Example-Directed Synthesis: A Type-Theoretic Interpretation

Jonathan Frankle
Princeton University, USA
jfrankle@cs.princeton.edu

Peter-Michael Osera
Grinnell College, USA
osera@cs.grinnell.edu

David Walker
Princeton University, USA
dpw@cs.princeton.edu

Steve Zdancewic
University of Pennsylvania, USA
stevez@cis.upenn.edu

Abstract

Input-output examples have emerged as a practical and user-friendly specification mechanism for program synthesis in many environments. While example-driven tools have demonstrated tangible impact that has inspired adoption in industry, their underlying semantics are less well-understood: what are "examples" and how do they relate to other kinds of specifications? This paper demonstrates that examples can, in general, be interpreted as refinement types. Seen in this light, program synthesis is the task of finding an inhabitant of such a type. This insight provides an immediate semantic interpretation for examples. Moreover, it enables us to exploit decades of research in type theory as well as its correspondence with intuitionistic logic rather than designing ad hoc theoretical frameworks for synthesis from scratch.

We put this observation into practice by formalizing synthesis as proof search in a sequent calculus with intersection and union refinements that we prove to be sound with respect to a conventional type system. In addition, we show how to handle negative examples, which arise from user feedback or counterexample-guided loops. This theory serves as the basis for a prototype implementation that extends our core language to support ML-style algebraic data types and structurally inductive functions. Users can also specify synthesis goals using polymorphic refinements and import monomorphic libraries. The prototype serves as a vehicle for empirically evaluating a number of different strategies for resolving the nondeterminism of the sequent calculus—bottom-up theorem-proving, term enumeration with refinement type checking, and combinations of both—the results of which classify, explain, and validate the design choices of existing synthesis systems. It also provides a platform for measuring the practical value of a specification language that combines "examples" with the more general expressiveness of refinements.

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

General Terms Languages, Theory

Keywords Functional Programming, Proof Search, Sequent Calculus, Program Synthesis, Type Theory, Refinement Types

1. Introduction

One of the great barriers to productivity in modern life is the rate at which we can write programs to automate repetitive tasks. Program synthesis—the act of generating a program from a high-level specification of a computational problem—offers the tantalizing prospect that much of this labor can, itself, be automated. To make synthesis worthwhile, however, specifying a program must be easier than simply writing it. Fortunately, in many domains, *examples* are readily available and require little work to marshal into a synthesis specification. For instance, spreadsheets already contain data that can form the basis of examples, an observation that Flash Fill exploits to synthesize transformations for the more than 1 billion worldwide users of Microsoft Office [14, 21]. Other research extends these ideas to a number of domains, including ad hoc data processing [10] and editor macro creation [19].

Yet, even with increasing industrial and academic attention devoted to example-directed synthesis, key theoretical questions remain unanswered. Most fundamentally, what exactly *are* "examples?" That is, how do we understand the semantics of examples as specifications in the context of existing formal systems? An answer to this question would do far more than satisfy academic curiosity—it could help us understand the meta-theoretic properties of synthesis procedures and inform the design of more efficient synthesis engines. It would also have explanatory powers, revealing the relationships between existing, ad hoc approaches to example-directed synthesis.

In this paper, we give such an answer, demonstrating that example-based specifications can be viewed as the *refinement types* first studied by Freeman and Pfenning [13]. Not only can refinements be given direct semantics using standard, type-based logical relations, but they can also be interpreted logically using the Curry-Howard isomorphism. Program synthesis thereby becomes the task of theorem-proving in intuitionistic logic, enabling us to exploit decades of research into algorithms and data structures for doing so efficiently (see Pfenning's course notes for an overview [23]). Furthermore, both type theory and logic suggest new ways to extend our specification language with more powerful mechanisms than in traditional example-directed synthesis, giving users a wider vocabulary with which to communicate intentions to the synthesizer.

We design a new theoretical framework for example-directed synthesis from the ground up based on these insights. We encode traditional examples with singleton types, function types, and intersections and illustrate how to add unions—for more general specifications—and negation—for counterexamples. While type systems are usually presented in natural deduction style, theorem-proving for intuitionistic logics often relies on sequent calculi. Guided by the literature [23], we develop such a sequent calculus and prove it sound (though not complete) with respect to a largely standard natural-deduction type system [4, 7].

The dividends of this type-theoretic interpretation of examples are more than just theoretical: we implement a new synthesis engine built on these principles. Given a refinement, it uses focused proof search in the sequent calculus to find an inhabitant over a pure subset of ML with algebraic datatypes, structurally inductive functions, and parametric polymorphism. It is also able to make use of monomorphic library functions. Our synthesizer is competitive with state-of-the-art example-directed systems, generating 51 simple functional programs with refinements that correspond closely to conventional "examples." Furthermore, using the far richer language of refinements, we were able to condense the specifications of 31 of these programs, commonly seeing decreases of 10-20% compared to previous work [22] (as measured in refinement abstract syntax tree nodes). Those programs that could be rewritten polymorphically saw even more dramatic reductions sometimes nearing 75%.

Although the sequent calculus and focusing remove many nondeterministic choices from the proof search process, the performance of the system as a whole hinges on the manner in which we handle those that remain. We evaluate four strategies for doing so, systematically exploring the design space using our formal understanding of the underpinnings of our prototype synthesizer. These experiments enable us to classify, validate, and explain the results of MYTH [22], whose seemingly simple approach (forced upon it by theoretical restrictions) proved to be most efficient in practice.

One of the limitations of our approach is that it is incomplete. The system does not automatically generate new "helper functions"—auxiliary recursive functions that take additional or different arguments. It also does not infer instantiation of polymorphic library functions, and the theoretical framework restricts the combination of union and intersection types. These limitations suggest avenues for future research based on this framework.

In summary, the central contributions of this work are:

- We observe and exploit the relationship between example-directed program synthesis, refinement type systems, and intuitionistic logic. (Section 2)
- We formalize a *synchronized sequent calculus* with intersection, union and singleton types to define the parameters of the synthesis problem. We prove this sequent calculus to be sound with respect to a more standard unsynchronized sequent calculus and then relate the unsynchronized sequent calculus to a natural deduction-style proof system. (Section 3)
- We develop a prototype implementation of our program synthesis procedure that extends the core calculus with a variety of additional features. We describe design choices and experimentally analyze strategies for resolving the non-determinism of the sequent calculus. We discuss methods for integrating existing functions (*libraries*) into the synthesis process. We re-specify existing benchmarks to demonstrate the practical benefits of constraining synthesis problems with refinements. (Section 4)
- We reinterpret other example-directed synthesis systems through the lens of refinements and proof search. (Section 5)

2. Synthesis with Refinements

2.1 What Are Examples?

Suppose we wish to describe the behavior of the list `length` function using a series of examples as we might in any number of example-directed synthesis frameworks. In accordance with the function's type, each example must specify a mapping from a list to a natural number. For instance, we might say that `length` maps the empty list $[\,]$ to the number zero 0.

But, semantically, what exactly are these "examples?" We interpret them as *singleton types*—types with exactly one inhabitant. For instance, the type 0 contains only the value 0.[1] When we attempted to describe list `length`, however, we were not looking for inputs or outputs in isolation. Instead, we sought a description of a *function* that maps specific inputs to outputs. To do so, we can use a function type in tandem with singletons: $[\,] \to 0$.

Typically, we wish to combine more than one example when we specify a function's behavior. For instance, in addition to mapping $[\,]$ to 0, `length` also maps $[1]$ to 1 and $[2; 1]$ to $[2]$. In other words, `length` satisfies the conjunction of three constraints:

$$[\,] \to 0 \;\;\wedge\;\; [1] \to 1 \;\;\wedge\;\; [2;1] \to 2$$

In type theory, conjunction typically takes two possible forms: intersection types and products. Since we want the same function to satisfy all three constraints, we need an intersection.

In summary, a specification language with singleton types, function types, and intersections captures the basic lexicon of example-directed synthesis and serves as an effective starting point for specifying function behavior using examples. Together, these components form the basis of a *refinement* type system [13].

2.2 Synthesis By Example

Interpreting examples as refinements, synthesis is the task of searching for an inhabitant of a type over this richer type system. The Curry-Howard isomorphism suggests another reading: treat the type as a theorem and search for a proof. Automated theorem-proving is a well-studied problem with an expansive body of existing research from which we can draw to design synthesis systems. While we could attempt proof search over a conventional, natural-deduction style inference system with rules for introduction and elimination forms, doing so would require us to mix elements of forward and backward search. In this paper, we instead choose to work over the *sequent calculus*. A sequent $\langle \Gamma \vdash r \rangle \rightsquigarrow e$ states that, given a goal refinement r and a context Γ binding names to refinements, we can synthesize the expression e. The sequent calculus uses *left rules* to simplify assumptions in the context and *right rules* to advance the goal. The left rules correspond to elimination forms, allowing us to apply a function or project the elements of a pair. The right rules allow us to construct our goal—*i.e.,* introduce a lambda expression or a pair. This sequent formulation allows us to perform pure bottom-up search.

To begin our running example, we specify an initial theorem-proving problem for `length` below. The black square indicates a hole that we would like to fill with a synthesized program.

$$\mathsf{let\ \ len} \;=\; \langle \cdot \vdash [\,] \to 0 \;\;\wedge\;\; [1] \to 1 \;\;\wedge\;\; [2;1] \to 2 \rangle \rightsquigarrow \blacksquare$$

Our first step is to apply the sequent calculus right rule for intersection, which directs us to break up our refinement into three separate sequents, or *worlds*, one for each constituent of our larger refinement. Each of these worlds should generate the same expression, which is the solution to the overall synthesis problem:

$$\mathsf{let\ \ len} \;=\; \langle \cdot \vdash [\,] \to 0 \rangle, \langle \cdot \vdash [1] \to 1 \rangle, \langle \cdot \vdash [2;1] \to 2 \rangle \rightsquigarrow \blacksquare$$

Since every goal refinement is at arrow type, we can apply the right rule for arrows, synthesizing a function.

$$\mathsf{let\ \ len} \;=\; \mathsf{fun\ x} \to \langle \mathsf{x} : [\,] \vdash 0 \rangle, \langle \mathsf{x} : [1] \vdash 1 \rangle, \langle \mathsf{x} : [2;1] \vdash 2 \rangle \rightsquigarrow \blacksquare$$

In creating a function, we have broken up our goal refinements into separate argument and result components. The result part becomes our new goal refinement, while the argument part is bound to a variable that we add to the context—the argument of the function we synthesized. Notice that x takes on a different value in every world. We might read the first sequent as, "when x has type $[\,]$, the expression we synthesize should have type 0."

[1] Rather than use special syntax to distinguish a singleton type from its corresponding value, we rely on context to do so.

At this point, our goals are all singleton refinements, granting us a number of choices in how we proceed. In practice, making the correct decision at this juncture in a performance-friendly manner proved to be the most difficult aspect of designing our prototype synthesizer. We spend the majority of Section 4 exploring this question. In the context of list `length`, supposing that natural numbers are defined as either zero (Z) or the successor of a natural number (S nat), one option is to apply the right rule for sum types, which would synthesize a constructor. Alternatively, we could try to use a variable from the context to solve our synthesis goal.

Notice, however, that x takes on two different forms: in the first world, it is the empty list, while in the second and third it contains an element. We can apply the left rule for sum types, pattern matching on x and creating one synthesis subproblem for each branch.

```
let  len  = fun x → match x with
   | [ ] → ⟨· ⊢ 0⟩ ⇝ ■₁
   | h :: t  → ⟨h : 1, t : [ ] ⊢ 1⟩, ⟨h : 2, t : [1] ⊢ 2⟩ ⇝ ■₂
```

When we pattern match, our worlds are partitioned between the two synthesis subproblems depending on the value x takes on. Worlds where x is the empty list are sent to the [ ] branch while worlds where x is non-empty go to the other branch. The names created from pattern-matching on x are added to the context in each world and x is removed. If we need x later, we can always reconstruct it.

In the first branch, there is a lone world whose goal refinement is a singleton. We can apply the right rule for sum types and synthesize the singleton's constructor, Z, to solve the synthesis problem. In the second branch, too, each of the goal refinements shares the same constructor, so we synthesize S and strip away one layer of the constructor from each of the goal refinements.

```
let  len  = fun x → match x with
   | [ ] → Z
   | h :: t  → S ⟨h : 1, t : [ ] ⊢ 0⟩, ⟨h : 2, t : [1] ⊢ 1⟩ ⇝ ■₂
```

To solve the second branch, it would be useful to make a recursive call to len. To do so, however, we need to add len and refinements describing it to the context. Which refinements do we use? The original synthesis problem, which is a specification of len that we have already written. Once added, we can use the left arrow rule to apply len to t in the context.[2]

```
let  rec  len  = fun x → match x with
   | [ ] → Z
   | h :: t  → S (let y = len t in
      ⟨h : 1, t : [ ], y : 0 ⊢ 0⟩, ⟨h : 2, t : [1], y : 1 ⊢ 1⟩ ⇝ ■₂)
```

We let-bind the application to a fresh name, y, to ensure that our context contains only variables. The application produces exactly the refinements we need to solve our goal, so we use the sequent calculus context rule to synthesize y and, finally, unwind the let-binding to produce our result:

```
let  rec  len  = fun x → match x with [ ] → Z | h::t → S (len t)
```

2.3 Richer Specifications

So far, we have seen that singletons, arrows, and intersections are expressive enough to capture example-directed synthesis and specify useful synthesis problems. Refinements, however, are not limited to just these components—we can export many other features to our synthesis specification language. We select two features in particular: *base types* and *unions*. We treat any base type as a valid refinement, meaning that we can describe, for example, the set of all non-empty lists by writing Cons (nat × list). With unions, we can combine arbitrary collections of refinements into larger types. For example, we could write the type of even numbers less than five as $0 \vee 2 \vee 4$. This richer refinement language grants us new forms of expression that can dramatically reduce the size of synthesis specifications for large classes of problems:

Eliminating redundancy. Disjunction enables us to condense cases of functions that map different inputs to the same output. For example, without disjunction the `decrement` function could be described with three cases: $0 \to 0 \wedge 1 \to 0 \wedge 2 \to 1$. Using disjunction, we can merge the first two cases together into $(0 \vee 1) \to 0$. When the arguments share constructors, we can structurally push this disjunction into the argument itself, further reducing the specification size. For instance, the lists [0; 1] and [0] could be combined into $\mathsf{Cons}(0 \times (\mathsf{Nil} \vee \mathsf{Cons}(1 \times \mathsf{Nil})))$.

Type refinements enable an even more powerful form of generalization, making it possible to express statements about all inhabitants of a type that obey certain structural properties. When specifying the list `map` function, we could write the higher-order argument that maps every natural number to 0 as $\mathsf{nat} \to 0$ rather than enumerating a list of cases. Similarly, we could write the function that tests whether a number is equal to 1 (useful in the specification of list `filter`) as $(1 \to \mathsf{true}) \wedge ((\mathsf{Z} \vee \mathsf{S\ S\ nat}) \to \mathsf{false})$.

Reducing excess information. Many specifications have more information than is strictly necessary to synthesize the intended function. For example, describing list `length` previously required us to choose values for the contents of example lists $([2; 1] \to 2)$ even though these values have no impact on the result of the function. Type refinements make it possible to express a form of "weak polymorphism" $([\mathsf{nat}; \mathsf{nat}] \to 2)$ where we are agnostic to the particular inhabitants of a type. Other examples require the synthesizer to differentiate between constructors but not explicit values. To generate list `unzip`, the synthesizer needs to know only that the structure of an example list is $[\mathsf{S\ nat} \times \mathsf{Z}; \mathsf{Z} \times \mathsf{S\ nat}]$, with different constructors, rather than any exact numbers, in each pair.

Simple properties. We can generalize much of the behavior of many functions by writing simple properties into their refinements. Consider the list `nth` function, which, given a list and an index, returns an option containing either the item at that index (if the list is long enough) or None (if the list is not). One property of this function is that it will always return None on an empty list: $\mathsf{Nil} \to \mathsf{nat} \to \mathsf{None}$. It will do the same on a list of length 1 when the index is greater than 0: $\mathsf{Cons}(\mathsf{nat} \times \mathsf{Nil}) \to \mathsf{S\ nat} \to \mathsf{None}$. These properties replace numerous instantiations as specialized to particular lists or numbers, greatly reducing specification sizes.

Parametric polymorphism. In our implementation, we also provide polymorphism, which furthers many of the benefits already discussed. Intuitively, synthesis benefits from the "free theorems" that polymorphic types imply [28]. Based on type information alone, we could even synthesize some programs, such as `compose`, without any refinements at all in a manner similar to Djinn for Haskell [3]. For other programs, polymorphism reduces the degrees of freedom available to a synthesizer, preventing it from introspecting into abstracted types. This dramatically diminishes the size of the search space that the synthesizer must explore while granting the user more general specification mechanisms. Far smaller refinements could describe polymorphic programs—in the case of list `map`, more than 66% smaller than with singletons alone.

2.4 Negation

Sometimes, the synthesizer fails to produce the program we want on the first attempt. For instance, provided the description of list `length` we crafted in Section 2.1 $([\,] \to 0 \wedge [1] \to 1 \wedge [2; 1] \to 2)$, the synthesizer could generate the following:

```
let  len  = fun x → match x with [ ] → Z | h::t → h
```

[2] For the sake of readability, we have elided len and its refinements from each of the contexts.

This program, the list `head` function, conforms to our specification and is even smaller than the program we were expecting—it is a perfectly reasonable, if unexpected, result. We found this experience to be quite common—synthesis is rarely a one-step process. Instead, it involves multiple rounds of iteration as a user gradually refines a specification until it is sufficiently precise.

Many existing synthesis systems provide for this contingency. Flash Fill [14] allows users to correct erroneously transformed spreadsheet cells, updating the synthesis specification with this additional information. Counterexample-guided inductive synthesis (CEGIS) [27] expects that the synthesizer will err and relies on this process to slowly construct a specification. We offer a similar capability in the form of negation, which makes it possible to articulate that a program is *not* an inhabitant of a particular refinement—to write a counterexample. To appropriately constrain the synthesizer on list `length`, for example, we might add the refinement $[0] \to \mathsf{not}(0)$ to our specification. Internally, we normalize negation into other refinements according to logical identities, meaning we do not have to make provision for it in our synthesis procedure.

Like disjunction and type refinements, negation is convenient shorthand that makes specifying programs easier. Yet it also enables new modes of interaction with the synthesizer, facilitating the iterative process of real-world synthesis and laying the groundwork for type-directed CEGIS, in which we replace the user with an oracle that drives synthesis with counterexamples.

3. Type Theory of Example-Directed Synthesis

In this section, we present the syntax and synthesis rules for a lambda calculus with intersections, unions, singletons and other basic type constructors—a type system sufficient to represent the constraints imposed by example-based specifications. The technical development proceeds in four stages. First, we present an (almost) standard natural-deduction style type system drawn from the work of Dunfield [7] and Barbanera [4]. This type system serves as the foundation for our work, and corresponds to a familiar presentation of the necessary type-theoretic constructs.

Second, we introduce an *unsynchronized* sequent calculus that more closely mirrors our implementation in its use of bottom-up left-rules rather than natural-deduction style top-down elimination rules. This sequent calculus also generates terms in a more restricted form—a kind of "A-Normal" form. Our A-Normal form represents the step-by-step choices made by the synthesis algorithm. It has the property that when let expressions are reduced, the resulting terms will find themselves in eta-long, beta-normal form. We prove it sound with respect to the natural deduction style formulation of the system. However, because the sequent calculus is unsynchronized, it suggests a proof strategy that uses more backtracking than is necessary.

Hence, the third step of our development presents a *synchronized* sequent calculus. This third calculus most closely resembles our implementation, though the correspondence is not perfect: our implementation uses memoization, focusing and additional optimizations that we have not formalized. We discuss these extensions informally in Section 4. The last step shows how to integrate negative examples into the system using a normalization procedure that transforms refinement goals prior to beginning synthesis itself.

3.1 Step 0: Syntax

Figure 1 presents the basic syntactic elements of our language. The refinements (r) we use include singleton refinements `true` and `false` at base type together with function and pair refinements ($r_1 \to r_2$ and $r_1 \times r_2$) at higher type. We generate more complex refinements using the bottom (empty) refinement $\bot$ as well as intersections and unions. Contexts Γ map variables to refinements; throughout this document, we maintain the invariant that no variable appears more than once in Γ. The language of expressions e, and its (call-by-value) semantics, are standard.

$$
\begin{array}{lrcl}
\text{(External Libraries)} & c & \in & \Psi \\
\text{(Base Values)} & b & ::= & \mathsf{true} \mid \mathsf{false} \\
\text{(Refinements)} & r & ::= & b \mid r_1 \to r_2 \mid r_1 \times r_2 \\
 & & \mid & \bot \mid r_1 \wedge r_2 \mid r_1 \vee r_2 \\
\text{(Values)} & v & ::= & c \mid b \mid \lambda x.e \mid \langle v_1, v_2 \rangle \\
\text{(Expressions)} & e & ::= & x \mid c \\
 & & \mid & b \mid \mathsf{if}\, e_0 \,\mathsf{then}\, e_1 \,\mathsf{else}\, e_2 \\
 & & \mid & \lambda x.e \mid e_1\ e_2 \\
 & & \mid & \langle e_1, e_2 \rangle \mid \pi_1\ e \mid \pi_2\ e \\
 & & \mid & \mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2 \\
\text{(Type contexts)} & \Gamma & ::= & \cdot \mid \Gamma, x : r
\end{array}
$$

Figure 1: Refinement Syntax

$$
\begin{array}{lll}
\boxed{\mathcal{V}[\![r]\!]} & \mathcal{V}[\![\bot]\!] & = \emptyset \\
 & \mathcal{V}[\![\mathsf{true}]\!] & = \{\mathsf{true}\} \\
 & \mathcal{V}[\![\mathsf{false}]\!] & = \{\mathsf{false}\} \\
 & \mathcal{V}[\![r_1 \to r_2]\!] & = \{v \mid \forall v_1 \in \mathcal{V}[\![r_1]\!] : v\, v_1 \in \mathcal{E}[\![r_2]\!]\} \\
 & \mathcal{V}[\![r_1 \wedge r_2]\!] & = \mathcal{V}[\![r_1]\!] \cap \mathcal{V}[\![r_2]\!] \\
 & \mathcal{V}[\![r_1 \vee r_2]\!] & = \mathcal{V}[\![r_1]\!] \cup \mathcal{V}[\![r_2]\!] \\
 & \mathcal{V}[\![r_1 \times r_2]\!] & = \{\langle v_1, v_2 \rangle : v_1 \in \mathcal{V}[\![r_1]\!], v_2 \in \mathcal{V}[\![r_2]\!]\} \\
 & & \\
\boxed{\mathcal{E}[\![r]\!]} & \mathcal{E}[\![r]\!] & = \{e \mid e \longrightarrow^* v \text{ and } v \in \mathcal{V}[\![r]\!]\} \\
 & & \\
\boxed{\mathcal{C}[\![r]\!]} & \mathcal{C}[\![r_1 \to r_2]\!] & = \{c \mid \forall v_1 \in \mathcal{V}[\![r_1]\!] : \mathcal{I}(c, v_1) \in \mathcal{V}[\![r_2]\!]\}
\end{array}
$$

Figure 2: Semantics of Refinements and Constants. $\mathcal{I}(c, v_1)$ is a function from constants and values to values. $e \longrightarrow^* v$ is the standard multi-step call-by-value operational relation.

3.2 Step 1: Natural Deduction

Selected rules from the natural deduction-style formulation of our type system appear in Figure 3. It comprises two judgement forms: subtyping ($r_1 \le r_2$) and type checking ($\Gamma \vdash e : r\ \mathsf{nd}$). There are many ways to formulate subtyping. For concreteness, we use rules from Dunfield's thesis [7]; because they have limited bearing on the remainder of our presentation they are elided. As is standard in natural deduction, typing is formulated using introduction (I) and elimination (E) rules. We elide the most familiar rules (those for pairs and functions) and focus on intersections and unions:

- Intersections and pairs are both forms of conjunction. The difference is that intersection uses the same proof term for both conjuncts, without extra syntax introducing the intersection, whereas pairs are introduced with new syntax and have separate proof terms for each conjunct. From a type *checking* perspective (when both types and expressions are supplied), intersections and pairs are equally easy. From a synthesis perspective, intersections are more challenging as one must find a single expression that satisfies two constraints, as opposed to two separate expressions that independently satisfy one constraint each.
- The elimination rules for unions are notoriously difficult. We have chosen a rule drawn from [4]. This rule states that we must first identify a subexpression e' with union type $r_1 \vee r_2$ and replace it with x. If the new expression has type r under both the hypotheses $x : r_1$ and $x : r_2$ then $e[e'/x]$ also has type r.

The most interesting rule is that for constants (ND-SAMPLE). Constants represent libraries of pre-existing code supplied before synthesis is to begin. Typically, a pre-defined signature ascribes a single type or refinement to each such constant. However, we allow any refinement consistent with the operational semantics of the constant to describe instance it is used. In practice, we use this rule in two ways.

1. We type check each library function using a standard simple type system and give it a refinement corresponding to its simple type. This type structure suffices to synthesize the shape of the eta-long, beta-normal forms that use the constant.
2. When necessary, we generate a concrete argument v and execute $c\ v$. If the result is v', we ascribe the refinement $r \rightarrow r'$, where r describes v and r' describes v', to the library function.

As an example, suppose a user supplies a library containing `xor`. It is trivial to type check `xor` to deduce that it has the simple refinement $\mathsf{Bool} \rightarrow \mathsf{Bool} \rightarrow \mathsf{Bool}$.[3] The structural information in this simple refinement suffices to guide a portion of our synthesis algorithm. However, we may need to extract additional information about `xor` to apply it during synthesis—its simple type may not suffice. We do so by executing (*sampling*) it on well-typed arguments to generate additional refinements. For instance, we could execute `xor` $\mathsf{true}\ \mathsf{false}$, see that it returns true, and produce the refinement $\mathsf{true} \rightarrow \mathsf{false} \rightarrow \mathsf{true}$ that is consistent with `xor`'s semantics. In our implementation, we sample on-demand when additional refinements could aid the proof search process, a technique that differentiates our system from much past work on pure theorem proving.

3.3 Step 2: Unsynchronized Sequents

The move from natural deduction to the unsynchronized sequent calculus is the first step towards a more algorithmic presentation of proof search (program synthesis) strategy. We formalize this system as a synthesis procedure with the form $\langle \Gamma \vdash r \rangle \rightsquigarrow s\ \mathsf{unsync}$ which may be read as "Given a context Γ and a goal refinement r, we synthesize a sequent-normal expression s." Sequent-normal expressions are a series of let expressions, each of which represents the synthesis algorithm's choice of which connective from the context to deconstruct. For example, the expression $\mathsf{let}\, x_2 = x_1\ s_1 \,\mathsf{in}\, s_2$ represents the choice to try to apply x_1 (which must have function type) to some other normal form s_1 and then to continue with s_2. The other sequent-normal forms are listed below.

$$
\begin{array}{rcl}
s & ::= & x \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\, x \,\mathsf{then}\, s_1 \,\mathsf{else}\, s_2 \\
 & \mid & \lambda x.s \mid \mathsf{let}\, x_2 = x_1\ s_1 \,\mathsf{in}\, s_2 \\
 & \mid & \langle s_1, s_2 \rangle \mid \mathsf{let}\, \langle x_1, x_2 \rangle = x \,\mathsf{in}\, s \\
 & \mid & \mathsf{let}\, x = c \,\mathsf{in}\, s
\end{array}
$$

These expressions may be seen as a subset of the natural deduction expressions by interpreting $\mathsf{let}\, \langle x_1, x_2 \rangle = x \,\mathsf{in}\, s$ as an abbreviation for $\mathsf{let}\, x_1 = \pi_1\ x \,\mathsf{in}\, \mathsf{let}\, x_2 = \pi_2\ x \,\mathsf{in}\, s$. Together, the syntax and sequent calculus typing rules enforce the invariant that, upon reduction of the let expressions, there are no further beta-reductions in the expression and all expressions are in eta-long form. For instance, the expression

$$\lambda x.\mathsf{let}\, \langle x_1, x_2 \rangle = x \,\mathsf{in}\, \mathsf{let}\, x_3 = x_1\ x_2 \,\mathsf{in}\, x_3$$

reduces to the beta-normal, eta-long expression $\lambda x.(\pi_1\ x)\ (\pi_2\ x)$ Searching for eta-long, beta-normal expressions can cut down the size of the synthesis search space by orders of magnitude.

Figure 4 presents the unsynchronized sequent calculus rules. For each connective, there are right rules, corresponding to the natural deduction introduction rules, and left rules, corresponding to (let-bound) instances of the elimination rules. For example, the left rule for pairs (U-LPAIR) shows how to deconstruct a hypothesis of the form $x : r_a \times r_b$. One does so by extracting the two components of the pair (x_1 and x_2) and continuing the computation with two new hypotheses $x_1 : r_a$ and $x_2 : r_b$ in the context.

The context rule (U-CTX) allows us to use a hypothesis from the context so long as it has base type (true, false or a disjunction here). This constraint is standard and forces terms into eta-long form—we must decompose any complex types before use.

The sample rule (U-SAMPLE) inherits the properties of the natural deduction-style formulation in that any refinement r' consistent with the semantics of library function may be ascribed to c.

Soundness The unsynchronized sequent calculus is sound with respect to the natural deduction type system.

Theorem 1 (Unsynchronized Sequent Calculus Soundness)**.** *If* $\langle \Gamma \vdash r \rangle \rightsquigarrow s\ \mathsf{unsync}$ *then* $\Gamma \vdash s : r\ \mathsf{nd}$.

A sketch of this proof appears in the extended version [12].

3.4 Step 3: Synchronized Sequents

While the unsynchronized sequent calculus is a concrete step toward an implementation, a direct realization of these rules will lead to significant backtracking and poor performance. For instance, consider a use of the right intersection rule followed by the creation of a function:

$$
\dfrac{\dfrac{\dfrac{\mathcal{D}\\ \vdots}{\langle \Gamma, x : r_{a1} \vdash r_{b1} \rangle \rightsquigarrow s}}{\langle \Gamma \vdash r_{a1} \rightarrow r_{b1} \rangle \rightsquigarrow \lambda x.s} \qquad \dfrac{\dfrac{\mathcal{E}\\ \vdots}{\langle \Gamma, x : r_{a2} \vdash r_{b2} \rangle \rightsquigarrow s}}{\langle \Gamma \vdash r_{b1} \rightarrow r_{a2} \rangle \rightsquigarrow \lambda x.s}}{\langle \Gamma \vdash r_{a1} \rightarrow r_{b1} \wedge r_{a2} \rightarrow r_{b2} \rangle \rightsquigarrow \lambda x.s}
$$

If our proof search strategy is to first finish $\mathcal{D}$ before starting on $\mathcal{E}$, we may waste effort synthesizing a complete program using $\mathcal{D}$ that has a single subexpression inconsistent with $\mathcal{E}$. Processing both derivations simultaneously allows us to find errors earlier. Hence, our synchronized strategy collects up constraints and pushes them up the derivation tree in unison. Pictorially:

$$
\dfrac{\dfrac{\dfrac{\vdots}{\langle \Gamma, x : r_{a1} \vdash r_{b1} \rangle, \langle \Gamma, x : r_{a2} \vdash r_{b2} \rangle \rightsquigarrow s}}{\langle \Gamma \vdash r_{a1} \rightarrow r_{b1} \rangle, \langle \Gamma \vdash r_{a2} \rightarrow r_{b2} \rangle \rightsquigarrow \lambda x.s}}{\langle \Gamma \vdash r_{a1} \rightarrow r_{b1} \wedge r_{a2} \rightarrow r_{b2} \rangle \rightsquigarrow \lambda x.s}
$$

In general, synchronized sequents have the form $\mathcal{W} \rightsquigarrow s\ \mathsf{sync}$ where $\mathcal{W}$ is a multi-set of *worlds* and each world w has the form $\langle \Gamma \vdash r \rangle$. The key constraint in the synchronized sequent calculus is that all of the worlds generate exactly the same expression s. Hence, one barrier to achieving synchronization arises in processing the U-LAND rule. Here the difficulty is that, in order to synthesize $s[x]$, an expression containing one variable x, U-LAND generates a subproblem requiring synthesis of $s[x_1, x_2]$, a term that contains two other variables, x_1 and x_2 in place. More generally, we may generate many structurally similar subproblems that differ only in the use of certain identifiers

$$
\dfrac{\begin{matrix} w_1 \rightsquigarrow s[x_1, x_2]\ \mathsf{unsync} \\ \vdots \end{matrix} \qquad \begin{matrix} w_1 \rightsquigarrow s[x_3, x_4]\ \mathsf{unsync} \\ \vdots \end{matrix}}{w \rightsquigarrow s\ \mathsf{unsync}}
$$

To avoid this issue, the synchronized sequent calculus eliminates the stand-alone left rules for intersection. Instead, we define a projection operation ($r_1 \rhd r_2$) that extracts a left or right component of an intersection, and use that operation[4] whenever a hypothesis is drawn from the context. Projection from the context and then use of a hypothesis in rule R is equivalent to bottom-up use of a series of LAND rules followed by R. This projection is sound, but does introduce an incompleteness. Consider the S-LOR rule: when r_0 has

[3] Bool may be represented as $\mathsf{true} \vee \mathsf{false}$ in our formal setting.

[4] or subsumption, which is a generalization of projection, but which introduces more non-determinism when there is no bound on the supertype.

$$\boxed{\Gamma \vdash e : r\ \mathsf{nd}}$$

$$\text{ND-VAR}\ \frac{x : r \in \Gamma}{\Gamma \vdash x : r\ \mathsf{nd}} \qquad \text{ND-SAMPLE}\ \frac{c \in \mathcal{C}[\![r]\!]}{\Gamma \vdash c : r\ \mathsf{nd}} \qquad \text{ND-TRUE}\ \frac{}{\Gamma \vdash \mathsf{true} : \mathsf{true}\ \mathsf{nd}} \qquad \text{ND-FALSE}\ \frac{}{\Gamma \vdash \mathsf{false} : \mathsf{false}\ \mathsf{nd}}$$

$$\text{ND-BOT}\ \frac{\Gamma \vdash e' : \bot\ \mathsf{nd}}{\Gamma \vdash e : r\ \mathsf{nd}} \qquad \text{ND-SUB}\ \frac{\Gamma \vdash e : r_1\ \mathsf{nd} \qquad r_1 \leq r_2}{\Gamma \vdash e : r_2\ \mathsf{nd}}$$

$$\text{ND-ITE-TRUE}\ \frac{\Gamma \vdash e_0 : \mathsf{true}\ \mathsf{nd} \qquad \Gamma \vdash e_1 : r\ \mathsf{nd}}{\Gamma \vdash \mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : r\ \mathsf{nd}} \qquad \text{ND-ITE-FALSE}\ \frac{\Gamma \vdash e_0 : \mathsf{false}\ \mathsf{nd} \qquad \Gamma \vdash e_2 : r\ \mathsf{nd}}{\Gamma \vdash \mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : r\ \mathsf{nd}}$$

$$\text{ND-ANDI}\ \frac{\Gamma \vdash e : r_1\ \mathsf{nd} \qquad \Gamma \vdash e : r_2\ \mathsf{nd}}{\Gamma \vdash e : r_1 \wedge r_2\ \mathsf{nd}} \qquad \text{ND-ANDE1}\ \frac{\Gamma \vdash e : r_1 \wedge r_2\ \mathsf{nd}}{\Gamma \vdash e : r_1\ \mathsf{nd}}$$

$$\text{ND-ANDE2}\ \frac{\Gamma \vdash e : r_1 \wedge r_2\ \mathsf{nd}}{\Gamma \vdash e : r_2\ \mathsf{nd}} \qquad \text{ND-ORI1}\ \frac{\Gamma \vdash e : r_1\ \mathsf{nd}}{\Gamma \vdash e : r_1 \vee r_2\ \mathsf{nd}} \qquad \text{ND-ORI2}\ \frac{\Gamma \vdash e : r_2\ \mathsf{nd}}{\Gamma \vdash e : r_1 \vee r_2\ \mathsf{nd}}$$

$$\text{ND-ORE}\ \frac{\Gamma \vdash e' : r_1 \vee r_2\ \mathsf{nd} \qquad (x \notin \Gamma) \qquad \Gamma, x : r_1 \vdash e : r\ \mathsf{nd} \qquad \Gamma, x : r_2 \vdash e : r\ \mathsf{nd}}{\Gamma \vdash e[e'/x] : r\ \mathsf{nd}}$$

Figure 3: Selected Natural Deduction Typing Rules

$$\boxed{\langle \Gamma \vdash r \rangle \rightsquigarrow s\ \mathsf{unsync}}$$

$$\text{U-CTX}\ \frac{x : r' \in \Gamma \qquad r' \leq r \qquad r \leq \mathsf{true} \vee \mathsf{false}}{\langle \Gamma \vdash r \rangle \rightsquigarrow x\ \mathsf{unsync}} \qquad \text{U-TRUE}\ \frac{\mathsf{true} \leq r}{\langle \Gamma \vdash r \rangle \rightsquigarrow \mathsf{true}\ \mathsf{unsync}}$$

$$\text{U-FALSE}\ \frac{\mathsf{false} \leq r}{\langle \Gamma \vdash r \rangle \rightsquigarrow \mathsf{false}\ \mathsf{unsync}} \qquad \text{U-BOT}\ \frac{x : \bot \in \Gamma}{\langle \Gamma \vdash r \rangle \rightsquigarrow s\ \mathsf{unsync}}$$

$$\text{U-ITE-TRUE}\ \frac{x : \mathsf{true} \in \Gamma \qquad \langle \Gamma \vdash r \rangle \rightsquigarrow s_1\ \mathsf{unsync}}{\langle \Gamma \vdash r \rangle \rightsquigarrow \mathsf{if}\ x\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2\ \mathsf{unsync}} \qquad \text{U-ITE-FALSE}\ \frac{x : \mathsf{false} \in \Gamma \qquad \langle \Gamma \vdash r \rangle \rightsquigarrow s_2\ \mathsf{unsync}}{\langle \Gamma \vdash r \rangle \rightsquigarrow \mathsf{if}\ x\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2\ \mathsf{unsync}}$$

$$\text{U-RARROW}\ \frac{x \notin \Gamma \qquad \langle \Gamma, x : r_a \vdash r_b \rangle \rightsquigarrow s\ \mathsf{unsync}}{\langle \Gamma \vdash r_a \to r_b \rangle \rightsquigarrow \lambda x.s\ \mathsf{unsync}}$$

$$\text{U-LARROW}\ \frac{x_1 : r_1 \to r_2 \in \Gamma \qquad \langle \Gamma \vdash r_1 \rangle \rightsquigarrow s_1\ \mathsf{unsync} \qquad x_2 \notin \Gamma \qquad \langle \Gamma, x_2 : r_2 \vdash r \rangle \rightsquigarrow s_2\ \mathsf{unsync}}{\langle \Gamma \vdash r \rangle \rightsquigarrow \mathsf{let}\ x_2\ =\ x_1\ s_1\ \mathsf{in}\ s_2\ \mathsf{unsync}}$$

$$\text{U-RAND}\ \frac{\langle \Gamma \vdash r_1 \rangle \rightsquigarrow s\ \mathsf{unsync} \qquad \langle \Gamma \vdash r_2 \rangle \rightsquigarrow s\ \mathsf{unsync}}{\langle \Gamma \vdash r_1 \wedge r_2 \rangle \rightsquigarrow s\ \mathsf{unsync}}$$

$$\text{U-LAND}\ \frac{x : r_a \wedge r_b \in \Gamma \qquad x_1, x_2 \notin \Gamma \qquad \langle \Gamma, x_1 : r_a, x_2 : r_b \vdash r \rangle \rightsquigarrow s\ \mathsf{unsync}}{\langle \Gamma \vdash r \rangle \rightsquigarrow s[x/x_1][x/x_2]\ \mathsf{unsync}}$$

$$\text{U-ROR1}\ \frac{\langle \Gamma \vdash r_1 \rangle \rightsquigarrow s\ \mathsf{unsync}}{\langle \Gamma \vdash r_1 \vee r_2 \rangle \rightsquigarrow s\ \mathsf{unsync}} \qquad \text{U-ROR2}\ \frac{\langle \Gamma \vdash r_2 \rangle \rightsquigarrow s\ \mathsf{unsync}}{\langle \Gamma \vdash r_1 \vee r_2 \rangle \rightsquigarrow s\ \mathsf{unsync}}$$

$$\text{U-LOR}\ \frac{\langle \Gamma, x : r_1, \Gamma' \vdash r \rangle \rightsquigarrow s\ \mathsf{unsync} \qquad \langle \Gamma, x : r_2, \Gamma' \vdash r \rangle \rightsquigarrow s\ \mathsf{unsync}}{\langle \Gamma, x : r_1 \vee r_2, \Gamma' \vdash r \rangle \rightsquigarrow s\ \mathsf{unsync}}$$

$$\text{U-RPAIR}\ \frac{\langle \Gamma \vdash r_a \rangle \rightsquigarrow s_1\ \mathsf{unsync} \qquad \langle \Gamma \vdash r_b \rangle \rightsquigarrow s_2\ \mathsf{unsync}}{\langle \Gamma \vdash r_a \times r_b \rangle \rightsquigarrow \langle s_1, s_2 \rangle\ \mathsf{unsync}}$$

$$\text{U-LPAIR}\ \frac{x : r_a \times r_b \in \Gamma \qquad x_1, x_2 \notin \Gamma \qquad \langle \Gamma, x_1 : r_a, x_2 : r_b \vdash r \rangle \rightsquigarrow s\ \mathsf{unsync}}{\langle \Gamma \vdash r \rangle \rightsquigarrow \mathsf{let}\ \langle x_1, x_2 \rangle\ =\ x\ \mathsf{in}\ s\ \mathsf{unsync}}$$

$$\text{U-SAMPLE}\ \frac{c \in \mathcal{C}[\![r']\!] \qquad x \notin \Gamma \qquad \langle \Gamma, x : r' \vdash r \rangle \rightsquigarrow s\ \mathsf{unsync}}{\langle \Gamma \vdash r \rangle \rightsquigarrow \mathsf{let}\ x\ =\ c\ \mathsf{in}\ s\ \mathsf{unsync}}$$

Figure 4: Unsynchronized Sequent Calculus

the form $r_a \wedge (r_1 \vee r_2)$ the projection loses information about r_a. We conjecture that if the natural deduction system allowed distribution of conjunction over disjunction this restriction could be lifted.

Formally, the synchronized sequent calculus is defined in Figure 5. In this system, when viewed bottom up, the right rule for intersection and the left rule for union add new worlds/constraints to be satisfied. Intersections and unions can progress independently in different worlds, but processing connectives such as pairing, functions and booleans, whose introduction and elimination are marked in the expressions, must happen in sync. For example, S-RARROW introduces a new hypothesis to all worlds simultaneously.

The synchronized sequent calculus is sound with respect to the unsynchronized calculus. We carried out the proof in two stages, using an intermediate calculus that is unsynchronized but has the same kind of projections as appear in the synchronized calculus. The extended version [12] includes a paper proof, but we did significant (but incomplete) verification in Coq.[5]

[5] We did not formalize substitution and alpha-conversion mechanically, but merely admitted such properties where necessary unchecked.

Theorem 2 (Synchronized Sequent Calculus Soundness)**.**

- *If* $\langle \Gamma \vdash r \rangle \rightsquigarrow s\ \mathsf{sync}$ *then* $\langle \Gamma \vdash r \rangle \rightsquigarrow s\ \mathsf{unsync}$.

3.5 Step 4: Negation

To integrate negation, we augment the refinement language with an additional operator, $\mathsf{not}(r)$. Intuitively, this refinement denotes any value (with the same simple type) that is not described by r. In the context of synthesis from examples, this feature is most useful in an iterative workflow. Specifically, after the system synthesizes a function, the user executes it on some new data r_1 and receives an unexpected result r_2. The user can re-run the synthesis procedure with the additional refinement $r_1 \to \mathsf{not}(r_2)$. By replacing the user with a verification oracle, this process becomes CEGIS.

Rather than integrate negation into our synthesis procedure directly, we normalize it into other refinements before synthesis begins. In some cases, however, our refinement language is not powerful enough to fully capture the result of doing so. For instance, consider $\mathsf{not}(\mathsf{nat} \to \mathsf{true})$, where nat describes all natural numbers. Intuitively, we need a refinement that specifies that there exists some

$$\boxed{r_1 \rhd r_2} \qquad\qquad \boxed{\Gamma \vdash x : r \ \mathsf{prj}}$$

$$\textsc{Prj-Refl}\ \frac{}{r \rhd r} \qquad \textsc{Prj-Left}\ \frac{r_1 \rhd r}{r_1 \wedge r_2 \rhd r} \qquad \textsc{Prj-Right}\ \frac{r_2 \rhd r}{r_1 \wedge r_2 \rhd r} \qquad \textsc{Prj}\ \frac{x : r' \in \Gamma \qquad r' \rhd r}{\Gamma \vdash x : r \ \mathsf{prj}}$$

$$\boxed{\mathcal{W} \rightsquigarrow s \ \mathsf{sync}}$$

$$\textsc{S-Ctx}\ \frac{\forall \langle \Gamma \vdash r \rangle \in \mathcal{W} : (x : r' \in \Gamma \ \wedge\ r' \leq r \ \wedge\ r \leq \mathsf{true} \vee \mathsf{false})}{\mathcal{W} \rightsquigarrow x \ \mathsf{sync}}$$

$$\textsc{S-RArrow}\ \frac{x \notin \Gamma_1, \ldots, \Gamma_n \qquad \langle \Gamma_1, x : r_{a1} \vdash r_{b1} \rangle \cdots \langle \Gamma_n, x : r_{an} \vdash r_{bn} \rangle \rightsquigarrow s \ \mathsf{sync}}{\langle \Gamma_1 \vdash r_{a1} \to r_{b1} \rangle \cdots \langle \Gamma_n \vdash r_{an} \to r_{bn} \rangle \rightsquigarrow \lambda x.s \ \mathsf{sync}}$$

$$\textsc{S-LArrow}\ \frac{\langle \Gamma_1 \vdash r_{a1} \rangle \cdots \langle \Gamma_n \vdash r_{an} \rangle \rightsquigarrow s_1 \ \mathsf{sync} \qquad \begin{array}{c} \forall i \in n : \Gamma_i \vdash x_1 : r_{ai} \to r_{bi} \ \mathsf{prj} \\ x_2 \notin \Gamma_1, \ldots, \Gamma_n \end{array} \qquad \langle \Gamma_1, x_2 : r_{b1} \vdash r_1 \rangle \cdots \langle \Gamma_n, x_2 : r_{bn} \vdash r_n \rangle \rightsquigarrow s_2 \ \mathsf{sync}}{\langle \Gamma_1 \vdash r_1 \rangle \cdots \langle \Gamma_n \vdash r_n \rangle \rightsquigarrow \mathsf{let}\, x_2 \ = \ x_1 \ s_1 \, \mathsf{in}\, s_2 \ \mathsf{sync}}$$

$$\textsc{S-Bot}\ \frac{\Gamma \vdash x : \bot \ \mathsf{prj} \qquad \mathcal{W} \rightsquigarrow s \ \mathsf{sync}}{\mathcal{W}\langle \Gamma \vdash r \rangle \rightsquigarrow s \ \mathsf{sync}} \qquad \textsc{S-RAnd}\ \frac{\mathcal{W}\langle \Gamma \vdash r_1 \rangle \langle \Gamma \vdash r_2 \rangle \rightsquigarrow s \ \mathsf{sync}}{\mathcal{W}\langle \Gamma \vdash r_1 \wedge r_2 \rangle \rightsquigarrow s \ \mathsf{sync}} \qquad \textsc{S-ROr1}\ \frac{\mathcal{W}\langle \Gamma \vdash r_1 \rangle \rightsquigarrow s \ \mathsf{sync}}{\mathcal{W}\langle \Gamma \vdash r_1 \vee r_2 \rangle \rightsquigarrow s \ \mathsf{sync}} \qquad \textsc{S-ROr2}\ \frac{\mathcal{W}\langle \Gamma \vdash r_2 \rangle \rightsquigarrow s \ \mathsf{sync}}{\mathcal{W}\langle \Gamma \vdash r_1 \vee r_2 \rangle \rightsquigarrow s \ \mathsf{sync}}$$

$$\textsc{S-LOr}\ \frac{r_0 \rhd r_1 \vee r_2 \qquad \mathcal{W}\langle \Gamma, x{:}r_1, \Gamma' \vdash r \rangle \langle \Gamma, x{:}r_2, \Gamma' \vdash r \rangle \rightsquigarrow s \ \mathsf{sync}}{\mathcal{W}\langle \Gamma, x{:}r_0, \Gamma' \vdash r \rangle \rightsquigarrow s \ \mathsf{sync}} \qquad \textsc{S-True}\ \frac{\forall \langle \Gamma \vdash r \rangle \in \mathcal{W} : \mathsf{true} \leq r}{\mathcal{W} \rightsquigarrow \mathsf{true} \ \mathsf{sync}} \qquad \textsc{S-False}\ \frac{\forall \langle \Gamma \vdash r \rangle \in \mathcal{W} : \mathsf{false} \leq r}{\mathcal{W} \rightsquigarrow \mathsf{false} \ \mathsf{sync}}$$

$$\textsc{S-ITE}\ \frac{\forall \langle \Gamma \vdash r \rangle \in \mathcal{W}_1 : \Gamma \vdash x : \mathsf{true} \ \mathsf{prj} \qquad \forall \langle \Gamma \vdash r \rangle \in \mathcal{W}_2 : \Gamma \vdash x : \mathsf{false} \ \mathsf{prj} \qquad \mathcal{W}_1 \rightsquigarrow s_1 \ \mathsf{sync} \qquad \mathcal{W}_2 \rightsquigarrow s_2 \ \mathsf{sync}}{\mathcal{W}_1 \mathcal{W}_2 \rightsquigarrow \mathsf{if}\, x \, \mathsf{then}\, s_1 \, \mathsf{else}\, s_2 \ \mathsf{sync}}$$

$$\textsc{S-Sample}\ \frac{\forall i \in n : c \in \mathcal{C}[\![r'_i]\!] \qquad x \notin \Gamma_1, \ldots, \Gamma_n \qquad \langle \Gamma, x : r'_1 \vdash r_1 \rangle \cdots \langle \Gamma, x : r'_n \vdash r_n \rangle \rightsquigarrow s \ \mathsf{sync}}{\langle \Gamma_1 \vdash r_1 \rangle \cdots \langle \Gamma_n \vdash r_n \rangle \rightsquigarrow \mathsf{let}\, x \ = \ c \, \mathsf{in}\, s \ \mathsf{sync}} \qquad \textsc{S-RPair}\ \frac{\begin{array}{c} \langle \Gamma_1 \vdash r_{a1} \rangle \cdots \langle \Gamma_n \vdash r_{an} \rangle \rightsquigarrow s_1 \ \mathsf{sync} \\ \langle \Gamma_1 \vdash r_{b1} \rangle \cdots \langle \Gamma_n \vdash r_{bn} \rangle \rightsquigarrow s_2 \ \mathsf{sync} \end{array}}{\langle \Gamma_1 \vdash r_{a1} \times r_{b1} \rangle \cdots \langle \Gamma_n \vdash r_{an} \times r_{bn} \rangle \rightsquigarrow \langle s_1, s_2 \rangle \ \mathsf{sync}}$$

$$\textsc{S-LPair}\ \frac{\forall i \in n : \Gamma_i \vdash x : r_{ai} \times r_{bi} \ \mathsf{prj} \qquad x_1, x_2 \notin \Gamma_1, \ldots, \Gamma_n \qquad \langle \Gamma_1, x_1 : r_{a1}, x_2 : r_{b1} \vdash r_1 \rangle \cdots \langle \Gamma_n, x_1 : r_{an}, x_2 : r_{bn} \vdash r_n \rangle \rightsquigarrow s \ \mathsf{sync}}{\langle \Gamma_1 \vdash r_1 \rangle \cdots \langle \Gamma_n \vdash r_n \rangle \rightsquigarrow \mathsf{let}\, \langle x_1, x_2 \rangle \ = \ x \, \mathsf{in}\, s \ \mathsf{sync}}$$

Figure 5: Synchronized Sequent Calculus

$$\begin{array}{lrcl}
\text{(Refinements)} & r & ::= & \ldots \mid \mathsf{not}(u) \\
\text{(Negated Refinements)} & u & ::= & \mathsf{true} \mid \mathsf{false} \mid u \times u \ \mid t \to u \\
 & & \mid & \bot \mid u \wedge u \ \mid u \vee u \ \mid \mathsf{not}(u) \\
\text{(Singletons)} & t & ::= & \mathsf{true} \mid \mathsf{false} \mid t_1 \times t_2 \\
\text{(Types)} & \tau & ::= & \mathsf{Bool} \mid \tau_1 \times \tau_2 \mid \tau_1 \to \tau_2
\end{array}$$

Figure 6: Negation Syntax

natural number n that maps to false. Unfortunately, while we have finitary unions, we do not have infinitary unions (*i.e.,* existentials). This problem arises whenever we negate a function whose argument refinement has infinitely many inhabitants. Hence, we only permit negation of functions whose argument refinements are singletons. This choice covers the common case of interest, rejecting concrete mappings such as $\mathsf{true} \to \mathsf{false}$.

Formally, we present the extended grammar in Figure 6. Here, u ranges over refinements that may be negated and t over singletons. We also define the sub-grammar of types, which describe the "universe" with respect to which we negate a refinement.

We eliminate negation prior to synthesis using a normalization function ($\mathcal{N}$) presented in Figure 7. Normalization is a partial function mapping a refinement and a type to a refinement; it fails to produce an output in cases where the refinement and type are incompatible. The type argument is necessary to clarify the result of $\mathsf{not}(\bot)$, which would otherwise be ambiguous. On non-negated inputs, the function $\mathcal{N}$ recursively normalizes a refinement's constituents. On negated inputs, it applies logical identities to eliminate negation or push it onto smaller subterms. For example, $\mathsf{not}(\mathsf{true})$ becomes false. Likewise, we use DeMorgan's laws to negate conjunction and disjunction. $\mathsf{not}(\bot)$ becomes the type argument and double negation cancels out.

$$\begin{array}{lcl}
\mathcal{N}(\mathsf{true}, \mathsf{Bool}) & = & \mathsf{true} \\
\mathcal{N}(\mathsf{false}, \mathsf{Bool}) & = & \mathsf{false} \\
\mathcal{N}(\bot, \tau) & = & \bot \\
\mathcal{N}(u_1 \wedge u_2, \tau) & = & \mathcal{N}(u_1, \tau) \wedge \mathcal{N}(u_2, \tau) \\
\mathcal{N}(u_1 \vee u_2, \tau) & = & \mathcal{N}(u_1, \tau) \vee \mathcal{N}(u_2, \tau) \\
\mathcal{N}(u_1 \times u_2, \tau_1 \times \tau_2) & = & \mathcal{N}(u_1, \tau_1) \times \mathcal{N}(u_2, \tau_2) \\
\mathcal{N}(u_1 \to u_2, \tau_1 \to \tau_2) & = & \mathcal{N}(u_1, \tau_1) \to \mathcal{N}(u_2, \tau_2) \\
\mathcal{N}(\mathsf{not}(\mathsf{true}), \mathsf{Bool}) & = & \mathsf{false} \\
\mathcal{N}(\mathsf{not}(\mathsf{false}), \mathsf{Bool}) & = & \mathsf{true} \\
\mathcal{N}(\mathsf{not}(\bot), \tau) & = & \tau \\
\mathcal{N}(\mathsf{not}(u_1 \wedge u_2), \tau) & = & \mathcal{N}(\mathsf{not}(u_1), \tau) \vee \mathcal{N}(\mathsf{not}(u_2), \tau) \\
\mathcal{N}(\mathsf{not}(u_1 \vee u_2), \tau) & = & \mathcal{N}(\mathsf{not}(u_1), \tau) \wedge \mathcal{N}(\mathsf{not}(u_2), \tau) \\
\mathcal{N}(\mathsf{not}(\mathsf{not}(u)), \tau) & = & \mathcal{N}(u, \tau) \\
\mathcal{N}(\mathsf{not}(u_1 \times u_2), \tau_1 \times \tau_2) & = & \mathcal{N}(\mathsf{not}(u_1), \tau_1) \times \tau_2 \ \vee \\
 & & \tau_1 \times \mathcal{N}(\mathsf{not}(u_2), \tau_2) \\
\mathcal{N}(\mathsf{not}(t \to u), \tau_1 \to \tau_2) & = & t \to \mathcal{N}(\mathsf{not}(u), \tau_2)
\end{array}$$

Figure 7: Negation Normalization $\mathcal{N} : u \times \tau \to u$

$$\begin{array}{lrl}
\text{(Types)} & \tau ::= & \alpha \mid \mathcal{B} \mid \mathsf{unit} \\
 & \mid & \tau_1 \rightarrow \tau_2 \ \mid \tau_1 \times \ldots \times \tau_m \\
\text{(Refinements)} & r ::= & \mathcal{B}^{\top} \mid \mathcal{B}^{\bot} \mid C\ r \ \mid \mathsf{unit}^{\top} \mid \mathsf{unit}^{\bot} \\
 & \mid & r_1 \times \ldots \times r_m \ \mid r_1 \rightarrow r_2 \\
 & \mid & r_1 \wedge r_2 \ \mid r_1 \vee r_2 \ \mid \mathsf{not}(r) \ \mid \alpha \mid \alpha_n \\
\text{(Expressions)} & e ::= & x \mid () \mid C\ e \\
 & \mid & \mathsf{match}\ e\ \mathsf{with}\ \overline{C_i\ x \rightarrow e_i}^{i \in m} \\
 & \mid & \langle e_1, \ldots, e_m \rangle \ \mid \pi_i\ e \\
 & \mid & \mathsf{fix}\ f\ x.e \mid e_1\ e_2
\end{array}$$

Figure 8: Surface Refinement and Expression Syntax

4. Implementation

The previous section defines the parameters for a synthesis procedure for a pure subset of ML. Building on this foundation, we have implemented a refinement-based synthesis engine from the ground up in approximately 4,000 lines of F#. This synthesizer generates polymorphic, structurally inductive, higher-order functional programs with algebraic datatypes. It uses the Curry-Howard isomorphism to treat an input refinement as a theorem to be proved, performing focused proof search over the sequent calculus to find a program that satisfies the specification. In the following sections, we discuss key features of our prototype and experiments assessing its performance. Our evaluation benchmarks are a library of synthesis problems similar to exercises assigned in an introductory functional programming class. To the extent possible, we use tests from Osera and Zdancewic [22] (with "examples" converted into refinements) to provide a clear basis for comparison between systems. Where necessary, we have added benchmarks to pinpoint relevant qualities of our implementation.

4.1 Language Extensions

Our implementation extends the formal language described in Section 3 significantly (see Figure 8). In particular, it includes recursive ML-style data types, such as lists and trees, as well as structurally inductive functions, and n-ary tuples. It also handles ML-style prenex polymorphism exclusively on the right, meaning users can specify polymorphic refinements as goals but that we cannot yet sample polymorphic libraries.

Each data type ($\mathcal{B}$) has a "top" refinement ($\mathcal{B}^{\top}$) describing elements of the data type as well as a "bottom" refinement ($\mathcal{B}^{\bot}$), describing none. A singleton refinement is created using a datatype constructor (C). For example, suppose the type of natural numbers is nat. We write the top refinement for natural numbers as $\mathsf{nat}^{\top}$ and the bottom refinement as $\mathsf{nat}^{\bot}$. Other refinements for natural numbers are created using the constructors for successor (S) and zero (Z). We can build singleton refinements (S (S Z)—the number 2) or non-singletons (S (S $\mathsf{nat}^{\top}$)—all numbers greater than or equal to 2) from these components. The implementation refinement language also includes intersections, unions, and negation.

To describe the behavior of a polymorphic function, we provide abstract refinements (α_n) representing distinct inhabitants of a polymorphic type (α), which itself is a valid refinement that captures *all* of its inhabitants (just as Z represents one inhabitant of nat). As is standard in ML, such refinements are implicitly universally quantified at the top-level. For instance, to specify a polymorphic map over lists, we write the following.

$$(\alpha_1 \rightarrow \beta_1) \rightarrow ([\,] \rightarrow [\,] \wedge [\alpha_1] \rightarrow [\beta_1] \wedge [\alpha_1; \alpha_1] \rightarrow [\beta_1; \beta_1])$$

Abstract refinements relay a great deal of information—the above specification is sufficient to synthesize list map.

The syntax of expressions is generalized to include constructors, pattern matching (as opposed to simple conditionals), n-ary tuples and recursive functions. We reduce the let expressions in the sequent-normal forms and return synthesized expressions in eta-long, beta-normal-form to the user (Figure 8 elides these details).

We require that a recursive function be structurally decreasing on its argument with a syntactic check similar to those found in Coq and Agda, ensuring that we never generate a non-terminating program. To enforce this rule, we permit recursive calls only on variables that result from pattern-matching on a function's argument. To generate a recursive function, we demand that refinements be *trace complete*. That is, when [2; 1] is an argument refinement, the synthesizer also needs refinements involving the arguments [1] and [ ] to clarify the results of structurally recursive calls. This limitation, which is also present in MYTH [22], serves as a the example-based equivalent of a loop invariant for recursive function specifications.

$$\begin{array}{lrl}
\text{(Normalized} & n ::= & \mathcal{B}^{\top} \mid \vee(\overline{C_i\ n_i}^{i \in m}) \ \mid \alpha \mid \vee(\overline{\alpha_i}^{i \in m}) \\
\text{Refinements)} & \mid & \mathsf{unit}^{\top} \mid \mathsf{unit}^{\bot} \\
 & \mid & \vee(\overline{n_{(i,1)} \times \ldots \times n_{(i,k)}}^{i \in m}) \\
 & \mid & \wedge(\overline{\vee(\overline{n_{(i,j,1)} \rightarrow n_{(i,j,2)}}^{j \in n_i})}^{i \in m})
\end{array}$$

Figure 9: Normalized Refinement Syntax

4.2 Refinement Preprocessing

The flexibility of the refinement language is a key benefit for users but a significant challenge for the implementer. By translating the surface syntax into a more restrictive form, we can streamline the internals of the synthesizer and improve performance using techniques like hash-consing. There are two stages of the translation:

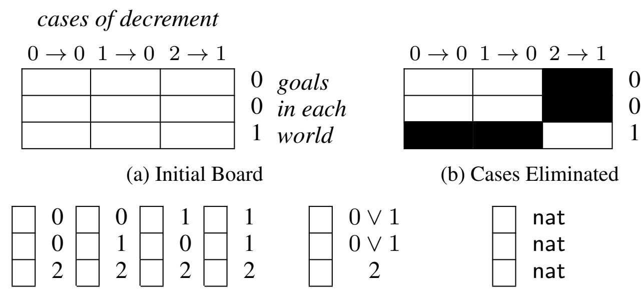
1. *Negation normalization.* Negation is eliminated according to the procedure in Section 3.5 (with additional rules for the refinement language in Figure 8). Since this step has already been treated in depth, we do not discuss it further here.
2. *Refinement normalization.* Nested conjunction and disjunction are collapsed and then pushed through other refinements where possible, ensuring that they only appear in specific places.

The internal refinement syntax processed by the synthesizer appears in Figure 9. Specifically, we permit unions of constructor refinements (provided no constructor is repeated), unions of tuples, arrows in conjunctive normal form (CNF), and unions of abstract refinements. We achieve this normal form by using logical identities and observing that intersection and union can be pushed through some of the other refinements.

For instance, at base type, we merge intersections and unions of common constructors. Hence, a disjunct of constructors such as (S Z) $\vee$ (S S Z) is normalized to S (Z $\vee$ S Z). Likewise, (S $\mathsf{nat}^{\top}$)$\wedge$ (S Z) is normalized to S Z. If we encounter an inconsistent refinement such as S Z $\wedge$ Z, the normalization procedure simply produces the appropriate bottom refinement. If bottom appears in goal position, the system signals the user that specification is inconsistent. The same logic guides the normal form for abstract refinements.

We can apply similar kinds of reasoning to other refinements. For instance, since $(r_{1a} \times r_{1b}) \wedge (r_{2a} \times r_{2b})$ is equivalent to $(r_{1a} \wedge r_{2a}) \times (r_{1b} \wedge r_{2b})$, we can push intersection (but not union) through tuples, leaving its normal form as a union of tuples. In addition, we can push intersection through union (and vice versa) by distributing. Neither operator can be pushed through arrows, so we put arrows in CNF by pushing union through intersection.[6]

[6] We chose CNF to ensure that, during synthesis, we process conjunction, which is both left and right invertible, before disjunction, which is only left invertible. See Section 4.3 for a full discussion of invertibility.



(a) Initial Board (b) Cases Eliminated

(c) Refinement Directed Search (d) Disjunction Enumeration (e) Type Enumeration

Figure 10: Argument Sudoku for Decrement

4.3 Focused Proof Search

In order to search for inhabitants of a refinement, we iteratively increase the depth of our derivation in a breadth-first manner until we arrive at a solution, ensuring that we find the smallest possible term (by depth) that satisfies our goal. We also periodically deepen the maximum allowed match statement depth and the scrutinee size; both metrics have a significant influence our search procedure's branching factor and must be carefully managed to avoid an early explosion in search space size.

To improve the performance of our theorem-prover, we exploit the well-known fact that certain logical rules are *invertible*—applying the rule without back-tracking does not cause us to lose completeness. For example, when our goal refinement is an arrow, we can always apply the corresponding right rule and immediately generate a function without loss of completeness. The right rules for pairs and intersections are also invertible, as are the left rules for unions, pairs and intersections. In general, whenever we have the opportunity to apply an invertible rule, we do so eagerly, reducing the number of ways of interleaving synthesis rules and wringing nondeterminism out of the synthesis process. In cases where no invertible rules are applicable (*i.e.*, disjunction and base type on the right) we branch, simultaneously trying all rules that might apply (*i.e.*, pattern matching, using terms from the context, etc.).

Focusing [2] is a generalization of these invertibility principles that makes it possible to chain together a series of non-invertible rules without losing completeness. For instance, even though arrow is not invertible on the left, we can focus on a function $f : r_1 \rightarrow r_2 \rightarrow r_3$ in the context and apply several instances of the left arrow rule in sequence—we do not have to backtrack at every individual application. When we reach an arrow in the context, we always choose to continue focusing rather than immediately search for possible arguments. Where arguments will eventually appear, we instead leave holes to be filled in later: $\mathsf{let}\ f_2 = f\ (\blacksquare_1 : r_1)\ \mathsf{in}\ \mathsf{let}\ f_3 = f_2\ (\blacksquare_2 : r_2)\ \mathsf{in}\ f_3 : r_3$. We handle polymorphism in a similar manner, eagerly focusing on a prenex-quanitifed type on the right and instantiating all type parameters at the very beginning of the synthesis process.

4.4 Argument Selection

The focused proof search process, which comprises the core of our synthesis algorithm, leaves unspecified the method for filling in arguments of functions in the context. At this point in the proof search process, the sequent calculus becomes highly nondeterministic, supplying little guidance for how to pick arguments efficiently. The performance of the system as a whole hinges on the algorithm for selecting arguments, so we discuss this process in detail, weighing the benefits and tradeoffs of five different strategies.

4.4.1 Argument "Sudoku."

Suppose we are trying to find arguments for the unary `decrement` function, which has the refinement $0 \rightarrow 0 \ \wedge \ 1 \rightarrow 0 \ \wedge \ 2 \rightarrow 1$. We can imagine the process of doing so as a game not unlike sudoku (Figure 10a). Each column represents a different "case" of `decrement` (*i.e.*, $1 \rightarrow 0$). [7] Each row represents a different world, where the refinements listed along the right edge of the box are the goal refinements that we need to satisfy. The aim of argument-selection sudoku is simple: synthesize an expression that typechecks against the argument refinement of one case in every row such that the corresponding output refinements are subtypes the goal refinements. An obvious first step is to eliminate all cases that are not subtypes of the goal (Figure 10b), leaving only the task of searching a now-smaller space for arguments.[8]

4.4.2 Strategy 0: Naive, Refinement-Directed Search

The proof theory from Section 3 suggests that we should create synthesis subproblems for every combination of cases that might produce the goal. In sudoku terms, we pick one box from each row and search for an argument that satisfies those constraints. Since we look for terms by depth in a breadth-first manner, we must examine all possible combinations of boxes simultaneously. Figure 10c shows all of the synthesis subproblems we need to create for the list `decrement` example in Figure10b. For instance, the first column in Figure 10c represents the choice of box 1 from row 1 (demanding we find an expression refined by 0), box 1 from row 2 (demanding we find another expression refined by 0), and box 3 from row 3 (demanding we find an expression refined by 2).The benefit of this approach is that we never have to check our answer: assuming that the synthesis algorithm is sound, any argument we find is guaranteed to lead to an application that satisfies our goal.

In the example of the `decrement` function, this algorithm seems manageable, producing four subproblems, but the number of combinations grows exponentially on functions with more cases and multiple arguments. This problem is particularly severe during the unconstrained search for match statement scrutinees. In practice, this strategy could generate list `length` but was unable to synthesize even the unary `sum` function.

4.4.3 Enumerative Search

As an alternative approach, instead of picking boxes first and then searching for candidates, we might find candidates first and then check to see whether they satisfy a box from each row.[9] There are several benefits to this enumerative approach:

1. Enumeration strategies do not inherently suffer from the same combinatorial explosion as does refinement-directed search (though some still do).
2. We can process multiple arguments effectively by searching for one argument at a time. Any first argument we choose will likely fail to satisfy many cases, which we can immediately cross off to narrow down the search space for the next argument. Of course, the first candidate we generate may not lead to a useful application if, for instance, it creates an impossibly difficult

[7] In general, the cases of a function need not be the same in every world, but assuming so simplifies this particular example. When synthesizing a recursive function, the same refinement is placed in the context of every world, so this situation does arise quite often in practice.

[8] When searching for match statement scrutinees, we must amend the game slightly since there are no goal refinements. Instead, we may attempt to pattern match on the result of any valid application and cannot eliminate any cases initially.

[9] To check that a candidate argument conforms to a refinement, we simply typecheck it. We cache all typechecking queries to avoid repeating work.

benchmark	**libraries**	**strat. 1**	**strat. 2**	**strat. 3**	**strat. 4**	**strat. 1**	**strat. 2**	**strat. 3**	**strat. 1**	**strat. 2**	**strat. 3**	**strat. 4**
		synthesis time s				*synthesis time %*			*enumeration cache hit %*			
bool-band		0.16	0.16	0.16	0.16	100.6	100.6	100.0	—	—	—	—
bool-bor		0.16	0.16	0.16	0.16	100.6	100.6	101.9	—	—	—	—
bool-impl		0.16	0.16	0.16	0.16	100.0	101.3	100.0	—	—	—	—
bool-neg		0.16	0.16	0.16	0.16	100.0	100.0	100.6	—	—	—	—
bool-xor		0.16	0.16	0.17	0.16	99.4	99.4	102.5	—	—	—	—
list-append		0.27	0.26	0.27	0.23	114.7	114.3	116.0	0.0	0.0	0.0	9.6
list-concat	*append*	0.29	0.29	0.29	0.28	102.1	100.7	103.2	83.1	83.1	83.1	84.0
list-dict-contains		0.64	0.62	0.66	0.27	236.4	229.4	244.1	3.4	4.0	3.4	16.4
list-dict-find		0.41	0.37	0.42	0.25	167.2	150.2	169.2	3.1	4.2	3.1	9.6
list-dict-replace		1.11	1.01	1.15	0.33	340.7	308.9	351.4	2.7	3.5	2.7	12.2
list-drop		0.46	0.46	0.48	0.26	177.3	176.2	185.8	0.0	0.0	0.0	12.3
list-even-parity		0.30	0.29	0.32	0.26	114.7	113.6	122.5	0.0	0.0	0.0	26.9
list-filter		0.79	0.53	0.89	0.29	274.6	184.7	310.8	4.7	11.3	4.7	27.2
list-filter-3		timeout	0.54	0.91	0.30	timeout	182.8	307.4	5.9	12.1	6.1	25.3
list-fst		0.22	0.22	0.22	0.22	100.5	101.9	102.3	0.0	0.0	0.0	11.5
list-hd		0.17	0.17	0.18	0.17	100.0	100.0	101.1	—	—	—	—
list-inc		0.22	0.23	0.22	0.22	100.4	100.9	99.1	28.0	28.0	28.0	28.0
list-last		0.23	0.23	0.23	0.22	104.9	102.7	104.9	0.0	0.0	0.0	12.9
list-length		0.22	0.21	0.22	0.22	100.5	99.1	100.9	0.0	0.0	0.0	13.6
list-map		0.31	0.29	0.32	0.24	128.4	118.9	132.1	5.1	8.2	5.1	19.0
list-map-2		timeout	0.30	0.34	0.25	timeout	120.2	136.5	0.1	9.3	6.1	17.9
list-map-3		timeout	0.31	0.38	0.26	timeout	121.4	146.3	5.0	10.5	7.5	19.8
list-map-4		timeout	0.32	0.40	0.25	timeout	126.5	157.7	4.8	10.2	7.2	19.8
list-nth		0.45	0.46	0.48	0.26	176.1	180.4	186.7	0.0	0.0	0.0	14.8
list-rev	*append*	3.95	3.85	4.19	3.87	102.0	99.4	108.2	91.9	91.9	91.9	92.6
list-rev	*fold*	0.32	0.32	0.32	0.32	99.7	100.3	99.7	86.9	86.9	86.9	86.9
list-rev	*snoc*	0.30	0.30	0.30	0.30	101.3	100.3	102.0	82.3	82.3	82.3	83.2
list-rev-tailcall		0.34	0.34	0.35	0.25	135.9	134.7	138.2	8.0	8.0	8.0	21.2
list-snoc		0.28	0.29	0.29	0.24	119.8	121.1	122.8	0.0	0.0	0.0	11.9
list-insertion-sort	*insert*	0.32	0.32	0.33	0.32	100.6	100.3	102.8	82.0	82.0	82.0	83.3
list-sorted-insert	*compare*	2.11	2.09	2.19	1.02	207.7	204.8	214.7	8.7	8.7	8.7	38.0
list-stutter		0.22	0.22	0.22	0.21	101.9	102.3	103.3	0.0	0.0	0.0	12.1
list-sum	*add, fold*	0.38	0.37	0.37	0.38	100.3	99.5	99.5	93.2	93.2	93.2	93.2
list-take		27.36	27.49	0.81	0.30	9274.9	9318.6	272.9	0.0	0.0	0.0	14.4
list-tl		0.18	0.18	0.18	0.18	99.4	100.0	99.4	—	—	—	—
list-tuple-sum	*add*	0.37	0.37	0.38	0.36	102.8	101.9	105.2	85.5	85.5	85.5	88.4
list-tuple-swap		0.22	0.22	0.22	0.22	100.9	101.4	101.4	0.0	0.0	0.0	11.1
list-unzip		0.23	0.23	0.23	0.22	103.2	104.1	105.9	12.8	12.8	12.8	25.4
nat-dec		0.16	0.16	0.16	0.16	100.6	100.6	100.0	—	—	—	—
nat-div2		0.22	0.21	0.22	0.21	104.9	103.9	105.4	0.0	0.0	0.0	15.1
nat-iseven		0.21	0.21	0.21	0.20	103.5	103.0	104.0	0.0	0.0	0.0	14.6
nat-max		0.21	0.21	0.21	0.21	100.5	100.5	100.0	19.3	19.3	19.3	19.3
nat-sum		0.24	0.24	0.24	0.21	116.0	115.5	116.5	0.0	0.0	0.0	9.1
tree-binsert	*compare*	4.55	4.80	4.73	1.81	251.4	265.2	261.4	8.0	8.0	8.0	37.3
tree-collect-leaves	*append*	0.55	0.56	0.55	0.53	104.2	105.9	105.3	93.8	93.8	93.8	94.6
tree-count-leaves	*sum*	0.38	0.38	0.38	0.37	102.2	102.2	102.7	90.0	90.0	90.0	90.4
tree-count-nodes	*sum*	0.45	0.46	0.46	0.45	99.1	102.0	102.0	91.1	91.1	91.1	92.3
tree-inorder	*append*	0.53	0.54	0.54	0.53	99.1	101.5	102.3	93.8	93.8	93.8	94.6
tree-map		0.31	0.28	0.32	0.24	128.8	115.6	132.5	0.1	2.9	0.1	14.6
tree-postorder	*append*	14.32	14.20	15.47	14.53	98.5	97.7	106.5	96.8	96.8	96.8	97.1
tree-preorder	*append*	0.46	0.45	0.46	0.44	105.5	104.1	105.7	92.8	92.8	92.8	93.7

Figure 11: Performance of the four enumerative argument search strategies on all benchmarks. Tests timed out if no solution was found after 100 seconds. The left section contains synthesis times in seconds. The center section contains relative synthesis times as percentages of the synthesis time using strategy 4. The right section contains enumeration cache hit percentages. Missing entries signify cases where no enumeration took place during the synthesis process. Entries in the *libraries* column were provided the listed libraries during synthesis. Benchmarks were measured on an Amazon EC2 m4.large instance with 8GB of RAM using mono.

search space for the second argument. As such, we must continue to enumerate even after we find one candidate.

3. Enumeration queries are readily cacheable. We save the results of every enumeration query at a particular depth with goal refinement r and context Γ for each world, ensuring that we never do the same enumeration work twice.

The key choice when designing an enumerative search strategy is that of the *size* and *shape* of the space over which to search. Although a notion of search space size is standard, shape is less so. Refinements tend to produce search spaces of irregular shapes, such as $1 \vee 2 \vee 3$, that are less likely to entirely coincide with one another than are types such as `nat`. In the sections that follow, we describe and evaluate four variants of enumerative search that trade off between these two metrics. Figure 11 displays the synthesis times and enumeration cache hit percentages of each of these strategies on our suite of benchmarks.

Strategy 1: Disjunction-based enumeration. When enumerating arguments, we can exploit all of the refinement information available to constrain our search space. To do so, we observe that intersection of function cases on the left corresponds to disjunction of arguments on the right. That is, in the first row of the sudoku game, our search space is $0 \vee 1$, the disjunction of the argument refinements of every case in consideration. We can apply this disjunction rule to every world, producing the smallest search space attainable with the refinement information available (Figure 10d).

Unfortunately, the shape of this search space makes it extremely difficult to traverse. We cannot push disjunction through tuple and arrow refinements, forcing our enumeration process to branch and consider every combination of cases. This process degenerates to naive, refinement-directed search and experiences the same combinatorial explosion of subproblems. We see evidence of this phenomenon in Figure 11—many of the higher-order tests timed out. We attempted to synthesize four variants of list `map` and two of list `filter`. Benchmarks `list-map` and `list-filter` did not require the synthesizer to find a higher-order argument and completed quickly. Benchmarks `list-map-n` and `list-filter-n` forced the synthesizer to search for a higher-order argument described by a refinement with n cases. The disjunction-based enumeration approach timed out when n was as small as 2.

Strategy 2: Hybrid enumeration. Still, the disjunction-based enumeration strategy is not entirely without merit. We can push disjunction through base type refinements, skirting the case explosion at higher types. We tested a hybrid approach, which uses the disjunction-based strategy at base type and simply enumerates all well-typed terms at higher types. This approach performed better, but struggled in some cases where constructors stored tuples (*i.e.,* Cons in lists). Disjunction pushed through these constructors would arrive at the tuples inside, still creating some of the difficult-to-traverse disjunction that this strategy sought to avoid. The `list-take` benchmark in particular continued to see poor performance as a product of this limitation.

Strategy 3: Unsound disjunction-based enumeration. We can eliminate disjunction completely by pushing it through tuples and arrows in an unsound manner. For example, we might convert the refinement $(1 \times 2) \vee (3 \times 4)$ into $(1 \vee 3) \times (2 \vee 4)$. Although the second refinement includes several inhabitants that were not described by the first, this search space is still far smaller than that of $\mathsf{nat} \times \mathsf{nat}$. The performance of this approach was consistent across all benchmarks, avoiding the pathological cases of the other disjunction-based approaches. For example, `list-take`, which did not benefit from hybrid enumeration, performed dramatically better once disjunction could be pushed through the tuples of its lists.

Strategy 4: Type-based enumeration. As a seemingly naive final attempt, we enumerated all expressions at the argument type (Figure 10e). This is the strategy that Osera and Zdancewic, who were unable to invert function examples and had no choice but to guess all well-typed arguments, pursued [22]. This search space has the largest possible size but the most uniform shape: we search the same space for any two arguments of the same type. Surprisingly, ignoring all refinement information and pursuing this strategy performed convincingly better than any refinement-based approach.

Search space shape and cache performance. The results of the previous section might seem surprising: why does enumerating over the massive search space of a type perform better than doing so over a space constrained by refinements? As we have already discussed, part of our answer is that some spaces are easier to traverse than others. The remainder is that the irregular shapes of the refinement-constrained search spaces make their results harder to cache.

As noted previously, we cache the results of every enumeration query. Enumerating over the language of types has a larger granularity that improves the chance we will make the same query twice and get a cache hit. In Figure 11, we confirm that every refinement-based enumeration strategy sees a lower cache-hit percentage in line with the finer granularity of its specification language. It is far less likely that two refinement queries will be identical: the refinement language is far more flexible, permitting a wider range of search space shapes. Furthermore, refinement-based enumeration queries must be world-aware, taking into account varying refinement information across multiple worlds. In contrast, type information is shared from world to world, meaning that queries differing only between worlds look identical to the cache. Cache hit percentages did not vary greatly amongst the refinement-based strategies, indicating minor differences in the structure of refinements did little to improve cache performance. We conclude that, to fully explain the performance of different enumeration strategies, and, thereby, the performance of the system of a whole, we need to consider the shapes, not just the sizes, of their search spaces.

4.5 Key optimizations.

Caching and hash-consing. The synthesizer spends most of its time on the repetitive tasks of term-enumeration, evaluation, and typechecking. We cache these queries so that the same work is never done twice. To use theses caches, however, we need to be able to hash deeply-nested refinements, expressions, and types. In early testing, the bottleneck of the synthesis process was refinement hashing and equality checking. By hash-consing these datastructures, we memoize much of this work.

Eta-long sidestepping. The sequent calculus naturally generates expressions in eta-long, beta-normal form. This property, which cuts down the search space size by eliminating some equivalent programs, sometimes leads us to synthesize programs that are larger than necessary. For example, it is far easier to enumerate to the depth necessary to generate the function type variable f than the expression $\lambda x. f\ x$, yet the sequent calculus will always lead to the latter. During argument-selection, we opt for the shorter representation where possible, reducing the number of enumeration steps necessary to produce tuple and function variables.

4.6 Library Sampling

The synthesis algorithm as described suffices for self-contained problems expressed as refinements, but we also aim to import external library code provided in the form of expressions. The sampling rule for the natural deduction system, which serves this purpose in the theory, is extraordinarily powerful: it states that we can extract any refinement that describes a library. As discussed in Section 3.2, it is impossible to implement this rule in its full generality. Instead, we make use of a library's type information and its behavior when evaluated. This restriction means that we can generate the refinements $\mathsf{nat}^\top \rightarrow \mathsf{nat}^\top$ and $\mathsf{S}\ \mathsf{Z} \rightarrow \mathsf{Z}$ for the `decrement` function but never $\mathsf{S}\ \mathsf{S}\ \mathsf{nat}^\top \rightarrow \mathsf{S}\ \mathsf{nat}^\top$. We have implemented two sampling strategies which balance precomputation with refinement size. Neither strategy is yet capable of sampling polymorphic libraries which, as the equivalent of universal quantification on the left, require sampling on both types and terms.

Ahead-of-time sampling. In ahead-of-time sampling, we produce massive input-output tables for each library by generating arguments, evaluating, and recursively sampling the arguments. This strategy allows us to share work between many synthesis problems by importing already-sampled libraries on demand. Generating these tables is tremendously expensive, and we failed to build tables large enough to run many of our library-centric tests. These tables are cumbersome to manage during synthesis: every candidate argument must be typechecked against a vast number of cases. This strategy therefore performs especially poorly when combined with disjunction-based argument selection strategies.

Just-in-time sampling. In practice, our synthesis procedure needs a few specific samples rather than a comprehensive list of every case. Just-in-time sampling samples on demand during the argument-selection phase of the synthesis process, extracting only the information we need for a particular synthesis problem. When we generate a candidate argument, we evaluate the library on the argument to produce an output refinement.

The challenge in doing so is that arguments may contain subexpressions that are described only in terms of refinements rather than expressions, making standard evaluation impossible. For example, we might attempt to evaluate `length` on the variable `x` refined by $\mathsf{Cons}(\mathsf{S\ Z} \times \mathsf{list}^{\top})$, which provides too little information to construct an output refinement. Our procedure therefore makes a best effort to evaluate refinements as far as they correspond to values, extracting tuples and constructors and treating arrow refinements as tables from inputs to outputs. When evaluation cannot continue, the candidate argument is rejected. This process is no more restrictive than ahead-of-time sampling, which can only generate tables containing singleton refinements. Refinement evaluation automatically fails when faced with refinements like $\mathtt{nat}^{\top}$, which can be unfolded infinitely, potentially preventing evaluation from terminating.

Just-in-time sampling amounts to another instance of enumeration like that employed for argument selection. It most closely resembles Strategy 4 from Section 4.4, in which we generate and check (this time by evaluation) all well-typed arguments in increasing order of size. As in Section 4.4, doing so is the most expensive part of the synthesis process. For an illustrative example, consider the performance of the `tree preorder`, `inorder`, and `postorder` benchmarks in Figure 11, which traverse a tree and return the values stored in each node in a list ordered as the names suggest. These tests are provided with the `append` library, which allow the functions to appropriately merge lists produced from recursive calls to left and right subtrees. The `preorder` test finishes most quickly, since it requires a single append of the results of the recursive calls:

$$\mathsf{Cons}(\ \mathsf{this\_value}\ ,\ \ \mathsf{append}\ \ \mathsf{recur\_left}\ \ \ \mathsf{recur\_right}\ )$$

The `inorder` test requires slightly longer, since it must apply `append` to a larger second argument:

$$\mathsf{append}\ \ \mathsf{recur\_left}\ \ \ (\mathsf{Cons}(\ \mathsf{this\_value}\ ,\ \ \mathsf{recur\_right}\ ))$$

Where `preorder` and `inorder` still finish in approximately half a second each, `postorder` takes more than an order of magnitude more time to complete. It must make a second, nested call to `append`, requiring two separate sampling procedures and a bigger argument for the outer append:

$$\mathsf{append}\ \ \mathsf{recur\_left}\ \ \ (\mathsf{append}\ \ \mathsf{recur\_right}\ \ \ [\ \mathsf{this\_value}\ ])$$

These subtle changes in the ordering and number of library calls demonstrate the extent to which the enumeration underlying sampling dominates the performance of the synthesizer.

4.7 Practical Impact of Richer Specifications

For the user, the key dividend of this paper is a richer specification language with union, negation, and polymorphism. To evaluate the practical impact of this syntax, we updated a subset of our benchmarks to make aggressive use of these new constructs. Our benchmark suite, drawn primarily from Osera and Zdancewic [22], is specified with the restrictive syntax of its origin: singletons, arrows, and intersection. In Figure 12, we assess the extent to which our wider vocabulary enables us to describe the same programs with smaller specifications. Since refinements lack a notion of an "example," we measure our results in refinement AST nodes. On a subset of programs updated with union and negation, reductions ranged from marginal to nearly 80%, with drops of 10-20% most common. Several noteworthy features of programs that made these reductions possible were detailed with examples in Section 2.3.

benchmark	**orig.**	**disj.**	**poly.**	**disj.**	**poly.**
	refn ast nodes			*refn %*	
bool-band	33	23	—	69.7	—
bool-bor	33	23	—	69.7	—
bool-impl	33	24	—	72.7	—
list-append	106	103	88	97.2	83.0
list-concat	113	107	—	94.7	—
list-dict-contains	82	73	63	89.0	76.8
list-dict-find	102	100	63	98.0	61.8
list-dict-replace	222	207	110	93.2	49.5
list-drop	146	73	59	50.0	40.4
list-filter	189	153	66	81.0	34.9
list-fst	54	46	40	85.2	74.1
list-hd	26	17	14	65.4	53.8
list-inc	86	18	—	20.9	—
list-last	50	46	23	92.0	46.0
list-length	45	28	28	62.2	62.2
list-map	112	90	38	80.4	33.9
list-nth	124	114	51	91.9	41.1
list-snoc	181	181	46	100.0	25.4
list-insertion-sort	122	98	—	80.3	—
list-stutter	55	46	43	83.6	78.2
list-take	197	137	67	69.5	34.0
list-tl	32	29	26	90.6	81.3
list-tuple-swap	64	58	46	90.6	71.9
list-unzip	76	70	58	92.1	76.3
nat-dec	20	18	—	90.0	—
nat-div2	40	35	—	87.5	—
nat-iseven	27	23	—	85.2	—
tree-collect-leaves	111	101	—	91.0	—
tree-count-leaves	155	126	—	81.3	—
tree-count-nodes	108	86	—	79.6	—
tree-inorder	104	90	—	86.5	—

Figure 12: Reduction in refinement sizes using new specification features. The left section contains refinement sizes of the original benchmark suite, updates using disjunction and negation, and updates using polymorphism. The right section contains refinement sizes as percentages relative to the original refinements. Changes in synthesis times due to these enhancements were negligible.

With polymorphism, a subset of cases see especially dramatic improvement. Polymorphism prevents the synthesizer from decomposing the elements of the refinement lists, narrowing the range of programs it can consider and thereby reducing the complexity of the refinement necessary to specify the one we want. For example, in the non-polymorphic `list-dict` benchmarks, which treat lists of `nat` pairs as dictionaries, the synthesizer attempted to increment or decrement keys and values rather than construct the expected dictionary operations. Designing a working specification required carefully selected patterns of numbers to prevent the synthesizer from doing so. With polymorphism, the keys and values could remain abstract, facilitating a more natural refinement.

Correspondingly, polymorphic examples make it possible to concisely generalize about the behavior of a function on all cases in ways that are impossible with singletons or disjunction. The refinement for `map` from Section 4.1 is more than 66% smaller than that from Osera and Zdancewic and 57% smaller than what we could achieve even with disjunction and negation.

Although polymorphism alone is not powerful enough to write the kinds of generalized loop invariants that would replace the need for refinement trace completeness as described in Section 4.1, the richer specification language makes it possible to write simple *properties* (the list `nth` function always returns None on an empty list) that subsume many examples that trace completeness would otherwise demand (Nil $\rightarrow$ nat $\rightarrow$ None).

The programs we tested in this experiment were selected because they were amenable to specification size reduction. In examples where we struggled to decrease refinement sizes, there were several features that made doing so difficult or impossible.

- *Required full information.* Many functions needed all information contained in a singleton, preventing us from using disjunction or negation to reduce the information content of a refinement. This was especially true of mathematical functions (natural number `sum`) and those requiring comparison (tree `insert`).
- *Complex properties.* Other functions had properties that could not be conveyed even in the richer specification language. Expressing that `sum` always produces its second argument when its first is 0 requires dependency, as does noting that `xor` returns false when its arguments are equal.
- *Minimal useful redundancy.* Some functions lacked redundancy that we could exploit. List `snoc`, which appends an element onto the end of a list, works in the opposite direction of the Cons constructor, making arguments difficult to structurally merge. List `snoc` did prove to be a beneficiary of polymorphism.

5. Related Work

So far, we have presented a framework that interprets synthesis with examples as the type inhabitation problem over a refinement type system. In addition to suggesting efficient implementation strategies, this foundational approach captures traits common to existing example-directed synthesis work. In this section, we survey the research literature to make these connections concrete.

MYTH [22] was a significant inspiration for this work, as it demonstrated that (simple) type- and example-directed search for the eta-long, beta-normal forms of small functional programs was tractable. The major difference between the two pieces of work is that MYTH considers examples to be a syntactically distinct class from types. It defines productive rules for decomposing examples, but fails to make a direct connection to the literature on intersection types and intuitionistic logic. This link explains why the MYTH synthesis system works efficiently. In their introduction, Osera and Zdancewic note informally that the intuition behind their system is an effort to "enumerate [programs] during evaluation" rather than to "enumerate [programs] and then evaluate [them]." The difference between these two strategies is formalized in the distinction between the synchronized and unsynchronized sequent calculi. Furthermore, the connection to refinement types allows our system to provide a specification language extending beyond MYTH's "examples." We demonstrated that these features allow users to specify their intentions more compactly than in MYTH. MYTH's synthesizer relies on natural deduction, although some rules were modified in sequent calculus style with focusing [11]. We apply these ideas to the entire framework.

Polikarpova and Solar-Lezama's work on SYNQUID [24] also uses refinement types to synthesize functional programs. SYNQUID includes dependent and polymorphic types, but not unions. A key contribution of the work is an algorithm for synthesis of polymorphic instantiations. In contrast to the work here, the authors do not connect their type system to intuitionistic logic or demonstrate how to interpret examples as type-based specifications.

ESCHER [1] also synthesizes programs using examples but operates over an untyped, Lisp-like language. Nevertheless, the system uses a form of forward search, generating terms in order of increasing size and checking whether they agree with any examples. This search strategy corresponds closely to a bottom-up reading of our rules. Furthermore, their SPLITGOAL rule which, given an expression that partially satisfies the examples, searches for another that satisfies the remainder and a boolean guard that partitions the example space accordingly, corresponds precisely to a bottom up reading of our if rule S-ITE along with disjunction refinements.

Several systems, such as λ^2 [9] and FLASHEXTRACT [20], push examples through applications of specific functions using custom axioms. For example, if the code we are to synthesize contains a use of `map` e, then specialized axioms set up a new goal for e, without having to "sample" from (*i.e.,* execute) `map`. We could likely add custom refinement checking rules for specific, built-in functions in a similar manner, though we have not done so. More generally, these systems do not make the connection to type refinements, but it would be interesting to investigate the interactions between such rewriting axioms and our refinements in the future.

Systems such as IGORII [17] and MAGICHASKELLER [16] perform example rewriting as part of their synthesis processes. Their primary mode of rewriting finds the *least general generalization* of the examples via antiunification and turns the sub-components where the examples differ into synthesis subproblems. They do not make an explicit connection with type refinement systems.

Scherer's work [26] on type inhabitation as program synthesis uses logical techniques similar to ours—focusing and proof search to find inhabitants of a type—but specifications are limited to simple types and polymorphism. Our work demonstrates how we can integrate additional forms of specification, namely examples via refinements. INSYNTH [15] is similar in that they also study type inhabitation as program synthesis without examples. To distinguish between likely and unlikely candidate functions, they introduce a ranking system with weights derived from emperical analysis of large corpuses of code. It seems likely that such ranking heuristics could be used in conjunction with our system.

Rehof studied the synthesis of programs through the lens of type inhabition of a finite combinatory logic with intersection types [25]. He uses intersection types in conjunction with semantic types to provide a rich specification language for synthesis in a system called CLS [6]. However, whereas our work employs intersection types in conjunction with singletons to provide an example-based specification language, their work focuses on a type-based logical specification language. Furthermore, the domains of the respective works are vastly different. CLS is limited to component-based synthesis—typed function symbols and applications of those symbols—whereas we synthesize programs with a larger range of program features, in particular, pattern matching over algebraic data types.

All of the related work discussed so far has interacted only with positive examples. In contrast, our system is capable of dealing with negative examples, *i.e.*, counterexamples. Counterexamples are at the heart of the CEGIS algorithm [27] that powers solver-based synthesizers such as SKETCH [27] and LEON [18]. The key difference between our approach and theirs is *where* the counterexamples originate: we require the user to supply them whereas the solvers that power SKETCH and LEON generate these counterexamples automatically as part of a CEGIS loop.

We owe much to past work on refinement type systems such as those of Freeman and Pfenning [13], Davies [5], Dunfield [7, 8] and others. We also borrow from the work of Barbanera [4], who defines natural deduction and sequent calculi for intersections and unions, though he does not use such systems for synthesis or evaluate their performance as theorem provers. Since his sequent calculus contains cut, it is less useful for automated proof search.

6. Future Work

Refinement extensions. Although we have expanded the vocabulary of example-directed synthesis to include a far richer and more general collection of constructs, there are a veriety of other components that we might adopt from type theory. Dependent types and refinements would allow us to write more expressive specifications than those descripted in Section 2.3. Similarly, we might add true base types — integers, floating point numbers, and strings — with corresponding theories integrated into the refinement system.

Library intake. Although we focus mainly on self-contained synthesis problems, real-world synthesis systems need the ability to make use of libraries of existing code. The approaches for doing so developed in this paper remain primitive, enumerating arguments for library functions in hopes that the proper expression will eventually be discovered. These strategies can only use libraries in cases where the candidate arguments are described by refinements that correspond to values, since evaluating a library on any other argument is impossible. Furthermore, the prototype is incapable of integrating polymorphic libraries, since doing so requires enumerating type parameters before doing the same for arguments. A more powerful library-intake system might make use of symbolic execution (rather than evaluation) to produce more general and descriptive refinements for libraries.

Intelligent proof search. Much of Section 4 explores methods for coping with the enormous nondeterminism inherent in the sequent calculus even after employing focusing and invertibility. Strategy 0, the naive refinement-directed search, fails because the synthesizer faces an overwhelming number of branches to explore in a breadth-first manner. Attempting even to visit each branch once proves prohibitively expensive. Rather than attempting to deviate from Strategy 0, perhaps we should instead reconsider the premise of performing breath-first search over the space of candidate programs. One such avenue would be to build a learning theorem-prover that intelligently pursues particular paths first based upon context and past experience, an approach that might make seemingly vast search spaces more tractable to traverse. This information could even be gleaned by examining repositories of existing code to prioritize programs that are more likely to be written in practice.

Acknowledgments

We would like to thank the anonymous reviewers for their comments and feedback on our work. This work was supported in part by the NSF under awards CCF-1138996 and CNS-1111520.

References

[1] A. Albarghouthi, S. Gulwani, and Z. Kincaid. Recursive program synthesis. In *Computer Aided Verification*, pages 934–950, 2013.

[2] J.-M. Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):297–347, 1992.

[3] L. Augustsson. [haskell] announcing djinn, version 2004-12-11, a coding wizard. Mailing List, 2004. `http://www.haskell.org/pipermail/haskell/2005-December/017055.html`.

[4] F. Barbanera, M. Dezaniciancaglini, and U. Deliguoro. Intersection and union types. *Information and Computation*, 119(2):202–230, June 1995.

[5] R. Davies. A practical refinement-type checker for standard ml.

[6] B. Düdder, M. Martens, and J. Rehof. Staged composition synthesis. In *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, pages 67–86, 2014. . URL `http://dx.doi.org/10.1007/978-3-642-54833-8_5`.

[7] J. Dunfield. *A unified system of type refinements*. PhD thesis, Carnegie Mellon University, 2007.

[8] J. Dunfield and F. Pfenning. Tridirectional typechecking. pages 281–292, 2004.

[9] J. K. Feser, S. Chaudhuri, and I. Dillig. Synthesizing data structure transformations from input-output examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2015.

[10] K. Fisher, D. Walker, K. Q. Zhu, and P. White. From dirt to shovels: Fully automatic tool generation from ad hoc data. 2008.

[11] J. Frankle. Type-directed synthesis of products, Oct. 2015. URL `http://arxiv.org/abs/1510.08121`.

[12] J. Frankle, P.-M. Osera, D. Walker, and S. Zdancewic. Example-directed synthesis: A type-theoretic interpretation (extended version). Technical Report MS-CIS-15-12, University of Pennsylvania, 2015.

[13] T. Freeman and F. Pfenning. *Refinement types for ML*, volume 26. ACM, 1991.

[14] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *ACM SIGPLAN Notices*, volume 46, pages 317–330. ACM, 2011.

[15] T. Gvero, V. Kuncak, I. Kuraj, and R. Piskac. Complete completion using types and weights. In *Proceedings of the 2013 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2013.

[16] S. Katayama. An analytical inductive functional programming system that avoids unintended programs. In *Proceedings of the ACM SIGPLAN 2012 Workshop on Partial Evaluation and Program Manipulation*, PEPM '12, pages 43–52, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1118-2.

[17] E. Kitzelmann. *A Combined Analytical and Search-based Approach to the Inductive Synthesis of Functional Programs*. PhD thesis, Fakulät für Wirtschafts-und Angewandte Informatik, Universität Bamberg, 2010.

[18] V. Kuncak, M. Mayer, R. Piskac, and P. Suter. Complete functional synthesis. In *Proceedings of the 31th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, 2010.

[19] T. Lau. *Programming by Demonstration: a Machine Learning Approach*. PhD thesis, University of Washington, 2001.

[20] V. Le and S. Gulwani. Flashextract: A framework for data extraction by examples. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14. ACM, 2014.

[21] Microsoft Corporation. Microsoft by the numbers, 2015. URL `http://news.microsoft.com/bythenumbers/ms\_numbers.pdf`.

[22] P.-M. Osera and S. Zdancewic. Type-and-example-directed program synthesis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2015.

[23] F. Pfenning. Automated theorem proving, 2004. URL `http://www.cs.cmu.edu/~fp/courses/atp/index.html`.

[24] N. Polikarpova and A. Solar-Lezama. Program synthesis from polymorphic refinement types, Oct. 2015.

[25] J. Rehof and P. Urzyczyn. Finite combinatory logic with intersection types. In L. Ong, editor, *Typed Lambda Calculi and Applications*, volume 6690 of *Lecture Notes in Computer Science*, pages 169–183. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-21690-9. . URL `http://dx.doi.org/10.1007/978-3-642-21691-6_15`.

[26] G. Scherer and D. Rèmy. Which simple types have a unique inhabitant? In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2015.

[27] A. Solar-Lezama. *Program Synthesis by Sketching*. PhD thesis, University of California, Berkeley, 2008.

[28] P. Wadler. Theorems for free! In *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 347–359. ACM, 1989.