduces the example burden.

1 Introduction

Recent advances have brought programming-by-example to richly-typed functional programming languages [1, 9, 13, 31]. These techniques—as well as complementary program synthesis techniques that use logical specifications [20, 22, 35]—contribute to ongoing efforts to integrate program synthesis into the general-purpose programming workflow.

Unfortunately, state-of-the-art example-based synthesis techniques suffer from two significant limitations.

Limitation A: Trace-Complete Examples. To synthesize a recursive function, the user (serving as an *oracle* [1]) must provide input-output examples for recursive calls internal to the eventual solution. Providing *trace-complete examples* "proved to be difficult initially" even for experts, and "discovering ways to get around this restriction ... would greatly help in converting this type-directed synthesis style into a usable tool." [31]

Limitation B: Independent, Top-Level Goals. The user must factor all synthesis tasks into completely unimplemented top-level functions, each of which must be equipped directly with (trace-complete) examples. The system attempts to synthesize each of these functions separately.

These limitations preclude natural programming scenarios. For example, Figure 1 shows an incomplete program to "stutter" each element of a list `n` times. The `stutter_n` function itself is complete but depends on an unfinished `replicate` helper function—with *holes*, written ??, denoting missing expressions to be synthesized. The holes in `replicate` are not top-level, and they are constrained indirectly—and interdependently—via two test cases for `stutter_n`, which are not trace-complete.

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

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

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

Figure 1. Program sketching in Sketch-n-Myth.

Existing example-based techniques cannot fill the holes based on these constraints.

Our Approach: Sketch-n-Myth. We present an example-based technique for sketching that addresses these limitations. Holes can appear in arbitrary expression positions, and they are constrained by types and by ordinary `assert` statements which give rise to example constraints. These are solved iteratively using type-and-example-directed synthesis techniques.

The technical challenge to combine program sketching with example-based synthesis is to identify suitable notions of *concrete evaluation* and *example satisfaction*—which form the central *guess-and-check* search strategy—when holes can appear in arbitrary expression positions. Our solution, called *live bidirectional evaluation*, comprises two parts:

1. A *live evaluator* $e \Rightarrow r$ that partially evaluates a sketch e by proceeding around holes, producing a result r which is either a value or a "paused" expression that, when the necessary holes are filled, will continue evaluating safely (an approach adapted from Omar et al. [28]); and
2. A *live unevaluator* $r \Leftarrow ex \dashv K$ that, given a result r to be checked against example ex, computes constraints K—over possibly many holes in the sketch—that, if satisfied, ensure the result will eventually produce a value satisfying ex.

Given the sketch in Figure 1, our implementation synthesizes the desired expressions—shown in boxes—to fill the holes. (Our exposition employs several syntactic conveniences not currently implemented. Differences are described in §5.)

Contributions. This paper extends the theory of *type-and-example-directed synthesis* in Myth [31] to support sketches and live bidirectional evaluation.

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* [28]—checks example satisfaction in the presence of sketches. Live bidirectional evaluation facilitates the core guess-and-check strategy for programs with holes, and can be integrated into other example-based synthesis approaches, e.g., using type refinement [13] and axiomatic deductive reasoning [9]. (§3.5)
- We observe that live bidirectional evaluation can also be used to simplify program assertions into the kinds of constraints required by example-based synthesis techniques. 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 generalize the Myth hole synthesis algorithm to employ live bidirectional evaluation. The resulting algorithm (a) alleviates the trace-completeness requirement and (b) iteratively solves multiple interdependent tasks. (§4.2)

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

To provide empirical validation of our approach, we implement Sketch-n-Myth and perform several experiments.

- We synthesize 37 of 43 tasks from the Myth benchmark suite [29, 31] in Sketch-n-Myth using 67% of the number of examples (without sketches) on average. (§5.1)
- We identify a simple *base case sketching strategy* and apply it systematically to the benchmarks. Base case sketches further reduce the number of examples that Sketch-n-Myth requires to complete many tasks. (§5.2)

The Supplementary Materials include proofs and our tool.

2 Overview

In this section, we work through several small programs to introduce how Sketch-n-Myth: employs live bidirectional evaluation to check example satisfaction of guessed expressions, i.e. sketches in our formulation (§2.1); supports user-defined sketches (§2.2); and derives examples from program `assert`s (§2.3). Then, in §3 and §4, we formally define live bidirectional evaluation and the synthesis pipeline.

We write holes $??_h$ below with explicit names h; our implementation generates these names and hides them from the user. Literals `0`, `1`, `2`, etc. are syntactic sugar for the corresponding naturals of `type Nat = Z | S Nat`. Some judgement forms below are simplified for the current discussion.

2.1 Synthesis Without Trace-Completeness

Consider the task to synthesize `plus : Nat -> Nat -> Nat` given three test cases shown in Figure 2. The resulting *example constraint* $K_0 = (- \vdash \bullet_0 \models \{0\ 1 \rightarrow 1,\ 2\ 0 \rightarrow 2,\ 1\ 2 \rightarrow 3\})$ requires that $??_0$ (hole name 0 generated for the definition of `plus`) be filled with an expression that, in the empty environment, conforms to the given input-output examples.

Given a set of constraints K_h, Sketch-n-Myth employs the *hole synthesis* search procedure $K_h \rightsquigarrow e_h \dashv K'$ to fill the hole $??_h$ with a valid expression e_h, assuming new constraints K' over other holes in the program. Following Myth [31], hole synthesis augments naïve guessing-and-checking with *example-directed refinement* to create partial solutions with independent subgoals. When these cannot be filled by guessing-and-checking, an expression on which to branch is guessed and the examples are *distributed* to subgoals for the branches.

We will describe the following search path, which produces the solution for `plus` shown in Figure 2 boxed in blue.

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

First, refinement synthesizes a recursive function literal, with subgoal $??_1$ for the body. The constraints K_1 (not shown) are created from K_0 by binding the input examples to `m` and `n` in the environments.

Second, after guessing-and-checking fails to solve $??_1$, the expression `m` is guessed to scrutinize, and then `m` is evaluated in each environment of the three constraints in K_1. One constraint in K_2 ($K_{2.1}$, shown below) is distributed to subgoal $??_2$ for the base case branch, and two constraints in K_3 (not shown) are distributed to subgoal $??_3$ for the recursive case.

Third, choosing to work on the recursive branch, refinement synthesizes the literal $\texttt{S}\ ??_4$. By removing a shared constructor head `S` from the output examples in K_3, the new subgoal is constrained by two examples in K_4 (shown below).

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

The fourth and fifth steps fill the remaining subgoals, $??_4$ and $??_2$, via guess-and-check as discussed below.

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

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

When considering `plus m' n`, the name `plus` binds the value

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

comprising the first three fillings and the "current" guess.

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

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

Although the function is incomplete, *live evaluation* [28] resolves two recursive calls to `plus`, before the hole $??_2$ in the base case reaches evaluation position; the resulting *hole closure*, of the form $[E]\,??_h$, captures the environment at that point. Comparing the result to `1` (i.e. `S Z`), unevaluation removes an `S` from each side and creates a new constraint $K_{2.2}$ (shown below) for the base case.

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

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

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

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

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

Notice that the test cases used to synthesize `plus` were *not* trace-complete: live bidirectional example checking recursively called `plus 1 0`, `plus 0 0`, and `plus 0 2`, none of which were included in the examples. Instead, Sketch-n-Myth *generated* additional constraints that the user would be required to provide in prior work [1, 13, 31].

2.2 User-Defined Sketches

Domain knowledge can often be split naturally across a sketch and examples. For instance, the task to compute the maximum of two naturals is apparently difficult for example-based synthesis—Sketch-n-Myth requires 9 examples, 5 of which include zero as one or both arguments. Instead, if the user sketches the zero cases for `max`, as shown in Figure 2, just a few examples are sufficient to complete the recursive case. (The library function `spec2` asserts input-output examples for a binary function.) User-provided sketches are handled in the same way as the internally-created sketches described above.

2.3 Deriving Examples from Assertions

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

```
plus =
  fix plus λm n ->
  case m of {Z -> n; S m' -> S (plus m' n)}

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

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

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

```
odd n =                      unJust mx =
  case n of                    case mx of
    Z       -> False             Nothing -> 0
    S Z     -> True              Just x  -> x
    S S n'' -> odd n''

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

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

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

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

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

Figure 2. Holes ?? (not shown) filled with code in boxes.

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

$$\begin{array}{lll} \texttt{odd}\,(\texttt{unJust}\;??_5) & \longrightarrow^* & \texttt{odd}\,(\texttt{unJust}\;([-]\,??_5)) \\ & \longrightarrow^* & \texttt{odd}\,(\texttt{case}\;([-]\,??_5)\;\mathit{unJust}) \\ & \Rightarrow & \texttt{case}\,(\texttt{case}\;([-]\,??_5)\;\mathit{unJust})\;\mathit{odd} \end{array}$$

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

How can we "indirectly" constrain $??_5$ given that this partially evaluated expression is asserted to be `True`?

Unevaluating Case Expressions. Unevaluation will run each of the three branches of *odd* "in reverse," attempting to reconcile each with the required example, `True`:

$$\texttt{case}\ (\texttt{case}\ ([-]\,??_6)\ \mathit{unJust})\ \mathit{odd} \;\Leftarrow\; \texttt{True} \;\dashv\; (1)(2)(3)$$

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

(2) The second branch expression, `True`, is equal to the example. But to take this branch, unevaluation must ensure that the scrutinee—itself an indeterminate case result, with two branches—will match the pattern `S Z` (i.e. `1`):

$$(\texttt{case}\ ([-]\,??_6)\ \mathit{unJust}) \;\Leftarrow\; \texttt{1} \;\dashv\; (2a)(2b)$$

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

(2b) Reasoning about the second branch expression is more involved: the variable `x` must bind the argument of `Just`, but we have not yet ensured that this branch will be taken! To bridge the gap, we bind `x` to the symbolic, and indeterminate, *inverse constructor application* $\texttt{Just}^{-1}\,([-]\,??_6)$ when evaluating the branch expression; unevaluation "transfers" the resulting example from the symbolic result to the scrutinee:

$$\texttt{x} \;\Rightarrow\; \texttt{Just}^{-1}\,([-]\,??_6) \;\Leftarrow\; \texttt{1} \;\dashv\; (- \vdash \bullet_6 \models \texttt{Just 1})$$

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

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

Altogether, live bidirectional evaluation untangles the interplay between indeterminate branching and assertions. For instance, Sketch-n-Myth can fill the holes in `minus` and `mult` in Figure 2, as well as the `stutter_n` program in Figure 1.

3 Live Bidirectional Evaluation

In this section, we formally define *live evaluation* $E\,;F \vdash e \Rightarrow r$ and *live unevaluation* $F \vdash r \Leftarrow ex \dashv K$ for a calculus called Core Sketch-n-Myth. We choose a natural semantics (big-step, environment-style) presentation [21], though our techniques can be re-formulated for a small-step, substitution-style model. Compared to earlier notation, here we refer to environments E and F—often typeset in light gray, because environments would "fade away" in a substitution-style presentation.

$$\begin{array}{llllll}
\textbf{Datatypes} & D & & \textbf{Variables} & f, x \\
\textbf{Constructors} & C & & \textbf{Hole Names} & h
\end{array}$$

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

Figure 3. Syntax of Core Sketch-n-Myth.

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 adapts the *live programming with holes* technique [28] to our setting; minor technical differences are described in Appendix C. 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 3 defines the syntax of Core Sketch-n-Myth, a calculus of recursive functions, unit, pairs, and (named, recursive) algebraic datatypes. We say "products" to mean unit and pairs.

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

Expressions and Holes. The expression forms on the first three lines are standard function, product, and constructor forms, respectively. The expressions $\texttt{prj}_1\ e$ and $\texttt{prj}_2\ e$ project the first and second components of a pair. 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.

Holes $??_h$ can appear anywhere in expressions (i.e. sketches). We assume that each hole in a sketch has a unique name h. We sometimes write ?? when the name is not referred to. Hole contexts Δ define a *contextual type* $(\Gamma \vdash \bullet : T)$ to describe what expressions can "fill" a given hole [27].

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

The four result forms on the first line of the result grammar would—on their own—correspond to values in a conventional natural semantics (without holes). In Core Sketch-n-Myth, these are *determinate* results that can be eliminated in a type-appropriate position; Appendix A defines a simple predicate r det to identify such results, and type checking is discussed below. Note that a recursive function closure $[E]\,\mathtt{fix}\,f\,(\lambda x.e)$ stores an environment E that binds the free variables of the function body e, except the name f of the function itself.

The four result forms on the second and third lines 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." Appendix A defines the predicate r 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 indet); we require that r_2 be final in accordance with our eager evaluation semantics, discussed below. An indeterminate projection $\mathtt{prj}_{\,i\in[2]}\ r$ appears when the argument has not yet evaluated to a pair (i.e. r indet). An indeterminate case closure $[E]\,\mathtt{case}\ r\ \mathtt{of}\ \{C_i\ x_i \rightarrow e_i\}^{\,i\in[n]}$ appears when the scrutinee has not yet evaluated to a constructor application (i.e. r indet)—like with function and hole closures, the environment E is used when evaluation resumes with the appropriate branch.

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

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

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

Live Evaluation $\boxed{E\,;F \vdash e \Rightarrow r}$

$$\frac{E \vdash e \Rightarrow r \qquad F \vdash r \Rightarrow r'}{E\,;F \vdash e \Rightarrow r'}$$

Expression Evaluation (excerpt) $\boxed{E \vdash e \Rightarrow r}$

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

Resumption (excerpt) $\boxed{F \vdash r \Rightarrow r'}$

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

Figure 4. Evaluation and Resumption (Appendix A).

3.2 Type Checking

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

$$\text{[T-Hole]}\ \frac{\Delta(??_h) = (\Gamma \vdash \bullet : T)}{\Sigma\,;\Delta\,;\Gamma \vdash ??_h : T}$$

3.3 Live Evaluation

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

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

The other three rules, suffixed "-Indet," are counterparts to rules E-App, E-Prj, and E-Case for determinate forms. For example, when a function evaluates to a result r_1 that is not a function closure, the E-App-Indet rule creates the indeterminate application result $r_1\ r_2$. The remaining rules (Appendix A) are similar. Evaluation is deterministic and produces final results; Appendix A formally establishes these propositions.

Resumption. Resumption evaluates results much like expression evaluation. When resuming a closure $[E]\,??_h$ over a hole that F fills with an expression e_h, the R-Hole-Resume rule evaluates e_h in the closure environment, producing a result r. Because e_h may refer to other holes now filled by F, r is recursively resumed to r'.

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

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

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

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

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

(Uneval.) Constraint Satisfaction $\boxed{F \models K}$

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

Figure 5. Example and Constraint Satisfaction.

3.4 Example Satisfaction

Live evaluation partially evaluates a sketch to a result. Figure 5 defines what it means for a result to satisfy an example.

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

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

3.5 Live Unevaluation

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

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

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

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

Figure 5 defines constraint satisfaction $F \models K$ by checking that (i) F subsumes any fillings F_0 in K and (ii) F satisfies the examples X_i for each hole $??_{h_i}$ constrained by K.

Figure 6 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 $\mathit{Merge}(K)$ 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 5) are the U-Top rule to unevaluate any result with $\top$ and the U-Unit, U-Pair, U-Ctor, and U-Fix rules to unevaluate determinate results. Notice that U-Fix refers to bidirectional example checking—evaluation followed by unevaluation—to "test" that a function is consistent with an input-output example.

The base case in which unevaluation generates example constraints is for hole closures $[E]\,??_h$—the U-Hole rule generates the (named) example constraint $\boxed{h \mapsto (E \vdash \bullet \models ex)}$. What remains is to transform "indirect" unevaluation goals for more complex indeterminate results into "direct" examples on holes.

Indeterminate Function Applications. Consider an indeterminate function application $r_1\ r_2$, with the goal to satisfy ex. In general, an arbitrary r_2 may include holes that appear in elimination position when evaluating the application; it is impossible to generate sufficient constraints locally to ensure that the result satisfies ex. We can, however, if r_2 is restricted to simple (first-order) values v_2. For indeterminate applications of the form $r_1\ v_2$, the U-App rule unevaluates the indeterminate function r_1 with the input-output example $\{v_2 \to ex\}$.

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

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

To compute this constraint, the U-Case rule considers each branch j. The first premise unevaluates the scrutinee r with $C_j\ \top$ to the scrutinee r, generating constraints K_1 required for r to produce an application of constructor C_j. If successful, the

(Uneval.) Constraint Merging (Figure 15 of Appendix A) $\boxed{K_1 \oplus K_2 = K}$ $\boxed{\Sigma ; \Delta ; Merge(K) \rhd K'}$

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

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

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

$$\text{[U-Top]}\quad \frac{}{F \vdash r \Leftarrow \top \dashv -} \qquad \text{[U-Unit]}\quad \frac{}{F \vdash () \Leftarrow () \dashv -}$$

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

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

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

$$\text{[U-Case]}\quad \frac{j \in [1, n] \qquad F \vdash r \Leftarrow C_j\, \top \dashv K_1 \qquad F \vdash e_j \rightleftharpoons (E,\, x_j \mapsto C_j^{-1}\, r \vdash \bullet \models ex) \dashv K_2}{F \vdash [E]\, \mathtt{case}\ r\ \mathtt{of}\ \{C_i\, x_i \rightarrow e_i\}^{i \in [n]} \Leftarrow ex \dashv K_1 \oplus K_2} \qquad \text{[U-Inverse-Ctor]}\quad \frac{F \vdash r \Leftarrow C\, ex \dashv K}{F \vdash C^{-1}\, r \Leftarrow ex \dashv K}$$

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

Figure 6. Live Bidirectional Example Checking via Live Unevaluation.

next step is to evaluate the corresponding branch expression e_j and check that it is consistent with the goal ex. However, the argument to the constructor will only be available after all constraints are solved and evaluation resumes.

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

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. As a more efficient approach in situations where the full expressiveness is not needed, the U-Case-Guess rule refers to an uninterpreted predicate $\mathit{Guesses}(\Delta, \Sigma, r)$ that "eagerly" chooses a hole-filling F' that determines the scrutinee r (i.e., resumes r to some constructor application C_j). We describe our concrete implementation of *Guesses* in §6. The U-Case-Guess rule is the source of hole-filling constraints F produced by unevaluation; recall that U-Hole is the source of example constraints, recorded in U.

4 Synthesis Pipeline

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

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

4.1 Constraint Collection

Figure 7 defines a *program* to be an expression followed by an `assert` $(e_1 = e_2)$ statement. Changes for `assert`s in arbitrary expressions are discussed in §6.

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.

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

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

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

Result **C**onsistency $\boxed{r \equiv_A r'}$

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

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

Assertion Satisfaction $\boxed{F \models A}$

$$\frac{\{\, F \vdash r_i \Rightarrow r_i' \qquad \lceil r_i' \rceil = v_i \,\}^{\,i\in[n]}}{F \models \{\, r_i \Rightarrow v_i \,\}^{\,i\in[n]}}$$

Assertion Simplification $\boxed{\mathit{Simplify}(A) \rhd K}$

$$\frac{\{\, r_i \texttt{ final} \qquad - \vdash r_i \Leftarrow \lfloor v_i \rfloor \dashv K_i \,\}^{\,i\in[n]}}{\mathit{Simplify}(\{\, r_i \Rightarrow v_i \,\}^{\,i\in[n]}) \rhd K_1 \oplus \cdots \oplus K_n}$$

Figure 7. Constraint Collection.

Rather than equality, the Eval-and-Assert rule in Figure 7 checks *result consistency*, $r_1 \equiv_A r_2$, a notion of equality modulo assumptions A about indeterminate results. Determinate results are consistent if structurally equal, as checked by the RC-Refl, RC-Pair, and RC-Ctor rules. Indeterminate results r are consistent with simple values v—the RC-Assert-* rules generate *assertion* predicates $r \Rightarrow v$ in such cases.

Figure 7 defines assertion satisfaction $F \models A$: for each assertion $r_i \Rightarrow v_i$ in A, the indeterminate result r_i should resume under filling F and produce the value v_i.

Assertion Simplification. The $\mathit{Simplify}(A)$ procedure in Figure 7 translates assertions $r_i \Rightarrow v_i$ into example constraints via live unevaluation (every simple value constitutes an example).

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

4.2 Constraint Solving

The constraints K, of the form $(U\,;F_0)$, include filled holes F_0 from unevaluation (cf. U-Case-Guess) and a set of U of unfilled holes constrained by examples. Figure 8 defines an algorithm to synthesize expressions for unfilled holes, using Myth-style techniques extended with live bidirectional evaluation.

Iterative Solving. In our formulation, the synthesis of one hole may assume constraints over others. $\mathit{Solve}(U\,;F)$ is thus an iterative procedure that terminates, via Solve-Done, when no unfilled holes remain.

Otherwise, the Solve-One rule chooses an unfilled hole $??_h$ and forms the synthesis goal $(\Gamma \vdash \bullet_h : T \models X)$ from the hole type and example contexts Δ and U. The *hole synthesis* procedure—discussed next—completes the task, generating new constraints K. The new constraints are combined with the existing ones using the semantic *Merge* operation (cf. §3.5), and the resulting constraints K' are recursively solved.

Hole Synthesis. Following Myth [31], the hole synthesis procedure $F\,;(\Gamma \vdash \bullet_h : T \models X) \rightsquigarrow_{\text{fill}} K\,;\Delta'$ augments guessing-and-checking (Guess-and-Check) with example-directed refinement (Refine) and branching (Branch); these rules are discussed in turn below.

In contrast to [31], the Core Sketch-n-Myth formulation (i) refers to the filling F from previous synthesis tasks completed by *Solve*; (ii) may generate example constraints over other holes in the program; (iii) may fill other holes in the program, besides the goal $??_h$; and (iv) includes a rule, Defer, to "fill" the hole with $??_h$ when all examples are top—these constraints are not imposed directly from program assertions, but are created internally by unevaluation.

Guessing-and-Checking. The Guess-and-Check rule uses the *guessing* procedure $(\Gamma \vdash \bullet : T) \rightsquigarrow_{\text{guess}} e$ to generate a well-typed expression without holes. Guessing amounts to straightforward inversion of expression type checking rules; Appendix A provides the full definition. The candidate expression e is checked for example consistency using live bidirectional example checking (cf. Figure 6). The resulting constraints K are the source of the aforementioned differences (i), (ii), and (iii) compared to the Myth hole synthesis procedure.

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

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

For function-type goals, the Refine-Fix rule synthesizes the function sketch $\texttt{fix}\, f\,(\lambda x.\,??_{h_1})$. The environments inside example constraints X_1 for the function body $??_{h_1}$ bind f to this

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

$$\textsc{[Solve-Done]}\quad \frac{}{\Sigma\,;\Delta\,;\mathit{Solve}(-\,;F) \rightsquigarrow F\,;\Delta}$$

$$\textsc{[Solve-One]}\quad \frac{\begin{array}{c} h \in \mathit{dom}(U) \qquad \Delta(h) = (\Gamma \vdash \bullet : T) \qquad U(h) = X \qquad F\,;(\Gamma \vdash \bullet_h : T \models X) \rightsquigarrow_{\text{fill}} K\,;\Delta' \\ \Sigma\,;\Delta +\!\!+ \Delta'\,;\mathit{Merge}((U \setminus h\,;F) \oplus K) \rhd K' \qquad \Sigma\,;\Delta +\!\!+ \Delta'\,;\mathit{Solve}(K') \rightsquigarrow F'\,;\Delta'' \end{array}}{\Sigma\,;\Delta\,;\mathit{Solve}(U\,;F) \rightsquigarrow F'\,;\Delta''}$$

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

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

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

$$\textsc{[Refine, Branch]}\quad \frac{(\Gamma \vdash \bullet : T \models X) \rightsquigarrow_{\{\text{refine},\text{branch}\}} e \dashv \{\, (\Gamma_i \vdash \bullet_{h_i} : T_i \models X_i)\,\}^{i \in [n]} \qquad \Delta' = \{\, h_i \mapsto (\Gamma_i \vdash \bullet : T_i)\,\}^{i \in [n]}}{F\,;(\Gamma \vdash \bullet_h : T \models X) \rightsquigarrow_{\text{fill}} ((h_1 \mapsto X_1,\, \ldots,\, h_n \mapsto X_n)\,;h \mapsto e)\,;\Delta'}$$

Type-Directed Guessing (Figure 16 of Appendix A) $\boxed{\Sigma\,;(\Gamma \vdash \bullet : T) \rightsquigarrow_{\text{guess}} e}$

Type-and-Example-Directed Refinement $\boxed{\Sigma\,;\Delta\,;(\Gamma \vdash \bullet : T \models X) \rightsquigarrow_{\text{refine}} e \dashv G}$

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

$$\textsc{[Refine-Unit]}\quad \frac{\mathit{Filter}(X) = (E_1 \vdash \bullet \models ()),\, \ldots,\, (E_n \vdash \bullet \models ())}{(\Gamma \vdash \bullet : () \models X) \rightsquigarrow_{\text{refine}} () \dashv -}$$

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

$$\textsc{[Refine-Ctor]}\quad \frac{\begin{array}{c} \mathit{Filter}(X) = \{\, (E_j \vdash \bullet \models C\; \mathit{ex}_j)\,\}^{j \in [m]} \qquad \Sigma(D)(C) = T \\ \textit{New Goal} \\ \boxed{\begin{array}{c} h_1 \text{ fresh} \qquad G_1 = (\Gamma \vdash \bullet_{h_1} : T \models X_1) \\ X_1 = (E_1 \vdash \bullet \models \mathit{ex}_1),\, \ldots,\, (E_m \vdash \bullet \models \mathit{ex}_m) \end{array}} \end{array}}{(\Gamma \vdash \bullet : D \models X) \rightsquigarrow_{\text{refine}} C\; ??_{h_1} \dashv G_1}$$

$$\textsc{[Refine-Fix]}\quad \frac{\begin{array}{c} \mathit{Filter}(X) = (E_1 \vdash \bullet \models \{v_1 \to \mathit{ex}_1\}),\, \ldots,\, (E_m \vdash \bullet \models \{v_m \to \mathit{ex}_m\}) \\ \textit{New Goal} \\ \boxed{\begin{array}{c} h_1 \text{ fresh} \qquad e = \mathtt{fix}\, f\, (\lambda x.??_{h_1}) \qquad G_1 = (\Gamma,\, f{:}T_1 \to T_2,\, x{:}T_1 \vdash \bullet_{h_1} : T_2 \models X_1) \\ X_1 = (E_1,\, f \mapsto [E_1]e,\, x \mapsto \lfloor v_1 \rfloor \vdash \bullet \models \mathit{ex}_1),\, \ldots,\, (E_m,\, f \mapsto [E_m]e,\, x \mapsto \lfloor v_m \rfloor \vdash \bullet \models \mathit{ex}_m) \end{array}} \end{array}}{(\Gamma \vdash \bullet : T_1 \to T_2 \models X) \rightsquigarrow_{\text{refine}} e \dashv G_1}$$

Type-and-Example-Directed Branching $\boxed{\Sigma\,;\Delta\,;(\Gamma \vdash \bullet : T \models X) \rightsquigarrow_{\text{branch}} e \dashv G}$

$$\textsc{[Branch-Case]}\quad \frac{\begin{array}{c} \Sigma(D) = \{C_i\; T_i\}^{i \in [n]} \qquad (\Gamma \vdash \bullet : D) \rightsquigarrow_{\text{guess}} e \qquad \mathit{Filter}(X) = X_1 +\!\!+ \cdots +\!\!+ X_n \\ \textit{New Goals, } i = 1, 2, \ldots, n \\ \boxed{\begin{array}{c} h_i \text{ fresh} \qquad G_i = (\Gamma,\, x_i{:}T_i \vdash \bullet_{h_i} : T \models X_i) \\ X_i = \{\, (E,\, x_i \mapsto r \vdash \bullet \models \mathit{ex}) \mid (E \vdash \bullet \models \mathit{ex}) \in \mathit{Filter}(X) \wedge\; E \vdash e \Rightarrow C_i\; r \dashv - \,\} \end{array}} \end{array}}{(\Gamma \vdash \bullet : T \models X) \rightsquigarrow_{\text{branch}} \mathtt{case}\; e\; \mathtt{of}\; \{C_i\; x_i \to ??_{h_i}\}^{i \in [n]} \dashv G_1,\, \ldots,\, G_n}$$

Figure 8. Constraint Solving with Guessing, Refinement, and Branching.

function sketch (closed by the appropriate environments E_i). As a result, any recursive calls to f will evaluate to closures of $??_{h_1}$, to be constrained by live bidirectional example checking and thus avoiding the need for trace-complete examples.[1]

Branching. Lastly, the Branch rule refers to the *branching* procedure $(\Gamma \vdash \bullet : T \models X) \rightsquigarrow_{\text{branch}} e \dashv G$ to guess an expression on which to branch. The signature of the branching procedure is the same as 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 and gathering those which share the constructor head C_i. The $\mathit{Filter}(X)$ premise ensures that every example is distributed to the subgoal for some branch.

The Branch-Case rule 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 $C_i\ \top$. We choose not to introduce this additional source of expressiveness and non-determinism here.

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

5 Implementation and Experiments

We implemented Sketch-n-Myth synthesis as an OCaml server, and extended the Sketch-n-Sketch bidirectional programming system [7, 25] to interface with the server. Our server and extensions consist of approximately 3,400 lines of OCaml code and 2,000 lines of Elm code, respectively.

Compared to the core language, our implementation supports Haskell/Elm-like syntax, n-ary tuples, `let`-bindings, and `let`-bound recursive function definitions. Our implementation also supports higher-order function examples, following [31]; this feature is orthogonal to the extensions in our work.

Our prototype lacks many of the syntactic conveniences used in code listings in §1 and §2, such as nested pattern matching, infix list operators `(::)` and `(++)`, and type inference for holes. Moreover, we do not support recursive functions whose first argument is not structurally decreasing. These are not fundamental challenges, but they result in slightly different code than shown in the paper.

Optimizations. We adopt two primary optimizations from Myth [29, 31]. The first is to guess and cache only *proof relevant* [4] elimination forms—variables x or calls $f\ e_1 \cdots e_n$ to variable-bound functions. The second is a *staging* approach to incrementally increase the maximum branching depth, the size of terms to guess as scrutinees, and the size of terms to guess in other goal positions. We generally use the same parameters as in [29], but with additional intermediate stages so that small solutions are found more quickly.

To rein in the non-determinism of case unevaluation, we bound the number of nested uses of the "lazy" U-Case rule, and we implement $\mathit{Guesses}(\Delta, \Sigma, r)$ to guess only small terms for the "eager" U-Case-Guess rule.

Experimental Design. In addition to our qualitative analysis of Sketch-n-Myth through programs in §1 and §2, we evaluated Sketch-n-Myth quantitatively on the set of benchmarks used to evaluate Myth.

First, we describe a baseline experiment which evaluates Sketch-n-Myth on the Myth benchmarks using the "full" examples reported by Osera [29] (§ 5.1). Then, we describe two experiments to measure whether or not Sketch-n-Myth requires fewer examples to synthesize these tasks than Myth (a) because trace-complete examples are not required (§ 5.2) and (b) when the user provides a partial implementation (§5.3). All experiments were run on a Mid 2012 MacBook Pro with a 2.5 GHz Intel Core i5 CPU and 16 GB of RAM.

5.1 Experiment 1: Full Examples

The first column of Figure 9 indicates that Sketch-n-Myth passes 37 of the same 43 benchmarks (without sketches) in a similar amount of time (cf. [29]).

Inside-Out Recursion. Of the remaining 6 not successfully synthesized in our implementation, Myth finds four solutions with *inside-out recursion* [29], which pattern match on a recursive call to the function being synthesized. Inside-out solutions are smaller than more natural ones, and sometimes they are the only solutions—because only elimination forms are guessed and because let-bindings are not synthesized [29].

Among these four, Sketch-n-Myth terminates with zero solutions (marked "none" in Figure 9) for `list_compress` and `list_pairwise_swap`; terminates with an overspecialized solution (marked "overspec") for `list_even_parity`; and does not terminate within 120 seconds (marked "timeout") for `tree_postorder`.

"Turning the Branch-Case knob" (cf. §4.2) ought to provide the necessary expressiveness for inside-out recursion, but the additional non-determinism may need to be tamed.

Remaining Benchmarks. Sketch-n-Myth does not terminate within 120 seconds on the remaining two benchmarks (`tree_insert` and `tree_nodes_at_level`). We have not determined the exact cause.

One major optimization in Myth that we have not implemented is refinement trees [31], which serve to cache introduction forms synthesized by the refinement procedure. This

[1] The refinement rule for recursive functions in [31] would, for the program in § 2.1, bind `plus` in $K_{4.1}$ and $K_{4.2}$ to trace-complete examples $\{0\ 1 \rightarrow 1\ ,\ 2\ 0 \rightarrow 2\ ,\ 1\ 2 \rightarrow 3\ ,\ \ldots\}$. In addition to usability implications of trace-completeness, the theory is complicated by a non-standard *value compatibility* notion [31, §3.3] to approximate value equality, because input-output examples serve as a "lookup table" to resolve recursive calls.

Experiment	**1**		**2**	**3**
Sketch	*None*			*Base Case*
#Benchmarks	*37/43 Myth benchmarks*			*24/37*
Objective	*Top-1*		*Top-1*	*Top-1-R*
Name	**#Ex**	**Time**	**#Ex**	**#Ex**
bool_band	4	0.003	3 (75%)	—
bool_bor	4	0.003	3 (75%)	—
bool_impl	4	0.004	3 (75%)	—
bool_neg	2	0.001	2 (100%)	—
bool_xor	4	0.007	3 (75%)	—
list_append	6	0.006	5 (83%)	1 (17%)
list_compress	13	none	—	—
list_concat	6	0.007	3 (50%)	2 (33%)
list_drop	11	0.025	5 (45%)	2 (18%)
list_even_parity	7	overspec	—	—
list_filter	8	0.092	4 (50%)	overspec
list_fold	9	0.697	3 (33%)	3 (33%)
list_hd	3	0.002	2 (67%)	—
list_inc	4	0.011	2 (50%)	—
list_last	6	0.006	4 (67%)	2 (33%)
list_length	3	0.002	3 (100%)	1 (33%)
list_map	8	0.036	4 (50%)	2 (25%)
list_nth	13	0.108	6 (46%)	2 (15%)
list_pairwise_swap	7	none	—	—
list_rev_append	5	0.094	3 (60%)	2 (40%)
list_rev_fold	5	0.028	2 (40%)	—
list_rev_snoc	5	0.008	3 (60%)	1 (20%)
list_rev_tailcall	8	0.006	3 (38%)	1 (13%)
list_snoc	8	0.012	4 (50%)	2 (25%)
list_sort_sorted_insert	7	0.012	3 (43%)	1 (14%)
list_sorted_insert	12	5.557	7 (58%)	overspec
list_stutter	3	0.002	2 (67%)	1 (33%)
list_sum	3	0.021	2 (67%)	—
list_take	12	0.061	6 (50%)	3 (25%)
list_tl	3	0.002	2 (67%)	—
nat_add	9	0.005	4 (44%)	1 (11%)
nat_iseven	4	0.003	3 (75%)	2 (50%)
nat_max	9	0.035	9 (100%)	4 (44%)
nat_pred	3	0.001	2 (67%)	—
tree_binsert	20	timeout	—	—
tree_collect_leaves	6	0.062	3 (50%)	2 (33%)
tree_count_leaves	7	2.885	3 (43%)	1 (14%)
tree_count_nodes	6	0.292	3 (50%)	2 (33%)
tree_inorder	5	0.101	4 (80%)	2 (40%)
tree_map	7	0.048	4 (57%)	3 (43%)
tree_nodes_at_level	11	timeout	—	—
tree_postorder	20	timeout	—	—
tree_preorder	5	0.126	3 (60%)	2 (40%)
Averages			61%*	29%

Figure 9. Experiments.

(1) "Full" Examples required by Myth [29]; No Sketch.
(2) Examples required by Sketch-n-Myth; No Sketch.
(2) Examples required by Sketch-n-Myth; Base Case Sketch.

Top-1: *1st solution valid.* ***Top-1-R***: *1st recursive solution valid.* ***#Ex***: *Percentage compared to baseline #examples in parentheses.* ***Time***: *Avg. of 5 runs, in seconds.* ***Averages***: *Non-blank rows. 61% for 37 benchmarks. (Upper bound: 67% for all 43.)*

optimization does not directly carry over to our setting, because in Sketch-n-Myth refinement may introduce different assumptions across different branches of search. One hypothesis is that suitably extending refinement trees to our setting could help synthesize the remaining tasks.

5.2 Experiment 2: Fewer Examples (without Sketches)

To determine how many fewer examples Sketch-n-Myth needs compared to Myth on the 37 benchmarks we synthesize, for each one we manually removed sets of examples from the full test suite, until Sketch-n-Myth no longer synthesized a correct solution (i.e. that conforms to the full examples).

Of the 37 benchmarks, Figure 9 shows that Sketch-n-Myth required fewer examples to synthesize all but three benchmarks (`bool_neg`, `list_length` and `nat_max`)—on average 61% of the number of examples—with similar running times as in the baseline configuration (Appendix B shows timing data).

If Sketch-n-Myth were extended with the trace-complete approach to synthesizing recursive functions as a backup—and thus that the 6 missing benchmarks would require 100% of the full examples—Sketch-n-Myth requires 67% of the number of examples on average for the entire benchmark suite.

For single, top-level holes, the major difference between the algorithms underlying Core Sketch-n-Myth and Myth lies in the synthesis of recursive functions (cf. the discussion of Refine-Fix in § 4.2). Qualitatively, many examples that we removed were inner calls for larger input-output examples. The reduction in examples for these benchmarks in Sketch-n-Myth can likely be attributed to not requiring trace-complete examples. However, because Sketch-n-Myth and Myth are separate implementations, likely with many incidental differences in search order, this experiment cannot rule out other factors that might contribute to the difference.

5.3 Experiment 3: Base Case Sketching Strategy

Recall the `max` program (§ 2.2), where sketching the base cases allowed the recursive case to be synthesized with many fewer examples compared to synthesizing the function entirely.

We measured the effectiveness of this *base case sketch strategy*—performing case analysis on the correct argument of the function, filling in the base case properly, and leaving a hole in the recursive branch—by systematically applying it to the Myth benchmarks.

As for the previous experiment, we manually removed sets of examples until Sketch-n-Myth no longer successfully completed the task. For this experiment, however, because the base case strategy pertains to recursive functions, we considered a task successful if the first (roughly smallest) *recursive* solution was correct, rather than simply the first (roughly smallest) solution overall. Figure 9 shows the results of this experiment on the 26 of 37 benchmarks that are recursive.

For 24 of these 26 benchmarks, Sketch-n-Myth required significantly fewer examples after having been provided the base case. On average, 29% of the full examples were needed—compared to 58% without a sketch (average, not shown, of 24 rows in the Experiment 2 column). No benchmark program for which this strategy was successful required more than 4 examples, with a mean of 1.88.

The base case sketch strategy was unsuccessful for two benchmarks, `list_filter` and `list_sorted_insert`. In each case, Sketch-n-Myth synthesized an overspecialized solution even with the full examples. The staging parameters increase branching depth before scrutinee size, and a relatively larger scrutinee is needed for the desired solution. Compared to when no sketch is provided, the staging parameters effectively "penalize" the sketch for having introduced a case. Future work may consider reconfiguring the staging parameters to account for the structure of user-provided sketches.

6 Discussion

The experiments in § 5.2 demonstrate that Sketch-n-Myth synthesizes prior benchmarks with fewer examples. The programs and experiments in § 1, § 2, and § 5.3 demonstrate that example-based synthesis can complete small sketches.

6.1 Limitations and Future Work

Nevertheless, several limitations of Sketch-n-Myth need to be addressed to further advance the goal for synthesis to become a tool for practical programming.

Usability. Like any approach based on examples, one challenge is how to select which input-output examples to provide. Though Sketch-n-Myth eliminates the trace-completeness requirement, small changes to the examples often lead to very different performance both in time and solution quality.

We could measure performance on randomly generated inputs as a proxy for usability [9], but the software engineering implications of synthesis techniques remain a largely unexplored topic—see [36] for one recent effort.

Osera [30] describes how a synthesis tool might interact with the user to help choose which search paths to explore. One can imagine also suggesting sketch strategies—we described one simple strategy; additional ones might be identified from existing code bases and edit histories—and allowing users to label desirable and undesirable parts of candidate solutions [32].

Scalability. In our experiments, each task was run with the minimal required context. Orthogonal techniques for scaling to large contexts with additional components [8, 17, 18] could be incorporated into our approach.

Unevaluation in Sketch-n-Myth introduces a new, significant source of non-determinism. To scale to much larger programs with complex control flow, static reasoning—interleaved with concrete evaluation—could be used to prune unsatisfiable, or heuristically "difficult," sets of example constraints.

Assertions. To allow `assert`s in arbitrary expressions, evaluation and resumption could be extended to generate assertions A as a side-effect, to be translated by *Simplify* into constraints for synthesis. We expect the algorithmic changes to be straightforward, but the extended definition of assertion satisfaction—and corresponding correctness properties—is a bit more delicate; Appendix C provides more discussion.

6.2 Related Work

Program synthesis is a large and active research area; Gulwani et al. [16] provide a recent survey of developments. We discussed technical differences between Sketch-n-Myth and Myth throughout the paper. Below, we discuss other closely related work. Appendix C describes technical differences in our formulation of live evaluation [28], as well as to more loosely related work on *bidirectional evaluation* [24, 25, 33].

Program Sketching. The sketching approach to synthesis was pioneered in Sketch [39–42], an imperative, C-like language in which holes ?? are completed at compile-time. Holes in Sketch can be used to encode syntax constraints to define grammars of desired completions [2]; it would be interesting to extend Sketch-n-Myth with such facilities.

Rosette [44, 45] is an untyped functional language based on Racket [10] in which holes are generated dynamically through a combination of concrete and symbolic execution; the program demands hole completions later during evaluation. Holes in Sketch and Rosette range over integer-typed expressions.

For richly-typed functional languages, Synquid [35] and Leon [22] synthesize recursive functions using solver-based techniques driven by logical specifications (e.g. SMT-based refinement types in Synquid). These techniques thus support sketching naturally.

Example-based and logic-based specifications are complementary: examples may allow for smaller or simpler specifications for functions which expose some representation details to clients, while logical specifications fare better when behavior is more abstract [35, § 4.3]. Combining these techniques is another interesting direction for future work.

Programming-by-Example. Several techniques employ examples to synthesize recursive programs in functional languages. Escher [1] is an untyped approach. Frankle et al. [13] recast examples in a type language of singletons, intersections, and unions, and their implementation allows symbolic values in examples—to help synthesize polymorphic functions. These approaches require trace-complete examples. Rather than trace-completeness, λ^2 [9] relies on deductive reasoning about `list` combinators (e.g. `map`).

Programming-by-example techniques have also been developed for numerous application domains, including string transformations [14] (including bidirectional ones [26]), shell scripting [15], web scraping [5], and generating vector graphics [19]. These approaches generally synthesize entire programs. To allow experts to provide partial implementations, it should be possible to formulate notions of live bidirectional evaluation of these domain-specific techniques.

References

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

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

[3] 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 (FTfJP)*.

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

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

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

[7] 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)*.

[8] 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)*.

[9] 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)*.

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

[11] 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).

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

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

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

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

[16] 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

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

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

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

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

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

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

[23] Xavier Leroy and Hervé Grall. 2009. Coinductive Big-step Operational Semantics. *Information and Computation* (2009).

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

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

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

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

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

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

[30] Peter-Michael Osera. 2016. Programming Assistance for Type-Directed Programming (Extended Abstract). In *Type-Driven Development (TyDe)*.

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

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

[33] 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)*.

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

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

[36] Mark Santolucito, Drew Goldman, Allyson Weseley, and Ruzica Piskac. 2018. Programming by Example: Efficient, but Not "Helpful". In *Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU)*.

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

[38] 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)*.

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

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

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

[42] 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)*.

[43] Warren Teitelman. 1972. Automated Programmering: The Programmer's Assistant. In *AFIPS Fall Joint Computing Conference (2)*. 917–921.

[44] 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!)*.

[45] 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 Additional Definitions and Proofs

This section provides additional definitions for §3 and §4, as well as soundness theorems and proofs.

A.1 Syntax

Datatypes. Rather than supporting arbitrary-arity constructors—as in the technical formulation of Osera and Zdancewic [31]—we choose single-arity constructors and products—following the formulation by Frankle [12]—to lighten the presentation of synthesis in §4.

Results. Figure 10 defines result classification.

Final Results and Environments $\boxed{r \ \mathtt{final}}$ $\boxed{E \ \mathtt{final}}$

$$\frac{r \ \mathtt{det}}{r \ \mathtt{final}} \qquad \frac{r \ \mathtt{indet}}{r \ \mathtt{final}} \qquad \frac{}{- \ \mathtt{final}} \qquad \frac{E \ \mathtt{final} \qquad r \ \mathtt{final}}{E, x \mapsto r \ \mathtt{final}}$$

Determinate Results $\boxed{r \ \mathtt{det}}$

$$\frac{}{() \ \mathtt{det}} \qquad \frac{\{ r_i \ \mathtt{final} \}^{i \in [2]}}{(r_1\,,\, r_2) \ \mathtt{det}} \qquad \frac{r \ \mathtt{final}}{C\, r \ \mathtt{det}} \qquad \frac{E \ \mathtt{final}}{[E]\, \mathtt{fix}\, f\, (\lambda x.e) \ \mathtt{det}}$$

Indeterminate Results $\boxed{r \ \mathtt{indet}}$

$$\frac{E \ \mathtt{final}}{[E]\, ??_h \ \mathtt{indet}} \qquad \frac{r_1 \ \mathtt{indet} \qquad r_2 \ \mathtt{final}}{r_1\, r_2 \ \mathtt{indet}} \qquad \frac{r \ \mathtt{indet}}{\mathtt{prj}_{\, i \in [2]}\; r \ \mathtt{indet}} \qquad \frac{E \ \mathtt{final} \qquad r \ \mathtt{indet}}{[E]\, \mathtt{case}\; r \;\mathtt{of}\; \{ C_i\, x_i \rightarrow e_i \}^{i \in [n]} \ \mathtt{indet}}$$

Figure 10. Result Classification. Final results are determinate or indeterminate.

Examples. We define three simple functions below. The coercion $\lfloor v \rfloor$ "upcasts" a simple value to a result. The coercion $\lceil r \rceil = v$ "downcasts" a result to a simple value. The $Filter(X)$ function removes top example constraints.

$$\frac{}{\lceil () \rceil = ()} \qquad \frac{\lceil r_1 \rceil = v_1 \qquad \lceil r_2 \rceil = v_2}{\lceil (r_1\,,\, r_2) \rceil = (v_1\,,\, v_2)} \qquad \frac{\lceil r \rceil = v}{\lceil C\, r \rceil = C\, v} \qquad Filter(X) = \{ (E \vdash \bullet \models ex) \in X \ \mid \ ex \neq \top \}$$

A.2 Type Checking

Figure 11 defines type checking for expressions, results, and examples. The result type checking $\Sigma ; \Delta \vdash r : T$ and example type checking $\Sigma ; \Delta \vdash ex : T$ judgements do not require a type context Γ, 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. Figure 12 defines type checking for constraints, solutions, programs, and assertions.

Expression Typing $\boxed{\Sigma ; \Delta ; \Gamma \vdash e : T}$

$$\text{[T-Fix]}\ \frac{\Sigma ; \Delta ; \Gamma,\, f : T_1 \to T_2,\, x : T_1 \vdash e : T_2}{\Sigma ; \Delta ; \Gamma \vdash \mathtt{fix}\, f\, (\lambda x.e) : T_1 \to T_2} \qquad \text{[T-Var]}\ \frac{\Gamma(x) = T}{\Sigma ; \Delta ; \Gamma \vdash x : T} \qquad \text{[T-Hole]}\ \frac{\Delta(??_h) = (\Gamma \vdash \bullet : T)}{\Sigma ; \Delta ; \Gamma \vdash ??_h : T}$$

$$\text{[T-Unit]}\ \frac{}{\Sigma ; \Delta ; \Gamma \vdash () : ()} \qquad \text{[T-Pair]}\ \frac{\{\, \Sigma ; \Delta ; \Gamma \vdash e_i : T_i \,\}^{\,i \in [2]}}{\Sigma ; \Delta ; \Gamma \vdash (e_1 \,,\, e_2) : (T_1 \,,\, T_2)} \qquad \text{[T-Ctor]}\ \frac{\Sigma(D)(C) = T \qquad \Sigma ; \Delta ; \Gamma \vdash e : T}{\Sigma ; \Delta ; \Gamma \vdash C\, e : D}$$

$$\text{[T-App]}\ \frac{\begin{array}{c}\Sigma ; \Delta ; \Gamma \vdash e_1 : T_2 \to T \\ \Sigma ; \Delta ; \Gamma \vdash e_2 : T_2\end{array}}{\Sigma ; \Delta ; \Gamma \vdash e_1\, e_2 : T} \qquad \text{[T-Prj]}\ \frac{\Sigma ; \Delta ; \Gamma \vdash e : (T_1 \,,\, T_2)}{\Sigma ; \Delta ; \Gamma \vdash \mathtt{prj}_{\,i \in [2]}\; e : T_i} \qquad \text{[T-Case]}\ \frac{\begin{array}{c}\Sigma ; \Delta ; \Gamma \vdash e : D \\ \Sigma(D) = \{C_i\ T_i\}^{\,i \in [n]} \\ \{\, \Sigma ; \Delta ; \Gamma,\, x_i : T_i \vdash e_i : T \,\}^{\,i \in [n]}\end{array}}{\Sigma ; \Delta ; \Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ \{C_i\ x_i \to e_i\}^{\,i \in [n]} : T}$$

Result Typing $\boxed{\Sigma ; \Delta \vdash r : T}$

$$\text{[RT-Fix]}\ \frac{\Sigma ; \Delta \vdash E : \Gamma \qquad \Sigma ; \Delta ; \Gamma \vdash \mathtt{fix}\, f\, (\lambda x.e) : T}{\Sigma ; \Delta \vdash [E]\, \mathtt{fix}\, f\, (\lambda x.e) : T} \qquad \text{[RT-Hole]}\ \frac{\Delta(??_h) = (\Gamma \vdash \bullet : T) \qquad \Sigma ; \Delta \vdash E : \Gamma}{\Sigma ; \Delta \vdash [E]\, ??_h : T}$$

$$\text{[RT-Unit]}\ \frac{}{\Sigma ; \Delta \vdash () : ()} \qquad \text{[RT-Pair]}\ \frac{\{\, \Sigma ; \Delta \vdash r_i : T_i \,\}^{\,i \in [2]}}{\Sigma ; \Delta \vdash (r_1 \,,\, r_2) : (T_1 \,,\, T_2)} \qquad \text{[RT-Ctor]}\ \frac{\Sigma(D)(C) = T \qquad \Sigma ; \Delta \vdash r : T}{\Sigma ; \Delta \vdash C\, r : D}$$

$$\text{[RT-App]}\ \frac{\begin{array}{c}\Sigma ; \Delta \vdash r_1 : T_2 \to T \\ \Sigma ; \Delta \vdash r_2 : T_2\end{array}}{\Sigma ; \Delta \vdash r_1\, r_2 : T} \qquad \text{[RT-Prj]}\ \frac{\Sigma ; \Delta \vdash r : (T_1 \,,\, T_2)}{\Sigma ; \Delta \vdash \mathtt{prj}_{\,i \in [2]}\; r : T_i} \qquad \text{[RT-Case]}\ \frac{\begin{array}{c}\Sigma ; \Delta \vdash r : D \qquad \Sigma(D) = \{C_i\ T_i\}^{\,i \in [n]} \\ \Sigma ; \Delta \vdash E : \Gamma \qquad \{\, \Sigma ; \Delta ; \Gamma,\, x_i : T_i \vdash e_i : T \,\}^{\,i \in [n]}\end{array}}{\Sigma ; \Delta \vdash [E]\, \mathsf{case}\ r\ \mathsf{of}\ \{C_i\ x_i \to e_i\}^{\,i \in [n]} : T}$$

Environment Typing $\boxed{\Sigma ; \Delta \vdash E : \Gamma}$

$$\frac{}{\Sigma ; \Delta \vdash - : -} \qquad \frac{\Sigma ; \Delta \vdash E : \Gamma \qquad \Sigma ; \Delta \vdash r : T}{\Sigma ; \Delta \vdash (E,\, x \mapsto r) : (\Gamma,\, x : T)}$$

Example Typing $\boxed{\Sigma ; \Delta \vdash ex : T}$

$$\text{[XT-Unit]}\ \frac{}{\Sigma ; \Delta \vdash () : ()} \qquad \text{[XT-Pair]}\ \frac{\{\, \Sigma ; \Delta \vdash ex_i : T_i \,\}^{\,i \in [2]}}{\Sigma ; \Delta \vdash (ex_1 \,,\, ex_2) : (T_1 \,,\, T_2)} \qquad \text{[XT-Ctor]}\ \frac{\begin{array}{c}\Sigma(D)(C) = T \\ \Sigma ; \Delta \vdash ex : T\end{array}}{\Sigma ; \Delta \vdash C\, ex : D} \qquad \text{[XT-Top]}\ \frac{}{\Sigma ; \Delta \vdash \top : T} \qquad \text{[XT-Input-Output]}\ \frac{\begin{array}{c}\Sigma ; \Delta \vdash \lfloor v \rfloor : T_1 \\ \Sigma ; \Delta \vdash ex : T_2\end{array}}{\Sigma ; \Delta \vdash \{v \to ex\} : T_1 \to T_2}$$

Figure 11. Expression, Result, and Example Type Checking.

Example Constraints, Unsolved Constraints, and Solution Typing $\boxed{\Sigma ; \Delta \vdash X : \Gamma ; T}$ $\boxed{\Sigma \vdash U : \Delta}$ $\boxed{\Sigma \vdash F : \Delta}$

$$\frac{\{\, \Sigma ; \Delta \vdash E_i : \Gamma \qquad \Sigma ; \Delta \vdash ex_i : T \,\}^{i \in [n]}}{\Sigma ; \Delta \vdash (E_1 \vdash \bullet \models ex_1), \ldots, (E_n \vdash \bullet \models ex_n) : \Gamma ; T} \qquad \frac{\{\, \Delta(??_{h_i}) = (\Gamma_i \vdash \bullet : T_i) \qquad \Sigma ; \Delta \vdash X_i : \Gamma_i ; T_i \,\}^{i \in [n]}}{\Sigma \vdash (h_1 \mapsto X_1, \ldots, h_n \mapsto X_n) : \Delta}$$

$$\frac{\{\, \Delta(??_{h_i}) = (\Gamma_i \vdash \bullet : T_i) \qquad \Sigma ; \Delta ; \Gamma_i \vdash e_i : T_i \,\}^{i \in [n]}}{\Sigma \vdash (h_1 \mapsto e_1, \ldots, h_n \mapsto e_n) : \Delta}$$

Program and Assertion Typing $\boxed{\Sigma ; \Delta \vdash p : T ; T'}$ $\boxed{\Sigma \vdash A : \Delta}$

$$\frac{\Sigma ; \Delta ; - \vdash e : T \qquad \{\, \Sigma ; \Delta ; (\mathtt{main}{:}T) \vdash e_i : T' \,\}^{i \in [2]}}{\Sigma ; \Delta \vdash \mathtt{let}\ \mathtt{main} = e\ \mathtt{in}\ \mathtt{assert}\ (e_1 = e_2) : T ; T'} \qquad \frac{\{\, \exists T \qquad \Sigma ; \Delta \vdash r_i : T \qquad \Sigma ; \Delta \vdash v_i : T \,\}^{i \in [n]}}{\Sigma \vdash \{\, r_i \Rightarrow v_i \,\}^{i \in [n]} : \Delta}$$

Figure 12. Constraint, Solution, Program, and Assertion Type Checking.

A.3 Type Soundness

The progress property is complicated by the fact that, in a big-step semantics, non-terminating computations are not necessarily distinguished from stuck ones [23]. Using a technique similar to that described by Ancona [3], we augment evaluation with a natural k that limits the beta-reduction depth of an evaluation derivation. The augmented evaluation judgment $E \vdash e \Rightarrow_k r$ (Figure 13) asserts that evaluation produced a particular result or that it reached the specified depth before doing so.

Figure 13 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.

Augmented Evaluation $\boxed{E \vdash e \Rightarrow_k r}$

$$\text{[E-Hole]}\ \frac{}{E \vdash ??_h \Rightarrow_k [E]\,??_h} \qquad \text{[E-Limit]}\ \frac{}{E \vdash e \Rightarrow_0 r}$$

$$\text{[E-Fix]}\ \frac{e = \mathtt{fix}\, f\,(\lambda x.e)}{E \vdash e \Rightarrow_k [E]\,e} \qquad \text{[E-Var]}\ \frac{x \mapsto r \in E}{E \vdash x \Rightarrow_k r} \qquad \text{[E-Unit]}\ \frac{}{E \vdash () \Rightarrow_k ()} \qquad \text{[E-Pair]}\ \frac{\{E \vdash e_i \Rightarrow_k r_i\}^{i \in [2]}}{E \vdash (e_1\,,\,e_2) \Rightarrow_k (r_1\,,\,r_2)} \qquad \text{[E-Ctor]}\ \frac{E \vdash e \Rightarrow_k r}{E \vdash C\,e \Rightarrow_k C\,r}$$

$$\text{[E-App]}\ \frac{\begin{array}{c} E \vdash e_1 \Rightarrow_k r_1 \qquad E \vdash e_2 \Rightarrow_k r_2 \\ r_1 = [E_f]\,\mathtt{fix}\, f\,(\lambda x.e_f) \qquad E_f,\, f \mapsto r_1,\, x \mapsto r_2 \vdash e_f \Rightarrow_{k-1} r \end{array}}{E \vdash e_1\,e_2 \Rightarrow_k r} \qquad \text{[E-App-Indet]}\ \frac{\begin{array}{c} E \vdash e_1 \Rightarrow_k r_1 \qquad E \vdash e_2 \Rightarrow_k r_2 \\ r_1 \neq [E_f]\,\mathtt{fix}\, f\,(\lambda x.e_f) \end{array}}{E \vdash e_1\,e_2 \Rightarrow_k r_1\,r_2}$$

$$\text{[E-Prj]}\ \frac{E \vdash e \Rightarrow_k (r_1\,,\,r_2)}{E \vdash \mathtt{prj}_{\,i \in [2]}\ e \Rightarrow_k r_i} \qquad \text{[E-Prj-Indet]}\ \frac{E \vdash e \Rightarrow_k r \qquad r \neq (r_1\,,\,r_2)}{E \vdash \mathtt{prj}_{\,i \in [2]}\ e \Rightarrow_k \mathtt{prj}_{\,i \in [2]}\ r}$$

$$\text{[E-Case]}\ \frac{\begin{array}{c} j \in [1,n] \qquad E \vdash e \Rightarrow_k C_j\,r \\ E,\, x_j \mapsto r \vdash e_j \Rightarrow_{k-1} r_j \end{array}}{E \vdash \mathtt{case}\ e\ \mathtt{of}\ \{C_i\ x_i \rightarrow e_i\}^{i \in [n]} \Rightarrow_k r_j} \qquad \text{[E-Case-Indet]}\ \frac{\begin{array}{c} E \vdash e \Rightarrow_k r \qquad \nexists j \in [1,n], r_j\ s.t.\ r = C_j\,r_j \\ r' = [E]\,\mathtt{case}\ r\ \mathtt{of}\ \{C_i\ x_i \rightarrow e_i\}^{i \in [n]} \end{array}}{E \vdash \mathtt{case}\ e\ \mathtt{of}\ \{C_i\ x_i \rightarrow e_i\}^{i \in [n]} \Rightarrow_k r'}$$

Figure 13. Augmented Evaluation with beta-depth limit. Only E-App and E-Case decrease the depth parameter.

Theorem A.1 (Determinism of Evaluation).
If $E \vdash e \Rightarrow r$ and $E \vdash e \Rightarrow r'$, then $r = r'$.

Theorem A.2 (Finality of Evaluation).
If $E \vdash e \Rightarrow r$, then r $\mathtt{final}$.

Type checking and evaluation are related by the following properties.

Theorem A.3 (Type Preservation).
If $\Sigma\,;\Delta\,;\Gamma \vdash e : T$ and $\Sigma\,;\Delta \vdash E : \Gamma$ and $E \vdash e \Rightarrow r$, then $\Sigma\,;\Delta \vdash r : T$.

Theorem A.4 (Progress).
For all k, if $\Sigma\,;\Delta\,;\Gamma \vdash e : T$ and $\Sigma\,;\Delta \vdash E : \Gamma$, there exists r s.t. $E \vdash e \Rightarrow_k r$ and $\Sigma\,;\Delta \vdash r : T$.

Proofs

Theorem A.1, Theorem A.2, and Theorem A.3.
Straightforward induction.

Theorem A.4 (Progress).
When $k = 0$, E-Limit will go through for any result. From the premise that e is well-typed ($\Sigma\,;\Delta\,;\Gamma \vdash e : T$), it's straightforward to derive a result of the same type.

When $k > 0$, the remaining cases go through by straightforward induction, thanks to the natural semantics.

A.4 Resumption

Figure 14 defines how to resume partially evaluated expressions. Resumption does not require an evaluation environment E, because results do not contain free variables.

Resumption $\boxed{F \vdash r \Rightarrow r'}$

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

$$\frac{F \vdash E \Rightarrow E'}{F \vdash [E]\,\texttt{fix}\, f\,(\lambda x.e) \Rightarrow [E']\,\texttt{fix}\, f\,(\lambda x.e)}\ \text{[R-Fix]} \qquad \frac{}{F \vdash () \Rightarrow ()}\ \text{[R-Unit]} \qquad \frac{F \vdash r_1 \Rightarrow r_1' \quad F \vdash r_2 \Rightarrow r_2'}{F \vdash (r_1, r_2) \Rightarrow (r_1', r_2')}\ \text{[R-Pair]} \qquad \frac{F \vdash r \Rightarrow r'}{F \vdash C\, r \Rightarrow C\, r'}\ \text{[R-Ctor]}$$

$$\frac{F \vdash r_1 \Rightarrow r_1' \quad F \vdash r_2 \Rightarrow r_2' \quad r_1' = [E_f]\,\texttt{fix}\, f\,(\lambda x.e_f) \quad E_f,\, f \mapsto r_1',\, x \mapsto r_2' \vdash e_f \Rightarrow r \quad F \vdash r \Rightarrow r'}{F \vdash r_1\, r_2 \Rightarrow r'}\ \text{[R-App]}$$

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

$$\frac{F \vdash r \Rightarrow (r_1, r_2)}{F \vdash \texttt{prj}_{\,i\in[2]}\, r \Rightarrow r_i}\ \text{[R-Prj]} \qquad \frac{F \vdash r \Rightarrow r' \quad r' \neq (r_1, r_2)}{F \vdash \texttt{prj}_{\,i\in[2]}\, r \Rightarrow \texttt{prj}_{\,i\in[2]}\, r'}\ \text{[R-Prj-Indet]}$$

$$\frac{\exists j \in [1,n] \quad F \vdash r \Rightarrow C_j\, r' \quad F \vdash ([E]\,\lambda x_j.e_j)\, r' \Rightarrow r_j}{F \vdash [E]\,\texttt{case}\; r\; \texttt{of}\; \{C_i\, x_i \rightarrow e_i\}^{\,i\in[n]} \Rightarrow r_j}\ \text{[R-Case]}$$

$$\frac{F \vdash r \Rightarrow r' \quad \nexists j \in [1,n], r_j\ s.t.\ r' = C_j\, r_j \quad F \vdash E \Rightarrow E' \quad r'' = [E']\,\texttt{case}\; r'\; \texttt{of}\; \{C_i\, x_i \rightarrow e_i\}^{\,i\in[n]}}{F \vdash [E]\,\texttt{case}\; r\; \texttt{of}\; \{C_i\, x_i \rightarrow e_i\}^{\,i\in[n]} \Rightarrow r''}\ \text{[R-Case-Indet]}$$

$$\frac{F \vdash r \Rightarrow C_j\, r_j}{F \vdash C_j^{\,-1}\, r \Rightarrow r_j'}\ \text{[R-Unwrap-Ctor]} \qquad \frac{F \vdash r \Rightarrow r' \quad r' \neq C_i\, r_i\ \text{(for any } i\text{)}}{F \vdash C^{-1}\, r \Rightarrow C^{-1}\, r'}\ \text{[R-Unwrap-Ctor-Indet]}$$

Environment Resumption $\boxed{F \vdash E \Rightarrow E'}$

$$\frac{}{F \vdash - \Rightarrow -} \qquad \frac{F \vdash E \Rightarrow E' \quad F \vdash r \Rightarrow r'}{F \vdash E,\, x \mapsto r \Rightarrow E',\, x \mapsto r'}$$

Figure 14. Resumption.

Theorem A.5 (Determinism of Resumption). *If $F \vdash r \Rightarrow r$ and $F \vdash r \Rightarrow r'$, then $r = r'$.*

Theorem A.6 (Finality of Resumption). *If $F \vdash r \Rightarrow r'$, then r'* `final`.

Theorem A.7 (Type Preservation of Resumption).
If $\Sigma \vdash F : \Delta$ and $\Sigma ; \Delta \vdash r : T$ and $F \vdash r \Rightarrow r'$, then $\Sigma ; \Delta \vdash r' : T$.

Lemma A.8 (Idempotency of Resumption).
If $F \vdash r_0 \Rightarrow r$, then $F \vdash r \Rightarrow r$.

Lemma A.9 (Simple Value Resumption).
If $\lceil r \rceil = v$, then $F \vdash r \Rightarrow r$.

Lemma A.10 (Resumption of App Operator).
If $F \vdash r_1 \Rightarrow r_1'$ and $F \vdash r_1'\, r_2 \Rightarrow r$, then $F \vdash r_1\, r_2 \Rightarrow r$.

Lemma A.11 (Resumption Composition).
If $F_1 \vdash r \Rightarrow r_1$ and $F_1 \oplus F_2 \vdash r_1 \Rightarrow r_2$, then $F_1 \oplus F_2 \vdash r \Rightarrow r_2$.

Lemma A.12 (Evaluation Respects Environment Resumption).
If $F_1 \vdash E \Rightarrow E'$ and $E \vdash e \Rightarrow r_1$ and $E' \vdash e \Rightarrow r_2$ and $F_1 \oplus F_2 \vdash r_1 \Rightarrow r_1'$ and $F_1 \oplus F_2 \vdash r_2 \Rightarrow r_2'$, then $r_1' = r_2'$.

Proofs

In the proofs in this section and the following sections, we assume that evaluation of a synthesized expression, and all its subterms, always terminate. This assumption is not valid in general. However, there are simple modifications we can make to ensure that this is true.

One approach is described in [29], whereby type contexts are annotated with tags that guarantee they obey structural recursion properties, ensuring that well-typed result environments do not contain non-terminating functions. Refine-Fix can synthesize recursive functions, but the structural recursion tagging in the context it uses to synthesize subterms would ensure that the function is structurally recursive and thus terminating. The only other way to synthesize a non-terminating term is by referring to one in the context via the Guess-Var rule, but an environment that type-checks against the context is protected from containing non-terminating terms via the structural recursion tagging. This approach is used in our implementation but is omitted from our theory for the sake of simplicity.

A simpler - though more limited - approach would be to restrict all fix terms - in the environment, and those produced by Refine-Fix - to be non-recursive.

Regardless the approach, the following proofs do rely on some such modification, since without it Refine-Fix may synthesize non-terminating functions which could then prevent evaluation or resumption from going through cleanly in the proof terms.

Similarly, there is a resumption case for all possible results, so the only way that resumption under a well-formed filling (if the filling for a given hole contains that hole, resumption of that hole can recurse infinitely) can fail to go through is if it relies on non-terminating evaluation. In the following proofs, all results that are resumed are evaluated to from synthesized expressions, or subterms of synthesized expressions. As such, we assume that both evaluation and resumption are total.

Theorem A.5, Theorem A.6, Theorem A.7, Theorem A.8, and Theorem A.9.
Straightforward induction.

Theorem A.10.

```
Given:
  (1)    F |= r1 => r1'
  (2)    F |= r1' r2 => r

Goal:    F |- r1  r2 => r

By inversion of resumption on (2), we get 2 cases:

Case 1: R-App
  (3)    F |- r1' => r1''
  (4)    F |- r2  => r2'
  (5)    F |- r1'' == [Ef] fix f (\x . ef)
  (6)    (Ef, f -> r1'', x -> r2') |- ef => r*
  (7)    F |- r* => r
By Idempotency of Resumption on (1)
  (8)    F |- r1' => r1'
By Determinism of Resumption on (3) and (8)
  (9)    r1' == r1''
Goal is given by R-App on (1) (observing (9)), (4), (5), (6), and (7)

Case 2: R-App-Indet
  (3)    F |- r1' => r1''
  (4)    F |- r2  => r2'
  (5)    r1'' =/= [E] fix f (\x . ef)
By Idempotency of Resumption on (1)
  (6)    F |- r1' => r1'
By Determinism of Resumption on (3) and (6)
  (7)    r1' == r1''
```

```
Goal is given by R-App-Indet on (1) (observing (7)), (4), and (5)
```

Theorem A.11 (Resumption composes).

Most cases are trivial or go through by straightforward induction, along with the evaluation and resumption assumptions. The non-trivial cases are considered in detail:

```
Case where first premise goes through R-Hole-Resume:
  Given:
    (1)    F |= [E]??h => r'
    (2)    F + F' |- r' => r''

  Goal:    F + F' |- [E]??h => r''

  Because this is the case where (1) goes through R-Hole-Resume,
  we can, by inversion, establish the premises of R-Hole-Resume
    (3)    F(h) = e
    (4)    E |- e => r
    (5)    F |- r => r'
  Since '+' is disjoint union, then by (3)
    (7)    (F + F')(h) = e
  By the induction hypothesis on (5) and (2)
    (8)    F + F' |- r => r''
  Goal is given by R-Hole-Resume on (7), (4), and (8)

Case where first premise goes through R-Hole-Indet,
    but second premise goes through R-Hole-Resume
  Given:
    (1)    F |= [E]??h => r'
    (2)    F + F' |- r' => r''

  Goal:    F + F' |- [E]??h => r''

  Because this is a case where (1) goes through R-Hole-Indet,
  we can, by inversion, establish the premises of R-Hole-Indet
    (3)    h not in F
    (4)    F |- E => E'
    (5)    r' == [E']??h
  Likewise, (2) goes through R-Hole-Resume (noting (5))
    (6)    (F + F')(h) = e
    (7)    E' |- e => r
    (8)    F + F' |- r => r''
  By the evaluation assumption
    (9)    E |- e => r*
  By the evaluation assumption
    (10)   F + F' |- r* => r*'
  By Evaluation Respects Environment Resumption on (4), (9), (7), (10), and (8)
    (11)   r'' => r*'
  Goal is given by R-Hole-Resume on (6), (9), and (10), observing (11)

Case where first premise goes through R-App:
  Given:
    (1)    F |= r1 r2 => r'
    (2)    F + F' |- r' => r''

  Goal:    F + F' |- r1 r2 => r''
```

```
  Because this is the case where the first premise goes through R-App,
  we can, by inversion, establish the premises of R-App
    (3)    F |- r1 => r1'
    (4)    F |- r2 => r2'
    (5)    r1' == [Ef'] fix f (\x . ef)
    (6)    (Ef', f -> r1', x -> r2') |- ef => r*
    (7)    F |- r* => r'
  By resumption assumption
    (8)    F + F' |- Ef' => Ef'+
    (9)    F + F' |- r2' => r2'+
  By R-Fix (observing (8))
    (10)   F + F' |- [Ef'] fix f (\x . ef) => [Ef'+] fix f (\x . ef)
  By the definition of environment resumption, (8), (9), and (10)
  (11)  F + F' |- (Ef', f -> [Ef'] fix f (\x . ef), x -> r2') => (Ef'+, f -> [Ef'+] fix f (\x . ef), x -> r2'+)
  By evaluation assumption
    (12)   (Ef'+, f -> [Ef'+] fix f (\x . ef), x -> r2'+) |- ef => r*+
  By the induction hypothesis on (7) and (2)
    (13)   F + F' |- r*  => r''
  By resumption assumption
    (14)   F + F' |- r*+ => r*++
  By Evaluation Respects Environment Resumption on (11), (6), (12), (13), and (14)
    (15)   r'' == r*++
  By the induction hypothesis on (3) and (10) (observing (5))
    (16)   F + F' |- r1 => [Ef'+] fix f (\x . ef)
  By the induction hypothesis on (4) and (9)
    (17)   F + F' |- r2 => r2'+
  By R-App on (16), (17), (trivial), (12), and (14)
    (18)   F + F' |- r1 r2 => r*++
  The goal is given by combining (15) and (18)

Case where first premise goes through R-Case
  Given:
    (1)    F |- [E] case r of {i < n | Ci xi -> ei} => r'
    (2)    F + F' |- r' => r''

  Goal:    F + F' |- [E] case r of {i < n | Ci xi -> ei} => r''

  Because this is the case where the first premise goes through R-Case,
  we can, by inversion, establish the premises of R-Case
    (3)    F |- r => Cj rj'
    (4)    F |- ([E] fix xj (\xj . ej)) rj' => r'
  By inversion of resumption on (4), we find that (4) can only go through
  R-App since the first argument is a syntactic fix (which by R-Fix will
  resume to a syntactic fix). So by inversion we can establish the premises
  of R-App (to establish premises (5) and (6), we use inversion again)
    (5)    F |- E => E'
    (6)    F |- [E] fix xj (\xj . ej) => [E'] fix xj (\xj . ej)
    (7)    F |- rj' => r2'
    (8)    (E', xj -> r2') |- ej => r*
    (9)    F |- r* => r'
  By R-Ctor on (7)
    (10)   F |- Cj rj' => Cj r2'
  By Idempotency of Resumption on (3)
```

```
      (11)   F |- Cj rj' => Cj rj'
    By Determinism of Resumption on (10) and (11)
      (12)   rj' == r2'
    By resumption assumption
      (13)   F + F' |- rj' => rj'+
      (14)   F + F' |- E'  => E'+
    By R-Ctor on (13)
      (15)   F + F' |- Cj rj' => Cj rj'+
    By the induction hypothesis on (3) and (15)
      (16)   F + F' |- r => Cj rj'+
    By R-Fix on (14)
      (17)   F + F' |- [E'] fix xj (\xj . ej) => [E'+] fix xj (\xj . ej)
    By the induction hypothesis on (6) and (17)
      (18)   F + F' |- [E] fix xj (\xj . ej) => [E'+] fix xj (\xj . ej)
    By Idempotency of Resumption on (13)
      (19)   F + F' |- rj'+ => rj'+
    By the evaluation assumption
      (20)   (E'+, xj -> rj'+) |- ej => r*+
    By the resumption assumption
      (21)   F + F' |- r* => r*++
      (22)   F + F' |- r*+ => r*'+
    By the definition of environment resumption, (14), and (13) (observing (12))
      (23)   F + F' |- (E', xj -> r2') => (E'+, xj -> rj'+)
    By Evaluation Respects Environment Resumption on (23), (8), (20), (21), and (22)
      (24)   r*++ == r*'+
    By R-App on (18), (19), (trivial), (20), and (22) (observing (24))
      (25)   F + F' |- ([E] fix xj (\xj . ej)) rj'+ => r*++
    By the induction hypothesis on (9) and (2)
      (26)   F + F' |- r* => r''
    By Determinism of Resumption on (21) and (26)
      (27)   r'' == r*++
    Goal is given by R-Case on (16) and (25) (observing (27))

  Case where first premise goes through R-Case-Indet and
  second premise goes through R-Case. We use inversion to establish
  the premises of these rules as givens.
    Given:
      (1)    F |- r => r'
      (2)    r' =/= Cj rj
      (3)    F |- E => E'
      (4)    F |- [E] case r of {i < n | Ci xi -> ei} => [E'] case r' of {i < n | Ci xi -> ei}
      (5)    F + F' |- r' => Cj rj'+
      (6)    F + F' |- ([E'] fix xj (\xj . ej)) rj'+ => r''

    Goal:    F + F' |- [E] case r of {i < n | Ci xi -> ei} => r''

    By resumption assumption
      (7)    F + F' |- E' => E'+
      (8)    F + F' |- ([E'+] fix xj (\xj . ej)) rj'+ => r''+
    By R-Fix on (7)
      (9)    F + F' |- [E'] fix xj (\xj . ej) => [E'+] fix xj (\xj . ej)
    By Resumption of App Operator on (9) and (8)
      (10)   F + F' |- ([E'] fix xj (\xj . ej)) rj'+ => r''+
    By Determinism of Resumption on (6) and (10)
```

```
    (11)   r'' == r''+
  By the induction hypothesis on (3) and (7)
    (12)   F + F' |- E => E'+
  By R-Fix on (12)
    (13)   F |- [E] fix xj (\xj . ej) => [E'+] fix xj (\xj . ej)
  By Resumption of App Operator on (13) and (8) (observing (11)
    (14)   F + F' |- ([E] fix xj (\xj . ej)) rj'+ => r''
  By the induction hypothesis on (1) and (5)
    (15)   F + F' |- r => Cj rj'+
  The goal is given by R-Case on (15) and (14)
```

Theorem A.12 (Evaluation respects environment resumption).
The E-Unit case is trivial.
The cases for E-Ctor and E-Pair go through by straightforward induction.
For E-Hole, if the hole is filled, the proof is also straightforward induction.
The unfilled case is very similar to the proof for the E-Fix case, detailed below.
The remaining cases are considered in detail:

```
E-Fix:
  Given:
    (1)    F |= E1 => E2
    (2)    E1 |- fix f (\x . e) => r1
    (3)    E2 |- fix f (\x . e) => r2
    (4)    F + F' |- r1 => r1'
    (5)    F + F' |- r2 => r2'

  Goal:    r1' == r2'

  By inversion of eval on (2)
    (6)    r1 == [E1] fix f (\x . e)
  By inversion of eval on (3)
    (7)    r2 == [E2] fix f (\x . e)
  By inversion of resumption on (6)
    (8)    F + F' |- E1 => E1'
    (9)    r1' == [E1'] fix f (\x . e)
  By inversion of resumption on (7)
    (10)   F + F' |- E2 => E2'
    (11)   r2' == [E2'] fix f (\x . e)
  By Resumption Composition (across all bindings in an environment) on (1) and (10)
    (12)   F + F' |- E1 => E2'
  By Determinism of Resumption (across all bindings in an environment) on (8) and (12)
    (13)   E1' == E2'
  Observing (13), (9) and (11) equate to form the goal

E-Var:
  Given:
    (1)    F |= E1 => E2
    (2)    E1 |- x => r1
    (3)    E2 |- x => r2
    (4)    F + F' |- r1 => r1'
    (5)    F + F' |- r2 => r2'

  Goal:    r1' == r2'

  By inversion of eval
```

```
      (6)    E1(x) == r1
      (7)    E2(x) == r2
    By (1), (6), and (7)
      (8)    F |- r1 => r2
    By Resumption Composition on (8) and (5)
      (9)    F + F' |- r1 => r2'
    By Determinism of Resumption on (4) and (9)
      (Goal) r1' == r2'

  For some expressions, evaluation can go through different rules,
  so the names of these cases will be given by the expression type
  rather than by the evaluation rule they go through.

  Applications
    Given:
      (1)    F |= E1 => E2
      (2)    E1 |- efun earg => r1
      (3)    E2 |- efun earg => r2
      (4)    F + F' |- r1 => r1'
      (5)    F + F' |- r2 => r2'

    Goal:    r1' == r2'

    By inversion of eval on (2) and (3), we get 4 cases:

    Case 1: E-App, E-App
      (6)    E1 |- efun => rfun1
      (7)    rfun1 == [Ef1] fix f1 (\x1 . ef1)
      (8)    E1 |- earg => rarg1
      (9)    (Ef1, f1 -> rfun1, x1 -> rarg1) |- ef1 => r1
      (10)   E2 |- efun => rfun2
      (11)   rfun2 == [Ef2] fix f2 (\x2 . ef2)
      (12)   E2 |- earg => rarg2
      (13)   (Ef2, f2 -> rfun2, x2 -> rarg2) |- ef2 => r2
    By R-Fix (observing (7) and (11))
      (14)   F + F' |- Ef1 => Ef1'
      (15)   F + F' |- rfun1 => [Ef1'] fix f1 (\x1 . ef1)
      (16)   F + F' |- Ef2 => Ef2'
      (17)   F + F' |- rfun2 => [Ef2'] fix f2 (\x2 . ef2)
    By the induction hypothesis on (1), (6), (10), (15), and (17)
      (18)   [Ef1'] fix f1 (\x1 . ef1) == [Ef2'] fix f2 (\x2 . ef2)
    By the resumption assumption
      (19)   F + F' |- rarg1 => rarg1'
      (20)   F + F' |- rarg2 => rarg2'
    By the induction hypothesis on (1), (8), (12), (19), and (20)
      (21)   rarg1' == rarg2'
    By the definition of environment resumption, (16), (17), and (20)
    (22)  F + F' |- (Ef2, f2 -> rfun2, x2 -> rarg2) => (Ef2', f2 -> [Ef2'] fix f2 (\x2 . ef2), x2 -> rarg2')
    By the evaluation assumption
      (23)   (Ef2', f2 -> [Ef2'] fix f2 (\x2 . ef2), x2 -> rarg2') |- ef2 => r2*
    By the resumption assumption
      (24)   F + F' |- r2* => r2*'
    By the induction hypothesis on (22), (13), (23), (5), and (24)
      (25)   r2*' == r2'
```

```
By the definition of environment resumption, (14), (18), (15), (19), and (21)
(26)  F + F' |- (Ef1, f1 -> rfun1, x1 -> rarg1) => (Ef2', f2 -> [Ef2'] fix f2 (\x2 . ef2), x2 -> rarg2')
By the induction hypothesis on (26), (9) (noting (18)), (23), (4), and (24)
  (27)   r2*' == r1'
Combining (25) and (27) gives the goal

Case 2: E-App, E-App-Indet
  (6)    E1 |- efun => rfun1
  (7)    rfun1 == [Ef1] fix f1 (\x1 . ef1)
  (8)    E1 |- earg => rarg1
  (9)    (Ef1, f1 -> rfun1, x1 -> rarg1) |- ef1 => r1
  (10)   r2 == r2fun r2arg
  (11)   E2 |- efun => r2fun
  (12)   r2fun =/= [Ef2] fix f2 (\x2 . ef2)
  (13)   E2 |- earg => r2arg
By R-Fix (observing (7))
  (14)   F + F' |- Ef1 => Ef1'
  (15)   F + F' |- rfun1 => [Ef1'] fix f1 (\x1 . ef1)
By the resumption assumption
  (16)   F + F' |- r2fun => r2fun'
  (17)   F + F' |- rarg1 => rarg1'
  (18)   F + F' |- r2arg => r2arg'
By the induction hypothesis on (1), (8), (13), (17), and (18)
  (19)   rarg1' == r2arg'
By the induction hypothesis on (1), (6), (11), (15), and (16)
  (20)   [Ef1'] fix f1 (\x1 . ef1) == r2fun'
By the evaluation assumption
  (21)   (Ef1', f -> [Ef1'] fix f1 (\x1 . ef1), x -> rarg1') |- ef1 => r*
By the resumption assumption
  (22)   F + F' |- r* => r*'
By R-App on (16), (18), (20), (21) (observing (19)), and (22)
  (23)   F + F' |- r2fun r2arg => r*'
By Determinism of Resumption on (5) and (23)
  (24)   r*' == r2'
By the definition of environment resumption, (14), (15), and (17)
(25)  F + F' |- (Ef1, f1 -> rfun1, x1 -> rarg1) => (Ef1', f1 -> [Ef1'] fix f1 (\x1 . ef1), x1 -> rarg1')
By the induction hypothesis on (25), (9), (21), (4), and (22)
  (26)   r*' == r1'
Combining (24) and (26) gives the goal

Case 3: E-App-Indet, E-App is analagous to the previous case

Case 4: E-App-Indet, E-App-Indet
  (6)    r1 == r1fun r1arg
  (7)    E1 |- efun => r1fun
  (8)    r1fun =/= [Ef1] fix f1 (\x1 . ef1)
  (9)    E1 |- earg => r1arg
  (10)   r2 == r2fun r2arg
  (11)   E2 |- efun => r2fun
  (12)   r2fun =/= [Ef2] fix f2 (\x2 . ef2)
  (13)   E2 |- earg => r2arg
By the resumption assumption
  (14)   F + F' |- r1fun => r1fun'
  (15)   F + F' |- r1arg => r1arg'
```

```
    (16)   F + F' |- r2fun => r2fun'
    (17)   F + F' |- r2arg => r2arg'
  By the induction hypothesis on (1), (7), (11), (14), and (16)
    (18)   r1fun' == r2fun'
  By the induction hypothesis on (1), (9), (13), (15), and (17)
    (19)   r1arg' == r2arg'
  Resumption of r1 and r2 could go through R-App or R-Indet, but in either case,
  the premises and conclusion are entirely determined by the resumptions of
  r1fun and r2fun, equated by (18), and r1arg and r2arg, equated by (19).
  As such, we can conclude that r1' == r2'

Prj (without loss of generality, we will only detail the prj_1 cases)
  Given:
    (1)    F |= E1 => E2
    (2)    E1 |- prj_1 e => r1
    (3)    E2 |- prj_1 e => r2
    (4)    F + F' |- r1 => r1'
    (5)    F + F' |- r2 => r2'

  Goal:    r1' == r2'

  By inversion of eval on (2) and (3), we get 4 cases:

  Case 1: E-Prj, E-Prj
    (6)    E1 |- e => (r1, r1*)
    (7)    E2 |- e => (r2, r2*)
  By the resumption assumption
    (8)    F + F' |- r1* => r1*'
    (9)    F + F' |- r2* => r2*'
  By R-Pair on (4) and (8)
    (10)   F + F' |- (r1, r1*) => (r1', r1*')
  By R-Pair on (5) and (9)
    (11)   F + F' |- (r2, r2*) => (r2', r2*')
  By the induction hypothesis on (1), (6), (7), (10), and (11)
    (12)   (r1', r1*') == (r2', r2*')
  The goal follows from (12)

  Case 2: E-Prj, E-Prj-Indet
    (6)    E1 |- e => (r1, r1*)
    (7)    r2 == prj_1 r2*
    (8)    E2 |- e => r2*
    (9)    r2* =/= (r-a, r-b)
  By the resumption assumption
    (10)   F + F' |- r1* => r1*'
    (11)   F + F' |- r2* => r2*'
  By R-Pair on (4) and (10)
    (12)   F + F' |- (r1, r1*) => (r1', r1*')
  By the induction hypothesis on (1), (6), (8), (12), and (11)
    (13)   (r1', r1*') == r2*'
  By R-Prj on (11) (observing (13) and (7))
    (14)   F + F' |- r2 => r1'
  Determinism of Resumption combined with (5) and (14) yield the goal

  Case 3: E-Prj-Indet, E-Prj is analagous to the previous case
```

```
    Case 4: E-Prj-Indet, E-Prj-Indet
      (6)    r1 == prj_1 r1*
      (7)    E1 |- e => r1*
      (8)    r1* =/= (r-a, r-b)
      (9)    r2 == prj_1 r2*
      (10)   E2 |- e => r2*
      (11)   r2* =/= (r-a, r-b)
    By the resumption assumption
      (12)   F + F' |- r1* => r1*'
      (13)   F + F' |- r2* => r2*'
    By the induction hypothesis on (1), (7), (10), (12), and (13)
      (14)   r1*' == r2*'
    Resumption of r1 and r2 could go through R-Prj or R-Prj-Indet, but in either case,
    the premises and conclusion are entirely determined by the resumptions of
    r1* and r2*, equated by (14). As such, we can conclude that r1' == r2'

  Case/Match
    Given:
      (1)    F |= E1 => E2
      (2)    E1 |- case e of {i < n | Ci xi -> ei} => r1
      (3)    E2 |- case e of {i < n | Ci xi -> ei} => r2
      (4)    F + F' |- r1 => r1'
      (5)    F + F' |- r2 => r2'

    Goal:    r1' == r2'

    By inversion of eval on (2) and (3), we get 4 cases:

    Case 1: E-Case, E-Case
      (6)    E1 |- e => Cj1 r1*  (for some j1 < n)
      (7)    (E1, xj1 -> r1*) |- ej1 => r1
      (8)    E2 |- e => Cj2 r2*  (for some j2 < n)
      (9)    (E2, xj2 -> r2*) |- ej2 => r2
    By the resumption assumption
      (10)   F + F' |- r1* => r1*'
      (11)   F + F' |- r2* => r2*'
    By R-Ctor on (10)
      (12)   F + F' |- Cj1 r1* => Cj1 r1*'
    By R-Ctor on (11)
      (13)   F + F' |- Cj2 r2* => Cj2 r2*'
    By the induction hypothesis on (1), (6), (8), (12), and (13)
      (14)   Cj1 r1*' == Cj2 r2*'
             By (14), Cj1 == Cj2 - since each constructor of a given type is unique,
               this implies that j1 == j2, and thus that xj1 == xj2 and ej1 == ej2
    By the resumption assumption
      (15)   F + F' |- E2 => E2'
    By Resumption Composition (across all bindings in an environment) on (1) and (15)
      (16)   F + F' |- E1 => E2'
    By the evaluation assumption
      (17)   (E2', xj2 -> r2*') |- ej2 => r*
    By the resumption assumption
      (18)   F + F' |- r* => r*'
    By the definition of environment resumption, (16), (10), and (14)
```

```
  (19)   F + F' |- (E1, xj1 -> r1*) => (E2', xj2 -> r2*')
By the induction hypothesis on (19), (7), (17), (4), and (18)
  (20)   r*' == r1'
By the definition of environment resumption, (15), and (11)
  (21)   F + F' |- (E2, xj2 -> r2*) => (E2', xj2 -> r2*')
By the induction hypothesis on (21), (9), (17), (5), and (18)
  (22)   r*' == r2'
Combining (20) and (22) gives the goal

Case 2: E-Case, E-Case-Indet
  (6)    E1 |- e => Cj1 r1*  (for some j1 < n)
  (7)    (E1, xj1 -> r1*) |- ej1 => r1
  (8)    E2 |- e => r2*
  (9)    r2* =/= Cj rj
  (10)   r2 == [E2] case r2* of {i < n | Ci xi -> ei}
By the resumption assumption
  (11)   F + F' |- r1* => r1*'
  (12)   F + F' |- r2* => r2*'
  (13)   F + F' |- E2 => E2'
By R-Ctor on (11)
  (14)   F + F' |- Cj1 r1* => Cj1 r1*'
By the induction hypothesis on (1), (6), (8), (14), and (12)
  (15)   Cj1 r1*' == r2*'
By R-Fix on (13)
  (16)   F + F' |- [E2] fix xj1 (\xj1 . ej1) => [E2'] fix xj1 (\xj1 . ej1)
By Idempotency of Resumption on (11)
  (17)   F + F' |- r1*' => r1*'
By the evaluation assumption
  (18)   (E2', xj1 -> r1*') |- ej1 => r*
By the resumption assumption
  (19)   F + F' |- r* => r*'
By R-App on (16), (17), (trivial), (18), and (19)
  (20)   F + F' |- ([E2] fix xj1 (\xj1 . ej1)) r1*' => r*'
By R-Case on (12) (noting (15)) and (20)
  (21)   F + F' |- r2 => r*'
By Determinism of Resumption on (5) and (21)
  (22)   r*' == r2'
By Resumption Composition (across all bindings in an environment) on (1) and (13)
  (23)   F + F' |- E1 => E2'
By the definition of environment resumption, (23), and (11)
  (24)   F + F' |- (E1, xj1 -> r1*) => (E2', xj1 -> r1*')
By the induction hypothesis on (24), (7), (18), (4), and (19)
  (25)   r*' == r1'
Combining (22) and (25) gives the goal

Case 3: E-Case-Indet, E-Case is analagous to the previous case

Case 4: E-Case-Indet, E-Case-Indet
  (6)    E1 |- e => r1*
  (7)    r1* =/= Cj rj
  (8)    r1 == [E1] case r1* of {i < n | Ci xi -> ei}
  (9)    E2 |- e => r2*
  (10)   r2* =/= Cj rj
  (11)   r2 == [E2] case r2* of {i < n | Ci xi -> ei}
```

```
By the resumption assumption
  (12)   F + F' |- r1* => r1*'
  (13)   F + F' |- r2* => r2*'
By the induction hypothesis on (1), (6), (8), (12), and (13)
  (14)   r1*' == r2*'
By the resumption assumption
  (15)   F + F' |- E2 => E2'
By Resumption Composition (across all bindings in an environment) on (1) and (15)
  (16)   F + F' |- E1 => E2'
Resumption of r1 and r2 could go through either R-Case or R-Case-Indet.
For R-Case, all aspects of the premises and conclusion depend only on the
resumptions of r1* and r2* - which are equated by (14) - except for the
'E' in the premise 'F |- ([E] \xj . ej) r' => rj'. Noting R-Fix, this premise
must go through R-App, whose premises and conclusions are entirely dependent
on the resumptions of the operands 'r_1' and 'r_2'. By R-Fix, (15), and (16),
the resumptions of [E1] fix xj (\xj . ej) and [E2] fix xj (\xj . ej)
are equal.
The situation for R-Case-Indet is similar, though simpler, since its premises
and conclusion depend only on the resumptions of r1* and r2* and the resumptions
of E1 and E2, which are equated by (15) and (16).
Since the resumptions of r1 and r2 (i.e. r1' and r2') depend entirely on values
which are equated, they must themselves be equated.
```

A.5 Unevaluation Constraint Merging

Figure 15 defines the merge operations for constraints.

(Syntactic) Constraint Merging $\boxed{F_1 \oplus F_2 = F}$ $\boxed{U_1 \oplus U_2 = U}$ $\boxed{K_1 \oplus K_2 = K}$

$$\frac{\forall ??_h \in dom(F_1) \cap dom(F_2).\ F_1(??_h) = F_2(??_h)}{F_1 \oplus F_2 = F_1 +\!\!+ F_2}$$

$$\frac{\begin{array}{c} U_1' = U_1 \setminus dom(U_2) \qquad U_2' = U_2 \setminus dom(U_1) \\ U_{12} = \{\, h \mapsto U_1(??_h) +\!\!+ U_2(??_h) \mid ??_h \in dom(F_1) \cap dom(F_2) \,\} \end{array}}{U_1 \oplus U_2 = U_1' +\!\!+ U_{12} +\!\!+ U_2'}$$

$$\frac{F_1 \oplus F_2 = F' \qquad U_1 \oplus U_2 = U'}{(U_1 \,;\, F_1) \oplus (U_2 \,;\, F_2) = (U' \,;\, F')}$$

(Semantic) Constraint Merging $\boxed{\Sigma \,;\, \Delta \,;\, Merge(K) \rhd K'}$

$$\frac{F(??_h) = e \qquad \Sigma \,;\, \Delta \,;\, F \vdash e \rightleftharpoons X \dashv K}{\Sigma \,;\, \Delta \,;\, Resolve(h \mapsto X \,;\, F) \rightsquigarrow K} \qquad \frac{??_h \notin F}{\Sigma \,;\, \Delta \,;\, Resolve(h \mapsto X \,;\, F) \rightsquigarrow (h \mapsto X \,;\, -)}$$

$$\frac{\{\, \Sigma \,;\, \Delta \,;\, Resolve(h_i \mapsto X \,;\, F) \rightsquigarrow K_i' \,\}^{\,i \in [n]}}{\Sigma \,;\, \Delta \,;\, Step(h_1 \mapsto X_1, \ldots, h_n \mapsto X_n \,;\, F) \rightsquigarrow (- \,;\, F) \oplus K_1' \oplus \cdots \oplus K_n'}$$

$$\frac{\begin{array}{c} \Sigma \,;\, \Delta \,;\, Step(K) \rightsquigarrow K' \qquad K \neq K' \\ \Sigma \,;\, \Delta \,;\, Merge(K') \rhd K'' \end{array}}{\Sigma \,;\, \Delta \,;\, Merge(K) \rhd K''} \qquad \frac{\Sigma \,;\, \Delta \,;\, Step(K) \rightsquigarrow K' \qquad K = K'}{\Sigma \,;\, \Delta \,;\, Merge(K) \rhd K'}$$

Figure 15. Constraint Merging.

A.6 Type-Directed Guessing

Figure 16 defines type-directed guessing rules analogous to expression type rules (Figure 11).

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

$$\text{[Guess-Unit]}\ \frac{}{(\Gamma \vdash \bullet : ()) \rightsquigarrow_{\text{guess}} ()} \qquad \text{[Guess-Pair]}\ \frac{\{\,(\Gamma \vdash \bullet : T_i) \rightsquigarrow_{\text{guess}} e_i\,\}^{\,i\in[2]}}{(\Gamma \vdash \bullet : (T_1\,,\,T_2)) \rightsquigarrow_{\text{guess}} (e_1\,,\,e_2)}$$

$$\text{[Guess-Ctor]}\ \frac{\Sigma(D)(C) = T \qquad (\Gamma \vdash \bullet : T) \rightsquigarrow_{\text{guess}} e}{(\Gamma \vdash \bullet : D) \rightsquigarrow_{\text{guess}} C\ e} \qquad \text{[Guess-Fix]}\ \frac{(\Gamma,\ f : T_1 \to T_2,\ x : T_1 \vdash \bullet : T_2) \rightsquigarrow_{\text{guess}} e}{(\Gamma \vdash \bullet : T_1 \to T_2) \rightsquigarrow_{\text{guess}} \mathtt{fix}\, f\,(\lambda x.e)}$$

$$\text{[Guess-Case]}\ \frac{\Sigma(D) = \{C_i\ T_i\}^{\,i\in[n]} \qquad (\Gamma \vdash \bullet : D) \rightsquigarrow_{\text{guess}} e \qquad \{\,(\Gamma,\ x_i : T_i \vdash \bullet : T) \rightsquigarrow_{\text{guess}} e_i\,\}^{\,i\in[n]}}{(\Gamma \vdash \bullet : T) \rightsquigarrow_{\text{guess}} \mathtt{case}\ e\ \mathtt{of}\ \{C_i\ x_i \to e_i\}^{\,i\in[n]}}$$

$$\text{[Guess-Var]}\ \frac{\Gamma(x) = T}{(\Gamma \vdash \bullet : T) \rightsquigarrow_{\text{guess}} x} \qquad \text{[Guess-App]}\ \frac{\begin{array}{c}(\Gamma \vdash \bullet : T_2 \to T) \rightsquigarrow_{\text{guess}} e_1 \\ (\Gamma \vdash \bullet : T_2) \rightsquigarrow_{\text{guess}} e_2\end{array}}{(\Gamma \vdash \bullet : T) \rightsquigarrow_{\text{guess}} e_1\ e_2} \qquad \text{[Guess-Prj]}\ \frac{(\Gamma \vdash \bullet : (T_1\,,\,T_2)) \rightsquigarrow_{\text{guess}} e}{(\Gamma \vdash \bullet : T_i) \rightsquigarrow_{\text{guess}} \mathtt{prj}_{\,i\in[2]}\ e}$$

Figure 16. Type-Directed Guessing.

Guessing Recursive Sketches. Guessing does not generate hole expressions. Guessing is, furthermore, limited to small terms and elimination forms in practice. However, if guessing were to generate recursive function sketches, the Guess-and-Check rule provides an additional antidote for trace-completeness: when guessing an expression $\mathtt{fix}\, f\,(\lambda x.e)$ to fill $??_h$, the extended hole-filling $F,\ h \mapsto \mathtt{fix}\, f\,(\lambda x.e)$ "ties the recursive knot" before checking example consistency.

A.7 Synthesis Soundness

Theorem A.13 (Type Soundness of Unevaluation).
If $\Sigma \vdash F : \Delta$ *and* $\Sigma ; \Delta \vdash r : T$ *and* $\Sigma ; \Delta \vdash ex : T$ *and* $F \vdash r \Leftarrow ex \dashv (U, F)$, *then* $\Sigma \vdash U : \Delta$ *and* $\Sigma \vdash F : \Delta$.

Theorem A.14 (Type Soundness of Checking).
If $\Sigma \vdash F' : \Delta$ *and* $\Sigma ; \Delta \vdash X : \Gamma ; T$ *and* $\Sigma ; \Delta ; \Gamma \vdash e : T$ *and* $F' \vdash e \rightleftharpoons X \dashv (U, F)$, *then* $\Sigma \vdash U : \Delta$ *and* $\Sigma \vdash F : \Delta$.

Theorem A.15 (Type Soundness of Guess).
If $\Sigma ; (\Gamma \vdash \bullet : T) \rightsquigarrow_{guess} e$, *then* $\Sigma ; \Delta ; \Gamma \vdash e : T$.

Theorem A.16 (Type Soundness of Refine/Branch).
If $\Sigma ; \Delta \vdash X : \Gamma ; T$ *and* $(\Gamma \vdash \bullet : T \models X) \rightsquigarrow_{\{refine, branch\}} e \dashv \{ (\Gamma_i \vdash \bullet_{h_i} : T_i \models X_i) \}^{i \in [n]}$,
then $\Sigma ; \Delta +\!\!+ \{ h_i \mapsto (\Gamma_i \vdash \bullet : T_i) \}^{i \in [n]} ; \Gamma \vdash e : T$ *and* $\Sigma ; \Delta \vdash X_i : \Gamma_i ; T_i$.

Theorem A.17 (Type Soundness of Fill).
If $\Sigma \vdash F' : \Delta$ *and* $\Sigma ; \Delta \vdash X : \Gamma ; T$ *and* $\Sigma ; \Delta ; F' ; (\Gamma \vdash \bullet_h : T \models X) \rightsquigarrow_{fill} (U ; F) ; \Delta'$
then $\Sigma \vdash (U, F) : \Delta +\!\!+ \Delta' +\!\!+ (h \mapsto (\Gamma \vdash \bullet : T))$.

Theorem A.18 (Type Soundness of Result Consistency).
If $\Sigma ; \Delta \vdash r : T$ *and* $\Sigma ; \Delta \vdash r' : T$ *and* $r \equiv_A r'$ *then* $\Sigma \vdash A : \Delta$.

Theorem A.19 (Type Soundness of Simplify).
If $\Sigma \vdash A : \Delta$ *and* $Simplify(A) \rhd (U, F)$, *then* $\Sigma \vdash U : \Delta$ *and* $\Sigma \vdash F : \Delta$.

Theorem A.20 (Type Soundness of Program Evaluation).
If $\Sigma ; \Delta \vdash p : T ; T'$ *and* $p \Rightarrow r ; A$, *then* $\Sigma ; \Delta \vdash r : T$ *and* $\Sigma \vdash A : \Delta$.

Theorem A.21 (Soundness of Example Unevaluation).
If $F \oplus F' \models K$ *and* r `final` *and* $F \vdash r \Leftarrow ex \dashv K$ *and* $F \oplus F' \vdash r \Rightarrow r'$ *then* $F \oplus F' \vdash r' \models ex$.

Theorem A.22 (Soundness of Live Bidirectional Example Checking).
If $F \oplus F' \models K$ *and* $F \vdash e \rightleftharpoons X \dashv K$, *then* $F \oplus F' \vdash e \models X$.

Theorem A.23 (Example Soundness of Refine).
If $\Sigma ; \Delta ; (\Gamma \vdash \bullet : T \models X) \rightsquigarrow_{refine} e \dashv \{ (\Gamma_i \vdash \bullet_{h_i} : T_i \models X_i) \}^{i \in [n]}$ *and* $\{ F \vdash ??_{h_i} \models X_i \}^{i \in [n]}$, *then* $F \vdash e \models X$.

Theorem A.24 (Example Soundness of Branch).
If $\Sigma ; \Delta ; (\Gamma \vdash \bullet : T \models X) \rightsquigarrow_{branch} e \dashv \{ (\Gamma_i \vdash \bullet_{h_i} : T_i \models X_i) \}^{i \in [n]}$ *and* $\{ F \vdash ??_{h_i} \models X_i \}^{i \in [n]}$, *then* $F \vdash e \models X$.

Theorem A.25 (Example Soundness of Fill).
If $\Sigma ; \Delta ; F ; (\Gamma \vdash \bullet_h : T \models X) \rightsquigarrow_{fill} K ; \Delta'$ *and* $F \oplus F' \models K$, *then* $(F \oplus F')(??_h) = e$ *and* $F \oplus F' \vdash e \models X$.

Theorem A.26 (Type Soundness of Semantic Merge).
If $\Sigma \vdash K : \Delta$ *and* $\Sigma ; \Delta ; Merge(K) \rhd K'$, *then* $\Sigma \vdash K' : \Delta$.

Theorem A.27 (Example Soundness of Semantic Merge).
If $F \models K'$ *and* $\Sigma ; \Delta ; Merge(K) \rhd K'$, *then* $F \models K$.

Theorem A.28 (Soundness of Solve).
If $\Sigma \vdash U : \Delta$ *and* $\Sigma \vdash F : \Delta$ *and* $\Sigma ; \Delta ; Solve(U, F) \rightsquigarrow F' ; \Delta'$, *then* $\Sigma \vdash F' : \Delta'$ *and* $F' \models (U, F)$.

Theorem A.29 (Soundness of Assertion Simplification).
If $Simplify(A) \rhd K$ *and* $F \models K$, *then* $F \models A$.

Theorem A.30 (Soundness of Synthesis).
If $\Sigma ; \Delta \vdash p : T ; T'$ *and* $p \Rightarrow r ; A$ *and* $Simplify(A) \rhd K$ *and* $\Sigma ; \Delta ; Solve(K) \rightsquigarrow F ; \Delta'$, *then* $\Sigma \vdash F : \Delta'$ *and* $F \models A$.

Lemma A.31 (Example Satisfaction of Simple Value).
If $\lceil ex \rceil = v$ *and* $F \vdash r \models ex$, *then* $\lceil r \rceil = v$.

Lemma A.32 (Constraint Satisfaction Implies Complete Resumption).
If $\lceil ex \rceil = v$ *and* r `final` *and* $- \vdash r \Leftarrow ex \dashv K$ *and* $F \models K$, *then* $F \vdash r \Rightarrow r'$ *and* $\lceil r' \rceil = v$.

Proofs

Theorem A.13 (Uneval) and Theorem A.14 (Check).
Straightforward mutual induction.

Theorem A.15 (Guess).
Straightforward induction.

Theorem A.16 (Refine/Branch).
Straightforward by way of Theorem A.15.

Theorem A.17 (Fill).
The Defer case is trivial.
The Refine,Branch case is straightforward by way of Theorem A.16.
The Guess-and-Check case goes through by Theorem A.14.

Theorem A.18 (Result consistency).
Straightforward induction.

Theorem A.19 (Simplify).
Straightforward by way of Theorem A.13.

Theorem A.20 (Program evaluation).
Straightforward by way of Theorem A.3 and Theorem A.18.

Theorem A.21 (Uneval implies example satisfaction).
The cases U-Top, U-Unit, U-Pair and U-Ctor are straightforward applications of their respective XS rules and induction.
U-Hole goes through because the premise $F \oplus F' \models K$ proves example satisfaction for the single generated constraint.
The remaining cases are considered in detail:

```
U-Fix:
  Given:
    (1)    F + F' |= K
    (2)    [E] fix f (\x. e) final
    (3)    F |- [E] fix f (\x. e) <= {v -> ex} -| K
    (4)    F + F' |- [E] fix f (\x. e) => r'

  Goal:    F + F' ; r' |= {v -> ex}

  By inversion of unevaluation on (3)
    (5)    F -| e <=> ((E, f -> [E] fix f (\x. e), x -> v), ex) -| K
  By inversion of the check judgment on (5)
    (6)    (E, f -> [E] fix f (\x. e), x -> v) |- e => r*
    (7)    F |- r* => r**
    (8)    F |- r** <= ex -| K
  By resumption assumption
    (9)    F + F' |- r** => r***
  By Resumption Composition on (7) and (9)
    (10)   F + F' |- r* => r***
  By the induction hypothesis on (1), Finality of Resumption, (8), and (9)
    (11)   F + F' ; r*** |= ex
  By inversion of resumption on (4)
    (12)   F + F' |- E => E'
    (13)   r' == [E'] fix f (\x . e)
  By evaluation assumption
    (14)   (E', f -> [E'] fix f (\x. e), x -> v) |- e => r*'
  By Simple Value Resumption
    (15)   F + F' |- v => v
  By resumption assumption
```

```
      (16)   F + F' |- r*' => r*''
    By Idempotency of Resumption on (4)
      (17)   F + F" |- [E'] fix f (\x . e) => [E'] fix f (\x . e)
    By R-App on (17), (15), (13), (14), and (16)
      (18)   F + F' |- ([E'] fix f (\x . e)) v => r*''
    By the definition of environment resumption, (12), (4), and (15)
      (19)   F + F' |- (E, f -> [E] fix f (\x. e), x -> v) => (E', f -> [E'] fix f (\x. e), x -> v)
    By Evaluation Respects Environment Resumption on (19), (6), (14), (10), and (16)
      (20)   r**+ == r*''
    By XS-Input-Output on (18), (20), and (11)
      (21)   F + F' ; [E'] fix f (\x . e) |= {v -> ex}
    Observing (13), (21) is the goal

  U-App:
    Given:
      (1)    F + F' |= K
      (2)    (r1 r2) final
      (3)    F |- r1 r2 <= ex -| K
      (4)    F + F' |- r1 r2 => r'

    Goal:    F + F' ; r' |= ex

    By inversion of unevaluation on (3)
      (5)    r2 value
      (6)    F |- r1 <= {r2 -> ex} -| K
    By the resumption assumption
      (7)    F + F' |- r1 => r1'
    By the induction hypothesis on (1), inversion of final on (2), (6) and (7)
      (8)    F + F' ; r1' |= {r2 -> ex}
    By inversion of example satisfaction on (8)
      (9)    F + F' |- r1' r2 => r
      (10)   F + F' ; r |= ex
    By Resumption of App Operator on (7) and (9)
      (11)   F + F' |- r1 r2 => r
    By Determinism of Resumption on (4) and (11)
      (12)   r == r'
    Goal is given by (10), observing (12)

  U-Prj-1 (the proof for U-Prj-2 is essentially equivalent):
    Given:
      (1)    F + F' |= K
      (2)    (prj_1 r) final
      (3)    F |- prj_1 r <= ex -| K
      (4)    F + F' |- prj_1 r => r'

    Goal:    F + F' ; r' |= ex

    By inversion of unevaluation on (3)
      (5)    F |- r <= (ex, T) -| K
    By the resumption assumption
      (6)    F + F' |- r => r+
    By the induction hypothesis on (1), inversion of final on (2), (5), and (6)
      (7)    F + F' ; r+ |= (ex, T)
    By inversion of example satisfaction on (7)
```

```
      (8)    r+ == (r+1, r+2)
      (9)    F + F' ; r+1 |= ex
    By R-Prj on (6) (observing (8))
      (10)   F + F' |- prj_1 r => r+1
    By Determinism of Resumption on (4) and (10)
      (11)   r' == r+1
    Goal is given by (9), observing (11)

  U-Case:
    Given:
      (1)    F + F' |= K1 + K2
      (2)    ([E] case r of {Ci xi -> ei | i <= n}) final
      (3)    F |- [E] case r of {Ci xi -> ei | i <= n} <= ex -| K1 ++ K2
      (4)    F + F' |- [E] case r of {Ci xi -> ei | i <= n} => r'

    Goal:    F + F' ; r' |= ex

    By inversion of unevaluation on (3), going through U-Case
      (5)    F |- r <= Cj T -| K1
      (6)    F |- ej <=> ((E, xj -> Cj-1 r), ex) -| K2
    By inversion of checking on (6)
      (7)    (E, xj -> Cj-1 r) |- ej => r0
      (8)    F |- r0 => r0'
      (9)    F |- r0' <= ex -| K2
    By Finality of Resumption on (8)
      (10)   r0' final
    By the resumption assumption
      (11)   F + F' |- r0' => r0'+
      (12)   F + F' |- r => r+
    By the induction hypothesis on (1), (10), (9), and (11)
      (13)   F + F' ; r0'+ |= ex
    By the induction hypothesis on (1), inversion of final on (2), (5), and (12)
      (14)   F + F' ; r+ |= Cj T
    By inversion of example satisfaction on (14)
      (15)   r+ == Cj r+'
    By inversion of resumption on (4), noting that on account of (15),
        (12) is the first premise of R-Case and precludes R-Case-Indet
      (16)   F + F' |- ([E] fix xj (\xj . ej)) r+' => r'
    By inversion of resumption on (16)
      (17)   F + F' |- [E] fix xj (\xj . ej) => [E'] fix xj (\xj . ej)
      (18)   F + F' |- r+' => r+'
      (19)   (E', xj -> r+') |- ej => r*
      (20)   F + F' |- r* => r'
    By Resumption Composition on (8) and (11)
      (21)   F + F' |- r0 => r0'+
    By R-Unwrap-Ctor on (12), observing (15)
      (22)   F + F' |- Cj-1 r => r+'
    By Evaluation Respects Environment Resumption on (17/22), (7), (19), (21), and (20)
      (23)   r' == r0'+
    Goal is given by (13), noting (23)

  U-Inverse-Ctor:
    Given:
      (1)    F + F' |= K
```

```
      (2)    C-1 r final
      (3)    F |- C-1 r <= ex -| K
      (4)    F + F' |- C-1 r => r'

    Goal:    F + F' ; r' |= ex

    By inversion of unevaluation on (3)
      (5)    F |- r <= C ex -| K
    By the resumption assumption
      (6)    F + F' |- r => r+
    By the induction hypothesis on (1), inversion of final on (2), (5), and (6)
      (7)    F + F' ; r+ |= C ex
    By inversion of example satisfaction on (7)
      (8)    r+ == C r+'
      (9)    F + F' ; r+' |= ex
    By R-Unwrap-Ctor on (6), observing (8)
      (10)   F + F' |- C-1 r => r+'
    By Determinism of Resumption on (4) and (10)
      (11)   r' == r+'
    Goal is given by (9), observing (11)

  U-Case-Guess:
    Given:
      (1)    F + F' |= (-, Fg) ++ K
      (2)    ([E] case r of {Ci xi -> ei | i <= n}) final
      (3)    F |- [E] case r of {Ci xi -> ei | i <= n} <= ex -| (-, Fg) ++ K
      (4)    F + F' |- [E] case r of {Ci xi -> ei | i <= n} => r'

    Goal:    F + F' ; r' |= ex

    By inversion of unevaluation on (3)
      (5)    Fg = Guesses(Dlt, Sig, r)
      (6)    F + Fg |- r => Cj rj
      (7)    F + Fg |- ej <=> ((E, xj -> rj), ex)) -| K
    By inversion of the check judgment on (7)
      (8)    (E, xj -> rj) |- ej => r0
      (9)    F + Fg |- r0 => r0'
      (10)   F + Fg |- r0' <= ex -| K
    By (1), Fg and K must be disjoint, since they're disjoint merged. Likewise,
    by (6) and others, F and Fg are disjoint. By the definition of `|=`, F + F'
    must be a supermapping of the first component of `(Fg, -) + K`, which,
    noting the previous observations, means F' is a supermapping of Fg
      (11)   F + F' == F + Fg + (F' - Fg)
      (12)   F + F' |= K
    By the resumption assumption
      (13)   F + F' |- r0' => r0+
    By the induction hypothesis on (12) (observing (11)), Finality of Resumption,
                                   (10), and (13)
      (14)   F + F' ; r0+ |= ex
    By Resumption Composition on (9) and (13) (observing (11))
      (15)   F + F' |- r0 => r0+
    By the resumption assumption
      (16)   F + F' |- rj => rj+
    By R-Ctor on (16)
```

```
      (17)   F + F' |- Cj rj => Cj rj+
    By Resumption Composition on (6) and (17) (observing (11))
      (18)   F + F' |- r => Cj rj+
    By inversion of resumption on (4), noting that (18) is the first premise of
    R-Case and precludes R-Case-Indet
      (19)   F + F' |- ([E] fix xj (\xj . ej)) rj+ => r'
    By the resumption assumption
      (20)   F + F' |- E => E+
    By R-Fix on (20)
      (22)   F + F' |- [E] fix xj (\xj . ej) => [E+] fix xj (\xj . ej)
    By inversion of resumption on (19), noting that (22) is the first premise of
    R-App and precludes R-App-Indet
      (23)   F + F' |- rj+ => rj+   (noting Idempotency of Resumption on (16))
      (24)   (E+, xj -> rj+) |- ej => r*
      (25)   F + F' |- r* => r'
    By the definition of environment resumption, (20), and (16)
      (26)   F + F' |- (E, xj -> rj) => (E+, xj -> rj+)
    By Evaluation Respects Environment Resumption on (26), (8), (24), (15), and (25)
      (27)   r' == r0+
    Goal is given by (14), observing (27)
```

Theorem A.22 (Check).

```
Given:
  (1)    F + F' |= K
  (2)    F |- e <=> {Ei, exi | i <= n} -| K

Goal:    F + F' |- e |= {Ei, exi | i <= n}

By inversion of checking on (2)
  (3)    Ei |- e => ri
  (4)    F |- ri => r'i
  (5)    F |- r'i <= exi -| Ki
  (6)    K == K1 ++ ... ++ Kn
By Finality of Resumption on (4)
  (7)    r'i final
By the resumption assumption
  (8)    F ++ F' |- r'i => r''i
By Soundness of Example Unevaluation on (1) (observing (6)), (7), (5), and (8)
  (9)    F ++ F' |- r''i |= exi
By Resumption Composition on (4) and (8)
  (10)   F ++ F' |- ri => r''i
Goal is given by Sat on (3), (10), and (9)
```

Theorem A.23 (Refine).

We only consider the most complicated case, Refine-Fix, in detail. The other cases are straightforward by similar reasoning.

```
Given:
  (1)    (Y |- ?? : T |= X) ->refine e -| { Y_i |- ??_h_i : T'_i |= X_i | i <= n }
  (2)    { F |- ??_h_i |= X_i | i <= n }

Goal:    F |- e |= X

By inversion of Refine on (1), assuming we go through Refine-Fix
  (3)    Filter(X) = { (E_j, {v_j -> ex_j}) | j <= m }
  (_)    h1 fresh
```

```
    (4)    e = fix f (\x . ??_h1)
    (5)    (Y_1 |- ??_h_1 : T'_1 |= X_1)
             = ((Y, f -> (T_1 -> T_2), x -> T_1) |- ??_h1 : T_2 |= X_1)
    (6)    X_1 = { ((E_j, f -> [E_j] fix f (\x . ??_h1), x -> v_j), ex_j) | j <= m }
  By inversion of Sat on (2), observing (5) and (6)
    (9)    (E_j, f -> [E_j] fix f (\x . ??_h1), x -> v_j) |- ??_h1 => r*j
    (10)   F |- r*j => r*'j
    (11)   F |- r*'j |= ex_j
  By E-Fix, observing (4)
    (12)   E_j |- e => [E_j] fix f (\x . ??_h1)
  By the resumption assumption
    (13)   F |- E_j => E_j'
  By R-Fix on (13)
    (14)   F |- [E_j] fix f (\x . ??_h1) => [E_j'] fix f (\x . ??_h1)
  By Simple Value Resumption
    (15)   F |- v_j => v_j
  By Idempotency of Resumption on (14)
    (16)   F |- [E_j'] fix f (\x . ??_h1) => [E_j'] fix f (\x . ??_h1)
  By the evaluation assumption
    (17)   (E_j', f -> [E_j'] fix f (\x. ??_h1), x -> v_j) |- ??_h1 => r**j
  By the resumption assumption
    (18)   F |- r**j => r**'j
  By the definition of environment resumption, (13), (14), and (15)
    (19)   F |- (E_j, f -> [E_j] fix f (\x. ??_h1), x -> v_j) =>
                   (E_j', f -> [E_j'] fix f (\x. ??_h1), x -> v_j)
  By Evaluation Respects Environment Resumption on (19), (9), (17), (10), and (18)
    (20)   r**'j == r*'j
  By R-App on (16), (15), (trivial), (17), and (18), observing (20)
    (21)   F |- ([E_j'] fix f (\x . ??_h1)) v_j => r*'j
  By XS-Input-Output on (21) and (11)
    (22)   F |- [E_j'] fix f (\x . ??_h1) |= {v_j -> ex_j}
  Goal is given by Sat on (12), (14), and (22),
    observing (3) and the fact that the filtered-out example constraints
    are trivially satisfied.
```

Theorem A.24 (Branch).

```
  Given:
    (1)    (Y |- ?? : T |= X) ->branch e' -| { Y_i |- ??_h_i : T'_i |= X_i | i <= n }
    (2)    { F |- ??_h_i |= X_i | i <= n }

  Goal:    F |- e' |= X

  By inversion of branch on (1)
    (3)    { C_i : T_i -> D | i <= n }
    (4)    (Y |- ?? : D) ~>guess e
    (_)    h_i fresh
    (5)    (Y_i |- ??_h_i : T'_i |= X_i) = ((Y, x_i -> T_i) |- ??_h_i : T |= X_i)
    (6)    X_i =
             { ((E_ij, x_i -> r_ij), ex_ij)
               | (E_ij, ex_ij) in Filter(X) and E_ij |- e => C_i r_ij -| - }
    (7)    Filter(X) = X_1 ++ ... ++ X_n
    (8)    e' = case e of { C_i x_i -> ??_h_i | i <= n }
  By inversion of Sat on (2), observing (6)
    (9)    (E_ij, x_i -> r_ij) |- ??_h_i => r*ij
```

```
  (10)   F |- r*ij => r*'ij
  (11)   F |- r*'ij |= ex_ij
By E-Case on (6) and (9)
  (12)   E_ij |- case e of { C_i x_i -> ??_h_i | i <= n } => r*ij
Goal is given by Sat on (12), (10), and (11)
  observing (7), (8), and the fact that the filtered-out example constraints
  are trivially satisfied
```

Theorem A.25 (Fill).

The Defer case is trivial.
The Refine,Branch case is straightforward by way of Theorem A.23 (refine) and Theorem A.24 (branch).
The Guess-And-Check case is straightforward by way of Theorem A.22 (check).

Theorem A.26 (Type soundness of merge).

Straightforward by way of induction and (eventually) Theorem A.14.

Technically, we must establish similar lemmas applying to Step and Resolve, but the definitions and proofs of these lemmas are straightforward.

Theorem A.27 (Example soundness of merge).

Straightforward by way of induction and (eventually) Theorem A.22 (check).

Technically, we must establish similar lemmas applying to Step and Resolve, but the definitions and proofs of these lemmas are straightforward.

Theorem A.28 (Soundness of solve).

```
Given:
  (1)    U : D
  (2)    F : D
  (3)    D |- Solve (U, F) ~> F' ; D''

Goal A:  F' : D''
Goal B:  F' |= (U, F)

By inversion of Solve on (3), going through Solve-One rule since
    Solve-Done is trivial
  (4)    h in U
  (5)    D(h) == (Y, T)
  (6)    U(h) == X
  (7)    F |- (Y |- ??h : T |= X) ~>fill K ; D'
  (8)    D ++ D' |- Merge((U / h, F) @ K) |> K'
  (9)    D ++ D' |- Solve(K') ~> F' ; D''
By definition of constraints typing on (1), (5), and (6)
  (10)   D |- X : Y ; T
By Type Soundness of Fill on (2), (10), and (7)
  (11)   K : D ++ D' ++ (h -> (Y, T))
By (11), observing (4) and (5)
  (T0)   K : D ++ D'
By observing that freshness premises ensure that D' is disjoint from D
  (T1)   U : D ++ D'
  (T2)   F : D ++ D'
By Type Soundness of Semantic Merge on (T0+T1+T2) and (8)
  (T3)   K' : D ++ D'
By the induction hypothesis on (T3), (T3), and (9)
  (12)   F' : D''
  (13)   F' |= K'
By Example Soundness of Semantic Merge on (13) and (8)
```

```
  (14)   F' |= (U / h, F) @ K
By the definition of constraint satisfaction and (14)
  (15)   F' |= (U / h, F)
  (16)   F' |= K
By Example Soundness of Fill on (7) and (16) (observing (15))
  (17)   F'(h) = e
  (18)   F' |- e |= X
By straightforward reasoning on (17) and (18)
  (19)   F' |- ??h |= X
Goal A is given by (12)
Goal B is given by combining (15), (6), and (19)
```

Theorem A.29 (Soundness of assertion simplification).
Straightforward by way of Theorem A.32.

Theorem A.30 (Soundness of synthesis).
Straightforward by way of Theorem A.20, Theorem A.19, Theorem A.28 (solve) and Theorem A.29.

Theorem A.31.
Straightforward induction.

Theorem A.32.

```
Given:
  (0a)   v simple value
  (0b)   r final
  (1)    - |- r <= v -| K
  (2)    F |= K

Goal:    F |- r => v

By the resumption assumption
  (3)    F |- r => r'
By Soundness of Example Unevaluation on (2), (0b), (1), and (3)
  (4)    F |- r' |= v
By Example Satisfaction of Simple Value on (0a) and (4)
  (5)    r' == v
Goal is given by (3), observing (5)
```

B Additional Experimental Data

Experiment	1		2		3	
Sketch	*None*				*Base Case*	
#Benchmarks	*37/43 Myth benchmarks*				*24/37*	
Objective	*Top-1*		*Top-1*		*Top-1-R*	
Name	**#Ex**	**Time**	**#Ex**	**Time**	**#Ex**	**Time**
bool_band	4	0.003	3 (75%)	0.003	—	—
bool_bor	4	0.003	3 (75%)	0.004	—	—
bool_impl	4	0.004	3 (75%)	0.004	—	—
bool_neg	2	0.001	2 (100%)	0.001	—	—
bool_xor	4	0.007	3 (75%)	0.007	—	—
list_append	6	0.006	5 (83%)	0.008	1 (17%)	0.013
list_compress	13	none	—	—	—	—
list_concat	6	0.007	3 (50%)	0.008	2 (33%)	0.011
list_drop	11	0.025	5 (45%)	0.015	2 (18%)	0.007
list_even_parity	7	overspec	—	—	—	—
list_filter	8	0.092	4 (50%)	0.073	overspec	
list_fold	9	0.697	3 (33%)	0.719	3 (33%)	2.475
list_hd	3	0.002	2 (67%)	0.003	—	—
list_inc	4	0.011	2 (50%)	0.009	—	—
list_last	6	0.006	4 (67%)	0.007	2 (33%)	0.005
list_length	3	0.002	3 (100%)	0.003	1 (33%)	0.003
list_map	8	0.036	4 (50%)	0.039	2 (25%)	0.717
list_nth	13	0.108	6 (46%)	0.025	2 (15%)	0.007
list_pairwise_swap	7	none	—	—	—	—
list_rev_append	5	0.094	3 (60%)	0.067	2 (40%)	0.046
list_rev_fold	5	0.028	2 (40%)	0.025	—	—
list_rev_snoc	5	0.008	3 (60%)	0.016	1 (20%)	0.046
list_rev_tailcall	8	0.006	3 (38%)	0.006	1 (13%)	0.015
list_snoc	8	0.012	4 (50%)	0.010	2 (25%)	0.006
list_sort_sorted_insert	7	0.012	3 (43%)	0.012	1 (14%)	0.043
list_sorted_insert	12	5.557	7 (58%)	2.589	overspec	
list_stutter	3	0.002	2 (67%)	0.004	1 (33%)	0.004
list_sum	3	0.021	2 (67%)	0.020	—	—
list_take	12	0.061	6 (50%)	0.040	3 (25%)	0.011
list_tl	3	0.002	2 (67%)	0.003	—	—
nat_add	9	0.005	4 (44%)	0.005	1 (11%)	0.007
nat_iseven	4	0.003	3 (75%)	0.003	2 (50%)	0.002
nat_max	9	0.035	9 (100%)	0.035	4 (44%)	0.030
nat_pred	3	0.001	2 (67%)	0.001	—	—
tree_binsert	20	timeout	—	—	—	—
tree_collect_leaves	6	0.062	3 (50%)	0.042	2 (33%)	0.019
tree_count_leaves	7	2.885	3 (43%)	1.112	1 (14%)	0.066
tree_count_nodes	6	0.292	3 (50%)	0.172	2 (33%)	0.068
tree_inorder	5	0.101	4 (80%)	0.092	2 (40%)	0.021
tree_map	7	0.048	4 (57%)	0.047	3 (43%)	0.675
tree_nodes_at_level	11	timeout	—	—	—	—
tree_postorder	20	timeout	—	—	—	—
tree_preorder	5	0.126	3 (60%)	0.129	2 (40%)	0.024
Averages			61%*		29%	

Figure 17. Experiments.

(1) Baseline: #Examples required by Myth [29]; No Sketch.
(2) #Examples required by Sketch-n-Myth; No Sketch. (2) #Examples required by Sketch-n-Myth; Base Case Sketch.

Top-1: *1st solution valid.* ***Top-1-R***: *1st recursive solution valid.* ***#Ex***: *Percentage compared to baseline #examples in parentheses.*
Time: *Avg. of 5 runs, in seconds.* ***Averages***: *Non-blank rows.* *61% for 37 benchmarks. (Upper bound: 67% for all 43.)*

C Additional Discussion

Live Evaluation. We borrow the technique for partially evaluating sketches from Hazelnut Live [28]. We note several technical differences in our formulation.

We choose a natural semantics presentation [21] 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 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 [37, 38], a language feature orthogonal to the (expression) synthesis motivations for our work. Omar et al. [28] present a bidirectional type system [6, 34] that, given type-annotated functions, computes hole environments Δ; 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. [33] 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 [24] propose a bidirectional evaluator—which forms a *lens* [11]—for manipulating first-order values in a language of *residual expressions*, containing no function applications in elimination positions. Mayer et al. [25] 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.

Assertions in Arbitrary Program Positions. To allow `asserts` to appear arbitrarily in expressions e, evaluation could be extended to $E \vdash e \Rightarrow r \dashv A$, generating assertions A as a side-effect.

$$\text{[T-Assert]}\ \frac{\Sigma\,;\Delta\,;\Gamma \vdash e_1 : T \qquad \Sigma\,;\Delta\,;\Gamma \vdash e_2 : T}{\Sigma\,;\Delta\,;\Gamma \vdash \texttt{assert}\ (e_1 = e_2) : ()} \qquad \text{[E-Assert]}\ \frac{E \vdash e_1 \Rightarrow r_1 \dashv A_1 \qquad E \vdash e_2 \Rightarrow r_2 \dashv A_2 \qquad r_1 \equiv_{A_3} r_2}{E \vdash \texttt{assert}\ (e_1 = e_2) \Rightarrow () \dashv A_1 +\!\!+ A_2 +\!\!+ A_3}$$

For existing evaluation rules, assertions would simply be propagated from premises to conclusions. Because resumption relies on evaluation, it would also be extended $F \vdash r \Rightarrow r' \dashv A$ to propagate assertions, in a straightforward way. Because live bidirectional example checking relies on resumption, it would need to use $\mathit{Simplify}(A)$ to generate unevaluation constraints K.

Besides these algorithmic changes, the appropriate definition of assertion satisfaction, and the proofs of appropriately modified soundness theorems, are left to future work. The set of assertions generated by resumption is sensitive to the order in which holes are filled and subterms are resumed, making it difficult to establish naïvely extended properties about the generated assertions. Furthermore, evaluation or resumption may generate extraneous assertions that are unrelated to provided input-output examples, which makes them difficult to satisfy. Future work may need to develop dependency-tracking machinery, beyond the scope of techniques we considered in this paper.

Future Work. We believe that granularly interleaving synthesis with evaluation, providing "live" feedback throughout the program development process, will contribute to the vision of a truly usable programmer's assistant [43]. 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 [2] and feedback to label desirable and undesirable parts of candidate solutions [32]. Heuristics and ranking algorithms—taking into account existing code repositories (e.g. [8, 18]) 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.