Equivalence by Canonicalization for Synthesis-Backed Refactoring

JUSTIN LUBIN, University of California, Berkeley, USA
JEREMY FERGUSON*, University of California, Berkeley, USA
KEVIN YE*, University of California, Berkeley, USA
JACOB YIM*, University of California, Berkeley, USA
SARAH E. CHASINS, University of California, Berkeley, USA

We present an enumerative program synthesis framework called *component-based refactoring* that can refactor "direct" style code that does not use library components into equivalent "combinator" style code that does use library components. This framework introduces a sound but incomplete technique to check the equivalence of direct code and combinator code called *equivalence by canonicalization* that does not rely on input-output examples or logical specifications. Moreover, our approach can repurpose existing compiler optimizations, leveraging decades of research from the programming languages community. We instantiated our new synthesis framework in two contexts: (i) higher-order functional combinators such as `map` and `filter` in the statically-typed functional programming language Elm and (ii) high-performance numerical computing combinators provided by the NumPy library for Python. We implemented both instantiations in a tool called Cobbler and evaluated it on thousands of real programs to test the performance of the component-based refactoring framework in terms of execution time and output quality. Our work offers evidence that synthesis-backed refactoring can apply across a range of domains without specification beyond the input program.

CCS Concepts: • **Software and its engineering → Automatic programming**.

Additional Key Words and Phrases: Program Synthesis, Refactoring, Program Equivalence Checking

ACM Reference Format:
Justin Lubin, Jeremy Ferguson, Kevin Ye, Jacob Yim, and Sarah E. Chasins. 2024. Equivalence by Canonicalization for Synthesis-Backed Refactoring. *Proc. ACM Program. Lang.* 8, PLDI, Article 223 (June 2024), 26 pages. https://doi.org/10.1145/3656453

1 INTRODUCTION

Functional programming languages often provide programmers with higher-order reusable components to achieve functionality like mapping a function over a list. For example, a programmer can write in a *combinator* style by stringing together these components using the "pipeline" operator (`|>`) common in languages like Elm, OCaml, and F#, here expressed in Elm:

*These authors contributed equally (order determined alphabetically).

Authors' Contact Information: Justin Lubin, justinlubin@berkeley.edu, University of California, Berkeley, Berkeley, California, USA; Jeremy Ferguson, jmfergie@berkeley.edu, University of California, Berkeley, Berkeley, California, USA; Kevin Ye, yekeviny@berkeley.edu, University of California, Berkeley, Berkeley, California, USA; Jacob Yim, jacobyim@berkeley.edu, University of California, Berkeley, Berkeley, California, USA; Sarah E. Chasins, schasins@cs.berkeley.edu, University of California, Berkeley, Berkeley, California, USA.

ACM 2475-1421/2024/6-ART223
https://doi.org/10.1145/3656453

```
main : (String → Bool) → (String → List Int) → List String → List Int
main p f list =
  list |> List.filter p |> List.map f |> List.concat
```

An equivalent *direct* style of this Elm program could be written as follows:

```
main : (String → Bool) → (String → List Int) → List String → List Int
main p f list =
  case list of
    [] → []
    head :: tail → if p head then f head ++ main p f tail else main p f tail
```

Lubin and Chasins [70] found that even when programmers may sometimes prefer the former (combinator) style of code, it can be more difficult to write from scratch than the latter (direct) style. The work includes quotes from programmers like:

> I really prefer the pipeline for readability, but it's a little bit harder to write because I don't feel like I have the streamlined version of how it's going to look, whereas if I write the `case` down, it's telling me, here's all the possibilities, just handle them—like, start with this and go to the next one.

In this paper, we present a component-based program synthesis framework called *component-based refactoring* to synthesize combinator-style equivalents of direct-style programs—that is, to synthesize compositions of library components that are semantically equivalent to a programmer's reference program—without any further specification. We instantiate our technique in two domains: (i) refactoring Elm programs (as above) and (ii) refactoring Python programs that use explicit `for` loops into programs that use NumPy's numeric computing combinators.

Our technique soundly refactors programs using only the direct-style reference program. Previous component-based synthesis approaches typically rely on one of two additional specifications:

(1) *Input-output examples.* λ^2 [40], Bigλ [102] and SyPet [39] use input-output examples as the specification for component-based synthesis (although they do not target refactoring tasks). To use one of these approaches to perform refactoring, we would need to (i) generate an appropriate set of inputs to form a specification, (ii) run the reference program on the selected inputs to produce input-output examples, and (iii) run candidate synthesis solutions on this set of input-output examples to check for behavior match. However, for (i), generating a complete set of inputs is not possible for all programs, making this approach unsound, and for (ii) and (iii), it is not always possible to run the programs if they are, for example, snippets of larger programs, rely on external resources, take a long time to execute, or do not terminate.

(2) *Logical specifications.* Oracle-guided inductive synthesis [55] and verified lifting [18, 26] can perform component-based synthesis for refactoring by relying on SMT solvers. Unlike an example-based refactoring approach, these approaches are sound and do not require running the programs; however, they require complete logical specifications for the components and reference programs. To apply this technique in our setting, we would need to be able to infer logical specifications for the input reference programs. This approach works well for difficult-to-write system kernels, but requiring complete logical specifications in domains where specifications are difficult to express in SMT theories can be a heavy burden.

To apply these lines of work, one might ask: How can we automatically generate an appropriate set of input-output examples or infer logical specifications for the program snippets we want to refactor? Instead, we ask a different question: Given that we already have a direct-style program, *how can we bypass the need for input-output examples or logical specifications entirely?*

Key insight. Although off-the-shelf optimization techniques do *not* convert our input direct-style programs into our output combinator-style programs, they *can* go the other way around and convert

combinator-style programs into direct-style programs. For example, we can obtain the direct-style Elm program above from the preceding combinator-style program via the classic off-the-shelf optimization technique of *catamorphism fusion* from the algebra of programming [20, 76], which is a generalization of deforestation [115]. In fact, direct-style programs often *look like* programs on which various compiler optimizations—inlining, constant folding, deforestation—have already been performed. Thus, for the particular problem of component-based refactoring—where we aim to identify whether a composition of components is equivalent to an inlined, constant-folded, and deforested program—the techniques we need for checking equivalence are exactly the same terminating, syntactic transformations that our community has been inventing for decades to use as compiler optimizations.

More broadly, if we have a terminating syntactic transformation ϕ that satisfies the property that any two input programs mapped to the same output program are semantically equivalent, then we can soundly check whether two programs P_1 and P_2 are *semantically* equivalent by checking if $\phi(P_1)$ and $\phi(P_2)$ are *syntactically* equivalent; we call such a function ϕ a *canonicalization function*. While this check—which we call *equivalence by canonicalization*—is not (and cannot be) complete for a Turing-complete language, if a canonicalization function maps many input programs that a programmer might write to the same output program while satisfying this property, then equivalence by canonicalization will be useful in practice despite its incompleteness. Our synthesis algorithm leverages this technique by enumerating compositions of library components until equivalence by canonicalization soundly deduces that a candidate solution is semantically equivalent to the reference program.

Contributions. This paper contributes:

(1) An automated technique to check the semantic equivalence of programs called *equivalence by canonicalization* (§4) that requires no further specification beyond the programs' source code. We also contribute a synthesis framework built around equivalence by canonicalization called *component-based refactoring* (§5). We prove both equivalence by canonicalization and component-based refactoring sound and provide guidance on how to use them in practice.
(2) Two instantiations of the component-based refactoring framework (§6): one for higher-order functional combinators, and one for high-performance numerical computing combinators.
(3) An implementation of these instantiations called Cobbler, evaluated on thousands of real programs from The Stack [60] to test the performance of the component-based refactoring framework, both in terms of execution time and quality of outputs (§8).

2 OVERVIEW

To illustrate the core insight of our approach, we consider a non-recursive example. Consider the following component library:

```
map : (a → a) → Maybe a → Maybe a
map f mx = case mx of Nothing → Nothing | Just x → Just (f x)

withDefault : a → Maybe a → a
withDefault d mx = case mx of Nothing → d | Just y → y
```

Suppose we wish to use this component library to refactor the following function:

```
main : (Int → Int) → Maybe Int → Int                    Input
main f mx = case mx of Nothing → 0 | Just x → f (f x)
```

Our algorithm uses top-down enumerative synthesis to generate, among other candidates, the following sketch, which we will call S:

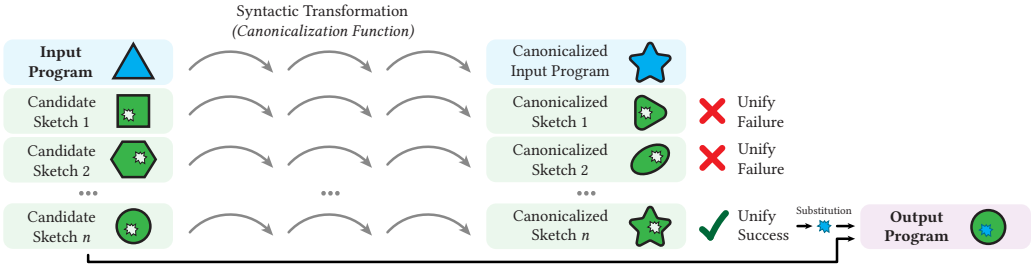


Fig. 1. **A visual overview of our component-based refactoring algorithm.** Given a library of components to refactor an input program with, the algorithm enumerates candidate library programs with holes (sketches). The algorithm applies a fixed syntactic transformation (called a *canonicalization function*) to both the provided input program and the enumerated candidate sketches until it finds a sketch whose canonicalized form unifies with the canonicalized input program. The algorithm concludes by using the result of this unification to fill the holes in the matching candidate sketch.

```
withDefault ?1 (map ?2 ?3)
```

The key question is then: How can we check whether this candidate sketch S can be used to refactor the input function, `main`? As a first step, we might consider inlining the library components into the candidate expression, which yields:

```
case (case ?3 of Nothing → Nothing | Just x → Just (?2 x)) of
  Nothing → ?1
  Just y → y
```

At this point, the candidate sketch is still quite distinct from the `main` function syntactically; even unification modulo the equational theory of the λ-calculus (*higher-order unification*) fails. However, drawing from the literature on optimizing compilers, we can apply a semantics-preserving program transformation known as *deforestation* [115], yielding:

```
case ?3 of Nothing → (case Nothing of Nothing → ?1 | Just y → y)
           Just x → (case Just (?2 x) of Nothing → ?1 | Just y → y)
```

The original motivation for deforestation was to remove intermediate allocations in the combinator-style code that functional programmers often want to write. It turns out that what makes this transformation so useful for optimizing functional code is exactly what makes it useful for this synthesis problem as well: It applies to code that functional programmers often want to write!

If we now partially evaluate any `case` expression whose scrutinee is a constructor literal on the deforested example, we arrive at the following sketch:

```
case ?3 of Nothing → ?1 | Just x → ?2 x
```

Higher-order unification of this sketch with the original `main` function now yields the substitution $\{$`?1` $\mapsto$ `0`, `?2` $\mapsto$ `λz. f (f z)`, `?3` $\mapsto$ `mx`$\}$, which, when applied to the enumerated sketch S, yields:

```
withDefault 0 (map (λz. f (f z)) mx)
```
Output

Figure 1 illustrates how we use this idea for our synthesis framework: We enumerate many candidate programs and apply a fixed syntactic transformation to each one until one unifies with the transformed input program. In the example above, the transformation is deforestation and unification is higher-order unification. While this technique is necessarily incomplete, our evaluation in Section 8 shows that this approach can be made to work well in practice.

Key insight: Naïve inlining (even modulo the equational theory of the λ-calculus) is not sufficient to unify a candidate sketch and the reference program; success requires an additional syntactic transformation (in this case, deforestation).

In Section 4, we define precisely what properties this syntactic transformation must satisfy; intuitively, it cannot map semantically distinct programs to the same program. In Section 6, we describe two particular such transformations, one that works for refactoring with higher-order functional combinators in the Elm programming language (as in the non-recursive example from this section and the recursive example from Section 1), and one that works for refactoring with numerical computing combinators in the NumPy library [50] for the Python programming language.

3 PROBLEM STATEMENT

We now formalize our problem statement via a series of definitions.

Definition 3.1. The **sketch closure** of a set X equipped with a notion of composition (such as function application) is notated $\mathrm{cl}_? X$ and is the closure of X under composition among elements in the set and the infinite set of **named holes** $\{?_i : i \in \mathbb{N}\}$, where we call i the **name** of the hole $?_i$. We call the elements of $\mathrm{cl}_? X$ **sketches**, even if they do not contain a hole.

Definition 3.2. For our purposes, a **language** $\mathcal{L}$ has

(1) an inductively defined set of programs $\mathsf{Prog}_{\mathcal{L}}$ (whose equality is syntactic and decidable);
(2) a set of values $\mathsf{Val}_{\mathcal{L}}$ (whose equality may be undecidable); and
(3) a semantics $[\![\cdot]\!]_{\mathcal{L}} : \mathsf{Prog}_{\mathcal{L}} \to \mathsf{Val}_{\mathcal{L}}$ (that may not be computable).

We further suppose the existence of a finite **library** $\mathsf{Lib}_{\mathcal{L}} \subseteq \mathsf{Prog}_{\mathcal{L}}$ of components, whose (infinite) sketch closure $\mathrm{cl}_? \mathsf{Lib}_{\mathcal{L}}$ consists of what will call **component sketches**. When clear from context or irrelevant, we omit the subscript $\mathcal{L}$ on these terms.

Definition 3.3. A **hole substitution** for a language $\mathcal{L}$ is a finite mapping from hole names in $\mathbb{N}$ to closed terms of $\mathsf{Prog}_{\mathcal{L}}$. We notate the set of all hole substitutions for $\mathcal{L}$ by $\Sigma_{\mathcal{L}}$ (again occasionally omitting the subscript). The **application of a hole substitution** $\sigma \in \Sigma_{\mathcal{L}}$ to a sketch $S \in \mathrm{cl}_? \mathsf{Prog}_{\mathcal{L}}$ is notated σS and is defined by replacing holes $?_i$ in the domain of σ present in S with $\sigma(i)$.

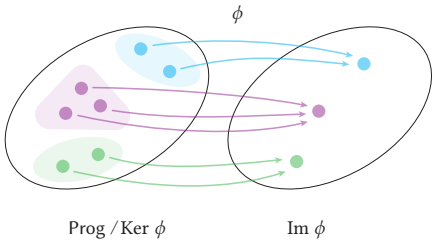
With these definitions, our goal is as follows:

*Definition 3.4 (**Problem statement**).* Given a program $P \in \mathsf{Prog}$, find a component sketch $S \in \mathrm{cl}_? \mathsf{Lib}$ and hole substitution $\sigma \in \Sigma$ such that $[\![\sigma S]\!] = [\![P]\!]$ without any further specification.

Remark 3.5. Sketches separate the parts of the solution that use library components from those that use fragments of the input program or other non-library expressions, which enables a variety of cost metrics, such as maximizing or minimizing the number of components in S, minimizing the number of holes in S, or something else entirely. Indeed, there is always at least one solution ($S = ?_0$ with $\sigma = [0 \mapsto P]$) and possibly infinitely many (if the identity function id is in Lib, then the n-fold composition of identity functions $S = \mathsf{id}^n(?_0)$ with $\sigma = [0 \mapsto P]$ is a solution for all $n \in \mathbb{N}$). Thus, in practice, an algorithm must balance between "too few" and "too many" components.

4 EQUIVALENCE BY CANONICALIZATION

Our problem statement requires establishing the semantic equivalence of a reference program P and a candidate filled component sketch σS in that it requires $[\![\sigma S]\!] = [\![P]\!]$. However, checking the equivalence of $[\![\sigma S]\!]$ and $[\![P]\!]$ may be undecidable, as, in general, (i) equality of values is undecidable



Algorithm 1 Basic equivalence by canonicalization

Parameter: Basic canonicalization function ϕ for $\mathcal{L}$
Input: Two programs $P_1, P_2 \in \text{Prog}_{\mathcal{L}}$
Output: Either $\top$ or $\bot$
1: **if** $\phi(e_1) = \phi(e_2)$ **then return** $\top$ **else return** $\bot$

Fig. 2. **Basic equivalence by canonicalization is an algorithmic version of Noether's first isomorphism theorem.** Prog/Ker ϕ is the set of equivalence classes of programs with the same canonicalized form.

and (ii) the semantics $[\![\cdot]\!]$ is uncomputable. Thus, the best we can hope for is an algorithm that can, in many practical cases, tell us whether a candidate filled component sketch σS is semantically equivalent to the reference program P, but in some cases simply return $\bot$ to indicate uncertainty.

Before we define such an algorithm, we first establish a few standard preliminary definitions:

Definition 4.1 (Preliminaries). The **quotient** of a set X by an equivalence relation R is the set of equivalence classes $X/R = \{[x]_R : x \in X\}$. An equivalence relation R is a **refinement** of an equivalence relation R' (written $R \leq R'$) if $(x_1, x_2) \in R$ implies $(x_1, x_2) \in R'$. The **kernel** of f is the equivalence relation $\text{Ker}\, f = \{(x_1, x_2) \in X \times X : f(x_1) = f(x_2)\}$.

With these definitions, we can say that our problem definition requires establishing that P and σS are in the same equivalence class of the quotient $\text{Prog}/\text{Ker}\, [\![\cdot]\!]$. These definitions also enable us to leverage the ideas behind *Noether's first isomorphism theorem for sets* (illustrated in Figure 2):

Theorem 4.2 (Noether's first isomorphism theorem for sets). *If X and Y are sets and $\phi : X \to Y$, then the image of ϕ is isomorphic to $X/\text{Ker}\, \phi$ via the bijection $[x]_{\text{Ker}\, \phi} \mapsto \phi(x)$.*

Key insight. The key insight is then to *choose ϕ such that* $\text{Ker}\, \phi$ *is a refinement of* $\text{Ker}\, [\![\cdot]\!]$. Noether's first isomorphism theorem then enables us to *soundly check the semantic equivalence of P_1 and P_1 by checking the syntactic equivalence of $\phi(P_1)$ and $\phi(P_2)$*. This motivates the following definition:

Definition 4.3. A function $\phi : \text{Prog}_{\mathcal{L}} \to \text{Prog}_{\mathcal{L}}$ is a **basic canonicalization function** for $\mathcal{L}$ if (i) it is computable and (ii) $\text{Ker}\, \phi \leq \text{Ker}\, [\![\cdot]\!]$.

Example 4.4. Let $\mathcal{L}$ be the language of integer arithmetic. Then ϕ_1, ϕ_2, and ϕ_3 below are basic canonicalization functions for $\mathcal{L}$ (even though ϕ_1 and ϕ_3 are not semantics-preserving) because they are computable and $\phi_i(P_1) = \phi_i(P_2)$ (syntactic equality) implies $[\![P_1]\!] = [\![P_2]\!]$ (semantic equality):

$$\phi_1(P) = [\![2 \cdot P]\!]; \qquad \phi_2(P) = P \text{ with all instances of } x \cdot 1 \text{ rewritten to } x; \qquad \phi_3(P) = 0 \cdot P.$$

However, the functions $f_1(P) = [\![0 \cdot P]\!]$ and $f_2(P) = [\![P \cdot P]\!]$ are not basic canonicalization functions for $\mathcal{L}$ because $f_i(-1) = f_i(1)$ yet $[\![-1]\!] \neq [\![1]\!]$.

We can now define Algorithm 1—an algorithmic version of Noether's first isomorphism theorem—to check whether two programs are semantically equivalent, then prove its soundness:

Theorem 4.5. *Algorithm 1 always terminates, and if it returns $\top$ on (ϕ, P_1, P_2), then $[\![P_1]\!] = [\![P_2]\!]$.*

For concision, we provide all proofs in an appendix in the supplementary materials.

Key takeaway: We can check semantic equivalence of programs using a combination of (i) syntactic equivalence and (ii) particular kinds of computable syntactic transformations.

Algorithm 1 captures the insight of our approach, but the following example shows that it suffers from two key limitations, which we overcome in the next section: (i) it cannot handle sketches, and (ii) syntactic equality can be too restrictive.

Example 4.6. Suppose ϕ is a basic canonicalization function for the simply-typed λ-calculus and M and N are terms with $\phi(M) = ?_1\ x$ and $\phi(N) = f\ (f\ x)$ (roughly as in Section 2). Conceptually, ϕ establishes the semantic equivalence of M and N even though $\phi(M)$ has a hole and $\phi(M) \neq \phi(N)$ because if $?_1$ is set to $\lambda z.\ f\ (f\ z)$, then the two terms are equal modulo the equational theory of the λ-calculus. We need (i) to allow holes to be filled and (ii) more flexibility than syntactic equality.

4.1 Explicitly Handling Holes

To handle cases like those in Example 4.6, we first introduce holes to our language as follows.

Definition 4.7. A **hole-free** language $\mathcal{L}$ is one with $\mathsf{Prog}_{\mathcal{L}} \cap \{?_i : i \in \mathbb{N}\} = \emptyset$ and $\mathsf{Val}_{\mathcal{L}} \cap \{?_i : i \in \mathbb{N}\} = \emptyset$. The **sketch language** of a hole-free language $\mathcal{L}$ is notated $\mathcal{L}^?$ and defined by setting

(1) $\mathsf{Prog}_{\mathcal{L}^?} = \mathsf{cl}_? \mathsf{Prog}_{\mathcal{L}}$;
(2) $\mathsf{Val}_{\mathcal{L}^?}$ = all mappings from $\Sigma_{\mathcal{L}}$ to $\mathsf{Val}_{\mathcal{L}}$;
(3) $[\![S]\!]_{\mathcal{L}^?} = \sigma \mapsto [\![\sigma S]\!]_{\mathcal{L}}$; and
(4) $\mathsf{Lib}_{\mathcal{L}^?} = \mathsf{Lib}_{\mathcal{L}}$.

We now define the most relaxed possible generalization of syntactic equality in this new language. Although it is undecidable because it relies on computing semantics and value equality, we will use it to define a correctness criterion for a broader notion of canonicalization functions momentarily.

Definition 4.8. Let $\mathcal{L}$ be a hole-free language. The **semantic unification relation** $\equiv_{\mathcal{L}} \subseteq \mathsf{Val}_{\mathcal{L}^?} \times \mathsf{Val}_{\mathcal{L}^?} \times \Sigma_{\mathcal{L}}$ for $\mathcal{L}$ holds on V_1, V_2, σ (written $V_1 \equiv^{\sigma}_{\mathcal{L}} V_2$) if $V_1(\sigma) = V_2(\sigma)$. A hole substitution $\sigma \in \Sigma_{\mathcal{L}}$ **semantically unifies** sketches $S_1, S_2 \in \mathsf{Prog}_{\mathcal{L}^?}$ if $[\![\sigma S_1]\!]_{\mathcal{L}} = [\![\sigma S_2]\!]_{\mathcal{L}}$ (i.e., $[\![S_1]\!]_{\mathcal{L}^?} \equiv^{\sigma}_{\mathcal{L}} [\![S_2]\!]_{\mathcal{L}^?}$). A relation $U \subseteq \mathsf{Prog}_{\mathcal{L}^?} \times \mathsf{Prog}_{\mathcal{L}^?} \times \Sigma_{\mathcal{L}}$ is a **partial semantic unification relation** for $\mathcal{L}$ if for all $S_1, S_2 \in \mathsf{Prog}_{\mathcal{L}^?}$ and $\sigma \in \Sigma_{\mathcal{L}}$ such that $U(S_1, S_2, \sigma)$, σ semantically unifies S_1 and S_2.

To take advantage of this definition, we must first define a suitable generalization of kernels:

Definition 4.9. Let X, Y, and Z be sets. The **generalized kernel** of a function $f : X \to Y$ with respect to a relation $R \subseteq Y \times Y \times Z$ is $\mathrm{Ker}_R f = \{(x_1, x_2, z) \in X \times X \times Z : R(f(x_1), f(x_2), z)\}$.

Remark 4.10. This definition generalizes the notion of a function kernel in the following sense. Let X and Y be sets with $f : X \to Y$. Take $Z = \{\top\}$ and $R(y_1, y_2, \top)$ to hold if and only if $y_1 = y_2$. Then $\mathrm{Ker}_R f \simeq \mathrm{Ker}\, f$ via the bijection $(y_1, y_2, \top) \mapsto (y_1, y_2)$. Alternatively, if we define $R'(y_1, y_2)$ to hold if $\exists z \in Z.\ R(y_1, y_2, z)$ and R' is an equivalence relation, then we can define $f' : X \to Y/R'$ by $f'(x) = [f(x)]_{R'}$ and view $\mathrm{Ker}_R f$ as a version of $\mathrm{Ker}\, f'$ that "keeps track" of the witness $z \in Z$ that causes $R(f(x_1), f(x_2), z)$ to hold for $x_1, x_2 \in X$.

We can now define the full form of a canonicalization function.

Definition 4.11. A function $\phi : \mathsf{Prog}_{\mathcal{L}^?} \to \mathsf{Prog}_{\mathcal{L}^?}$ is a **canonicalization function** for a hole-free language $\mathcal{L}$ with respect to a relation $U \subseteq \mathsf{Prog}_{\mathcal{L}^?} \times \mathsf{Prog}_{\mathcal{L}^?} \times \Sigma_{\mathcal{L}}$ if (i) it is computable and (ii) $\mathrm{Ker}_U\, \phi \leq \mathrm{Ker}_{\equiv_{\mathcal{L}}} [\![\cdot]\!]_{\mathcal{L}^?}$.

Remark 4.12. A basic canonicalization function is a canonicalization function with respect to the relation that discards its third argument and checks syntactic equality of its first two arguments.

The following lemma is one convenient way to make a canonicalization function (however, as in Example 4.4, canonicalization functions need not be semantics-preserving). We provide other sufficient conditions for canonicalization functions in an appendix in the supplementary materials.

Algorithm 2 Equivalence by canonicalization: $\textsc{EbC}(\phi, u, S_1, S_2)$

Parameter: Canonicalization function ϕ for hole-free language $\mathcal{L}$ with respect to U
Parameter: Inference algorithm u for U
Input: Two sketches $S_1, S_2 \in \text{cl}_? \text{Prog}_{\mathcal{L}}$
Output: Either a hole substitution σ or $\bot$
1: **if** $u(\phi(S_1), \phi(S_2)) = \sigma$ **then return** σ **else return** $\bot$

Lemma 4.13. *Let U be a partial semantic unification relation for $\mathcal{L}$ and $\phi : \text{Prog}_{\mathcal{L}^?} \to \text{Prog}_{\mathcal{L}^?}$ be computable and semantics-preserving. Then ϕ is a canonicalization function for $\mathcal{L}$ with respect to U.*

Finally, before generalizing Algorithm 1, we introduce one more definition.

Definition 4.14. Let X, Y, and Z be sets. A **semi-inference algorithm** for a relation $R \subseteq X \times Y \times Z$ is a computable function $r : X \times Y \to Z \cup \{\bot\}$ with the property that if $r(x, y) = z \in Z$, then $R(x, y, z)$; r is an **inference algorithm** if $r(x, y) = \bot$ implies there is no $z \in Z$ such that $R(x, y, z)$.

We now introduce the full equivalence by canonicalization algorithm in Algorithm 2. *The crucial subtlety of this algorithm is that we can soundly apply the substitution σ generated by u on the **outputs** of ϕ to the **inputs** of ϕ*; this is precisely why we need to "keep track" of hole substitutions as described in Remark 4.10. The following soundness theorem captures exactly this property.

Theorem 4.15 (Termination and soundness of equivalence by canonicalization). *Let ϕ be a canonicalization function for a hole-free language $\mathcal{L}$ with respect to U and u be an inference algorithm for U. Then $\textsc{EbC}(\phi, u, \cdot, \cdot)$ is a semi-inference algorithm for $\text{Ker}_{\equiv_{\mathcal{L}}} [\![\cdot]\!]_{\mathcal{L}^?}$.*

Key takeaway: To support sketches and equality modulo theories, we can loosen the restriction of syntactic equivalence when performing equivalence by canonicalization.

4.2 How Should We Choose a Canonicalization Function?

Algorithm 2 is parameterized by a canonicalization function ϕ, but not all choices of ϕ are equally useful. Intuitively, we want to choose a function ϕ that *maps many input programs to the same output program* while still satisfying the canonicalization function requirements. We can formalize this idea by again turning to the notion of a refinement. Refinements give a necessary condition for basic canonicalization functions ($\text{Ker}\ \phi$ must be a refinement of $\text{Ker}\ [\![\cdot]\!]$), but they also tell us "how good" a choice of ϕ is: we can say that ϕ_1 is better choice than ϕ_2 if $\text{Ker}\ \phi_2$ is a strict refinement of $\text{Ker}\ \phi_1$—that is, if ϕ_1 maps more expressions to the same result than does ϕ_2. To extend this idea to full canonicalization functions, we first introduce the following auxiliary definition.

Definition 4.16. Let X and Z be sets. For a relation $K \subseteq X \times X \times Z$, the **forgotten version** of K is $\mathcal{F}(K) = \{(x_1, x_2) \in X \times X : \exists z \in Z.\ K(x_1, x_2, z)\}$.

Whereas before we wanted to keep track of hole substitutions, now we explicitly do *not* want to do so when assessing if one canonicalization function is better than another; so long as the hole substitutions are valid, we do not care whether the use of one canonicalization function returns the same hole substitutions as the use of another, but rather whether the use of one canonicalization function succeeds *more often* than the use of another. The following definition leverages the preceding auxiliary definition to formalize this idea, which we visually depict in Figure 3.

Definition 4.17 (Goodness of canonicalization functions). A canonicalization function ϕ_1 with respect to U_1 is **no worse than** a canonicalization function ϕ_2 with respect to U_2 (written $\phi_1 \succeq \phi_2$) if $\mathcal{F}(\text{Ker}_{U_1}\ \phi_1) \geq \mathcal{F}(\text{Ker}_{U_2}\ \phi_2)$. If $\mathcal{F}(\text{Ker}_{U_1}\ \phi_1) > \mathcal{F}(\text{Ker}_{U_2}\ \phi_2)$, then ϕ_1 is **better than** ϕ_2 ($\phi_1 \succ \phi_2$).

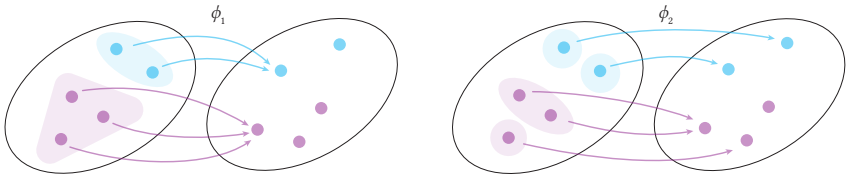


Fig. 3. The canonicalization function ϕ_1 is better than the canonicalization function ϕ_2 ($\phi_1 \succ \phi_2$) because the equivalence classes of the kernel of ϕ_1 are coarser than those of ϕ_2.

Example 4.18. Let $\mathcal{L}$ be the language of integer arithmetic. Then ϕ_1, ϕ_2, and ϕ_3 from Example 4.4 (extended to $\mathcal{L}^?$) are canonicalization functions for $\mathcal{L}$ with respect to the syntactic unification relation $\approx$ and $\phi_1 > \phi_2 > \phi_3$. For example, $(2+1, 3) \in \mathcal{F}(\mathrm{Ker}_\approx\, \phi_1) \setminus \mathcal{F}(\mathrm{Ker}_\approx\, \phi_2)$ because $\phi_1(2+1) = 6 = \phi_1(3)$ yet $\phi_2(2+1) = 2+1 \not\approx 3 = \phi_2(3)$; moreover, $(2 \cdot 1, 2) \in \mathcal{F}(\mathrm{Ker}_U\, \phi_2) \setminus \mathcal{F}(\mathrm{Ker}_U\, \phi_3)$ because $\phi_2(2 \cdot 1) = 2 = \phi_2(2)$ yet $\phi_3(2 \cdot 1) = 0 \cdot (2 \cdot 1) \not\approx 0 \cdot 2 = \phi_3(2)$.

Theorem 4.19 (Better canonicalization implies better equivalence checking). *Let ϕ_1 and ϕ_2 be canonicalization functions for a hole-free language $\mathcal{L}$ with respect to U_1 and U_2 and let u_1 and u_2 be inference algorithms for U_1 and U_2. Suppose $\phi_1 \succeq \phi_2$. Then:*

(1) If $\textsc{Ebc}(\phi_2, u_2, S_1, S_2) = \sigma_2$ for some hole substitution $\sigma_2 \in \Sigma$, then $\textsc{Ebc}(\phi_1, u_1, S_1, S_2) = \sigma_1$ for some hole substitution $\sigma_1 \in \Sigma$.

(2) If $\phi_1 \succ \phi_2$, there exist $S_1, S_2 \in \mathrm{cl}_?\, Prog$ with $\textsc{Ebc}(\phi_2, u_2, S_1, S_2) = \bot$ yet $\textsc{Ebc}(\phi_1, u_1, S_1, S_2) = \sigma_1$ for some hole substitution $\sigma_1 \in \Sigma$.

Remark 4.20. Theorem 4.19 provides a theoretical grounding for what makes a good canonicalization function. In practice, the goal is to choose ϕ to map component sketches and *programs that the user is likely write* to the same output program; accordingly, equivalence by canonicalization benefits greatly from accurate program characterization of the target programming domain.

Key takeaway: We should choose canonicalization functions that map many input programs to the same output program.

5 SYNTHESIS FRAMEWORK

We now describe our top-level synthesis framework. At a high level, the framework is an enumerative synthesizer whose search space is candidate sketches and whose satisfaction criterion is given by equivalence by canonicalization (Algorithm 2).

Definition 5.1. An **enumerator** for a set X is an algorithm $\textsc{Enum}(\cdot)$ that, given a reference program in X, lazily generates a (possibly infinite) sequence of elements of X.

Remark 5.2. There are many kinds of enumerators in the synthesis literature [10, 67, 78, 88]; the choice of enumerator is orthogonal to the techniques and soundness results we present here.

Using this definition of an enumerator and our earlier definition of equivalence by canonicalization in Algorithm 2, we now introduce our component-based refactoring algorithm in Algorithm 3, which we illustrated earlier in Figure 1. The following theorem establishes its soundness, proving that, upon termination, it solves the problem posed in Definition 3.4.

Theorem 5.3 (Soundness of component-based refactoring). *Let ϕ be a canonicalization function for a hole-free language $\mathcal{L}$ with respect to U, u be an inference algorithm for U, $\textsc{Enum}$ be an enumerator $Lib_{\mathcal{L}}$, and P be a program in $Prog_{\mathcal{L}}$. If Algorithm 3 terminates, it returns a component sketch $S \in \mathrm{cl}_?\, Lib_{\mathcal{L}}$ and hole substitution σ such that $[\![\sigma S]\!]_{\mathcal{L}} = [\![P]\!]_{\mathcal{L}}$.*

Algorithm 3 Component-based refactoring

Parameter: Canonicalization function ϕ for hole-free language $\mathcal{L}$ with respect to U
Parameter: Inference algorithm u for U
Parameter: Enumerator Enum for $\mathsf{Lib}_{\mathcal{L}}$
Input: A program $P \in \mathsf{Prog}_{\mathcal{L}}$
Output: A component sketch $S \in \mathsf{cl}_? \, \mathsf{Lib}_{\mathcal{L}}$ and hole substitution σ
1: **for** $S \in \textsc{Enum}(P)$ **do**
2: **if** $\textsc{EbC}(\phi, u, P, S) = \sigma$ **then return** (S, σ)

Remark 5.4 (Completeness and termination). The completeness of Algorithm 3 is dependent on the choices of $\mathsf{Lib}_{\mathcal{L}}$, the canonicalization function ϕ, the relation U, and the enumerator Enum. For sufficiently complex languages and libraries, it is impossible to achieve completeness (which would amount to solving the halting problem), but in general two changes bring the algorithm "closer" to completeness: (i) increasing the coverage of Enum and (ii) choosing ϕ high in the better-than partial order (Definition 4.17). Of course, (i) introduces a tension between completeness and termination: If the enumerator provides an infinite sequence of programs, then Algorithm 3 may not terminate. Otherwise, however, it will terminate, as equivalence by canonicalization always terminates.

Key takeaway: To solve the component-based refactoring problem posed in Definition 3.4, we can slot the equivalence by canonicalization algorithm into an enumerative synthesizer by using it as a notion of satisfaction.

Remark 5.5 (Optimizations). An easy optimization is to inline the equivalence by canonicalization procedure and only compute $\phi(P)$ once. A more exciting possibility is to leverage *term indexing* to separate the algorithm into an "offline" and "online" stage. Briefly, the offline stage would run the enumerator up to a fixed bound and store each canonicalized sketch as a key and the original sketch as a value; then, the online stage would canonicalize a given input program and look it up in the term index. Term indexes for languages beyond first-order logic are a long-running and active area of research [31, 43, 44, 57, 68, 93, 110]; consequently, we defer this extension to future work.

6 FRAMEWORK INSTANTIATIONS

To demonstrate the applicability of the component-based refactoring (CBR) synthesis framework to different domains, we instantiated it in two different contexts:

(§6.1) **CBR-Elm**: Refactoring code in the statically-typed functional programming language Elm that uses pattern matches and explicit recursion into code that uses higher-order combinators like `List.map` and `List.filter`.

(§6.2) **CBR-Python**: Refactoring code in Python that uses `for` loops into code that uses numerical computing functions provided by the NumPy library like `np.sum` and `np.convolve`.

In this section, we discuss the high-level choices we made to instantiate the synthesis framework we described in Section 5; we defer low-level implementation details to Section 7. We present these instantiations via detailed worked examples, as the exact formal definitions of the syntactic transformations we rely on (such as catamorphism fusion) are not a contribution of our work.

As we discussed in Remark 3.5, algorithms solving the problem statement in Definition 3.4 must balance between choosing "too few" and "too many" library components; we opt to return the smallest solution with *at least one* component (ruling out the trivial solution of always returning a hole) by performing top-down enumeration starting from single components of the library rather

than starting from a top-level hole. As we will see in the user study in Section 8.3 and performance evaluation in Section 8.4, this heuristic was sufficient to result in desirable programs.

6.1 Higher-Order Functional Combinators in Elm

For CBR-Elm, we synthesize programs using 17 functions mostly from the Elm standard library (listed in an appendix in the supplementary materials) relating to the `Bool`, `Maybe`, `Result`, and `List` types (we include `find` and `findMap` functions for `List`, which happen to be absent in the standard library). We note, however, that our approach extends to arbitrary user-provided libraries with no additional work beyond writing the functions in standard Elm code.

To make CBR-Elm's task harder, we purposely do not include the `Maybe`, `Result`, and `List` catamorphisms in its library (known as `maybe`, `either`, and `foldr` in Haskell) even though to do so would simply require defining these functions in standard Elm code (which we do in Section 9.1). Many real-world functions that use structural recursion can easily be rewritten to use a single catamorphism without any synthesis whatsoever (and, as we will see, this is actually the first step to our canonicalization function). Thus, to force CBR-Elm to find nontrivial compositions of components, we do not provide it the opportunity to return the trivial result of a single catamorphism.

Our CBR-Elm canonicalization function operates with respect to higher-order unification[1] and has two main stages: (1) rewrite the program using catamorphisms, and (2) perform catamorphism fusion wherever possible. We present a worked example here with the notion of a catamorphism specialized to `if` and `foldr` and refer the reader to Meijer et al. [76] for additional background.

Consider the following input program:

Input

```
main p f xs =
  case xs of [] → []
             hd :: tl → if p hd then f (f hd) :: main p f tl else main p f tl
```

CBR-Elm produces the following output for this program:

Output

```
List.map (λx. f (f x)) (List.filter p xs)
```

To do so, CBR-Elm first canonicalizes the input program. Step (1) is to rewrite the right-hand side using catamorphisms:

```
foldr (λy z → if p y then f (f y) :: z else z) [] xs
```

Step (2) is to perform catamorphism fusion, which in this case is not possible.

CBR-Elm then enumerates sketches such as `List.map` $?_1$ `(List.filter` $?_2$ $?_3$`)` to canonicalize. Step (1) of canonicalization is to rewrite the sketch using catamorphisms:

```
foldr (λhd acc → ?1 hd :: acc) []
  (foldr (λhd acc → if ?2 hd then hd :: acc else acc) [] ?3)
```

Step (2) is to perform catamorphism fusion to rewrite the program into the form `foldr g []` $?_3$. Meijer et al. [76]'s catamorphism fusion theorem says that it is sufficient for `g` to satisfy

```
g y (foldr (λhd acc → ?1 hd :: acc) [] ?3) =
    foldr (λhd acc → ?1 hd :: acc) [] (if ?2 y then y :: ?3 else ?3)
```

for all `y`, which is non-constructive in that it does not provide a definition for `g`. Partial evaluation [41, 56] does not help solve for `g` here, but we can recursively perform catamorphism fusion on the `if` expression (also known as *deforestation* [115]):

[1]Technically, higher-order unification is undecidable [42]. Therefore, we settle for the finer relation of running Huet's higher-order unification semi-decision algorithm [52] for 1,000 steps. We could have also chosen from the many variants of higher-order unification such as higher-order pattern unification [86], which is complete for a strict subset of the λ-calculus.

```
g y (foldr (λhd acc → ?1 hd :: acc) [] ?3) =
  if ?2 y then foldr (λhd acc → ?1 hd :: acc) [] (y :: ?3)
          else foldr (λhd acc → ?1 hd :: acc) [] ?3
```

We can then perform partial evaluation on the `then` branch, which yields:

```
g y (foldr (λhd acc → ?1 hd :: acc) [] ?3) =
  if ?2 y then ?1 y :: foldr (λhd acc → ?1 hd :: acc) [] ?3
          else foldr (λhd acc → ?1 hd :: acc) [] ?3
```

Finally, we can replace `foldr (λhd acc → ?1 hd :: acc) [] ?3` with a fresh variable `z` to obtain `g y z = if ?2 y then ?1 y :: z else z`, resulting in the following overall canonicalized sketch:

```
foldr (λy z → if ?2 y then ?1 y :: z else z) [] ?3
```

This expression higher-order unifies with the canonicalized input program, resulting in the substitution $\{?_1 \mapsto \lambda x.\ f\ (f\ x), ?_2 \mapsto p, ?_3 \mapsto xs\}$ and the overall output listed above.

By Lemma 4.13, this procedure is a canonicalization function for Elm with respect to higher-order unification because it is computable and semantics-preserving—with one exception. Elm is a strict language, and catamorphism fusion is technically valid only under a lazy evaluation scheme because it can change whether or not a program diverges; even in Haskell, the presence of the `seq` operator makes catamorphism fusion an unsound transformation in general. We inherit the general limitation of syntactic transformations often becoming unsound in the presence of diverging code.

6.2 Numerical Computing Combinators in NumPy

For CBR-Python, we synthesize programs operating over 1D arrays using 21 NumPy functions (listed in an appendix in the supplementary materials), including some that return numbers such as `np.sum`, fixed-sized arrays such as `np.add` and `np.convolve`, and variable-sized arrays such as filtering (e.g. `x[x > 0]`). We also include two additional functions we call *cosmetic* (`list` and `np.vectorize`) that provide no performance benefits but can expose opportunities to apply other functions.

Unlike in CBR-Elm, our CBR-Python canonicalization function is specific to these library functions, and adding additional functions would require additional work; it would be interesting future work to compute the necessary rewrites automatically via array-aware program slicing.

Our CBR-Python canonicalization function operates with respect to a small set of relations backed by an e-graph, including symmetric arithmetic rewrites such as $a \leftrightarrow a \cdot 1$ and asymmetric rewrites that capture NumPy broadcasting functionality (in which a scalar can be treated as an array) such as $a[i] = n \rightarrow a[i] = \textsc{Broadcast}(n)[i]$. It has two main stages: (1) non-recursively inline the top-level NumPy function call, if possible, and (2) perform partial evaluation on the result.

We present two worked examples to illustrate this procedure. First, consider the following input program that computes a vector dot product of `x` and `y`:

Input

```
s = 0
for i in range(len(x)):
    s += x[i] * y[i]
s
```

CBR-Python produces the following output for this program:

Output

```
np.sum(np.multiply(x, y))
```

CBR-Python's canonicalization function happens not to modify this input program. However, consider the candidate sketch `np.sum(np.multiply(?1, ?2))`. Step (1) of canonicalization is to inline the top-level function (below, left), and Step (2) is to perform partial evaluation (below, right).

```
?3 = 0
for ?4 in range(len(np.multiply(?1, ?2))):
    ?3 += np.multiply(?1, ?2)[?4]
?3
```

```
?3 = 0
for ?4 in range(len(?1)):
    ?3 += ?1[?4] * ?1[?4]
?3
```

Here partial evaluation uses the rules `len(np.multiply(a, b))` ⟶ `len(a)` and `np.multiply(a, b)[i]` ⟶ `a[i] * b[i]`. The second expression unifies with the canonicalized input program, yielding the substitution {?1 ↦ x, ?2 ↦ y, ?3 ↦ s, ?4 ↦ i} and overall output above.

As a second example, consider the following input program that computes a rolling sum of `x`:

Input

```
y = np.zeros(len(x) - WINDOW_SIZE + 1)
for i in range(len(y)):
    s = 0
    for j in range(WINDOW_SIZE):
        s += x[i + j]
    y[i] = s
y
```

CBR-Python produces the following output for this program:

Output

```
np.convolve(x, np.full(WINDOW_SIZE, 1), mode="valid")
```

Again, CBR-Python's canonicalization function happens not to modify this input program. However, consider the candidate sketch `np.convolve(?1, np.full(?2, ?3), mode="valid")`. Step (1) is to inline the top-level NumPy function:

```
?4 = np.zeros(len(?1) - len(np.full(?2, ?3)) + 1)
for ?5 in range(len(?4)):
    ?6 = 0
    for ?7 in range(len(np.full(?2, ?3))):
        ?6 += ?1[?5 + ?7] * np.full(?2, ?3)[len(np.full(?2, ?3)) - ?7 - 1]
    ?4[?5] = ?6
?4
```

Step (2) is to perform partial evaluation. Here the partial evaluator uses the rules `len(np.full(a, b))` ⟶ `a` and `np.full(a, b)[i]` ⟶ `b`:

```
?4 = np.zeros(len(?1) - len(np.full(?2, ?3)) + 1)
for ?5 in range(len(?4)):
    ?6 = 0
    for ?7 in range(?2):
        ?6 += ?1[?5 + ?7] * ?3
    ?4[?5] = ?6
?4
```

This unifies with the canonicalized input program (modulo $a \leftrightarrow a \cdot 1$), yielding the substitution {?1 ↦ x, ?2 ↦ WINDOW_SIZE, ?3 ↦ 1, ?4 ↦ y, ?5 ↦ i, ?6 ↦ s, ?7 ↦ j} and overall output above.

Unfortunately, it is impossible to define a semantics-preserving program transformation in Python due to the ability of Python code to arbitrarily inspect the call stack. Such features make it impossible to prove any program transformation is a canonicalization function. However, for the set of programs for which the above procedure *is* semantics-preserving, Lemma 4.13 proves that it is a canonicalization function; in practice, we believe this space of programs to be quite large, and none of the synthesized programs in our Section 8.4 evaluation fell outside this space.

7 IMPLEMENTATION

We implemented our component-based refactoring instantiations in a tool called Cobbler in approximately 5,000 lines of OCaml code (not including the frontend nor the experimental setup we describe in Section 8). We support only a subset of Elm and Python, omitting features such as partially-applied data constructors and nested patterns in Elm, and dictionaries, method calls, and statements not of the form of an assignment, a call to `.append()`, an `if`, or a `for` in Python. To work with e-graphs, we use the OCaml library Ego [114], which is based on the Rust library egg [117].

8 EVALUATION

Although we are most interested in the synthesis framework itself—rather than our two particular instantiations—we used the instantiations as opportunities to explore if our framework can work well in practice. Thus, for our empirical evaluation, we investigated two main research questions:

RQ1. How fast does Cobbler run on real-world programs?
RQ2. To what extent does Cobbler improve real-world programs?

We investigated these questions via three experiments detailed in the following sections using a corpus of programs from The Stack dataset [60]. In Experiment 1, we answered **RQ1** by measuring Cobbler's synthesis time. In Experiment 2, we answered **RQ2** for Elm by running a user study ($n = 159$) to explore when Cobbler's outputs are preferable to its inputs. In Experiment 3, we answered **RQ2** for Python by measuring how long Cobbler's outputs take to run on increasing data sizes (accounting for synthesis time) compared to its inputs.

Our evaluation does not compare against a baseline. An appropriate baseline would need to take a program and, without further specification, soundly refactor it using a given set of library components. The Haskell linter `hlint` [80] technically meets these requirements, as it provides hard-coded linting rules that replace pattern matches over `Maybe` expressions with (single) combinators. However, `hlint` does not support refactoring over other types or using more than one component, yet, as we will see, only 16% of Cobbler Elm transformations required just one component. Thus, we felt a comparison against `hlint` would not be informative to understanding Cobbler nor be fair to `hlint` given its purpose as a linter rather than synthesizer. As we discuss in Section 10.2.1, other than hard-coded approaches, Cobbler is the first tool in the category we outline above.

Additional questions. In addition to our main research questions, we asked three additional research questions to gain a better fundamental understanding of our framework and instantiations:

RQ3. How much does semantic unification help Cobbler?
RQ4. How does Cobbler scale with the number of components it uses for a solution?
RQ5. How long do each of the Cobbler sub-components take to run?

We answer **RQ3** with an ablation study, **RQ4** by running Cobbler on synthetic programs that require increasingly many components, and **RQ5** by timing Cobbler at a granular level.

8.1 Input Programs

We drew real-world input programs for our experiments from The Stack [60], a dataset of permissively-licensed open-source code from GitHub repositories that includes programs written in over 300 languages. From all 90,637 Elm files, we drew 3,371 Elm functions that immediately pattern-match on a variable of type `Maybe`, `Result`, or `List`; these constituted our input programs for CBR-Elm. From 1,000,000 of the Jupyter notebooks, we drew 572 cells that included a set of variable definitions followed by a `for` loop followed by a final line consisting of a variable, as in Section 6.2; these constituted our input programs for CBR-Python. These numbers exclude programs using features orthogonal to the synthesis task but which would require additional engineering effort, such as

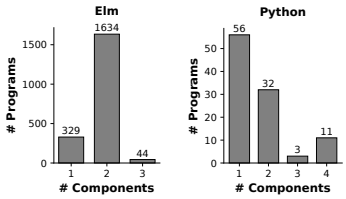


Fig. 4. **The number of components Cobbler's test set solutions use.** 1,678/2,007 Elm programs and 46/102 Python programs in the test set require more than a single component.

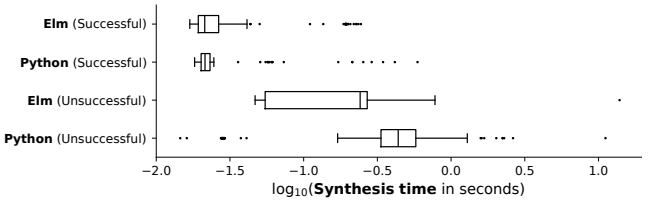


Fig. 5. **Cobbler's synthesis time on the test set broken down by language and whether or not synthesis succeeded.** Median synthesis time is <0.5s in all conditions. Cobbler is considered unsuccessful if it cannot find a solution in the provided component depth maximum (no hard time cutoff). Whiskers are ±1.5 × IQR.

nested patterns in Elm and dictionaries in Python (see Section 7) and Python programs using the pandas library [109]. Aside from this constraint, we drew all such Elm programs and Jupyter cells.

We split our input programs into training and test sets. We made all implementation decisions for Cobbler based on the training set (including which library components to use) and ran Cobbler on the test set only for the final evaluation.[2] To split the training and test sets, we split the input files into training and test sets, and all programs extracted from a given file went into the associated set. Using this process, we reserved 411 of the 3,371 Elm programs and 217 of the 572 Python programs for the training set, leaving a test set of 2,960 Elm programs and 355 Python programs.

8.2 Experiment 1: Answering RQ1 (on Cobbler Performance)

8.2.1 Setup. We ran Cobbler on each input program and collected the following information: (i) whether synthesis succeeded or failed, (ii) the synthesized program, if successful, and (iii) how long synthesis took (median of 10 runs). We did not use a hard time cutoff; instead, based on performance on the training set, we chose a synthesis depth cutoff of 3 for Elm and 4 for Python. We ran this experiment (and all following) on a 2020 MacBook Pro with a 2.3 GHz Quad-Core Intel Core i7 processor and 32 GB of RAM running macOS Big Sur with OCaml v4.14 and Python v3.11.

With this setup, Cobbler applied to 2,007/2,960 Elm programs and 102/355 Python programs[3] in the test set. Figure 4 summarizes how many components each refactored program contains; because Cobbler enumerates sketches in increasing size, this metric serves as a rough proxy for the complexity of the task and indicates that Cobbler does more than naïve inlining.

8.2.2 Results. **Cobbler's median synthesis time was less than 0.5s across both successful and unsuccessful runs for both Elm and Python, and all successful runs took less than 1s.**

Theoretically, as an exhaustive enumerative synthesizer, the asymptotic running time of Cobbler is at least exponential in the number of components used (which we validate in Section 8.5.2). In practice, the successful Elm and Python runs both took median synthesis times of 0.02s and the unsuccessful Elm and Python runs took median synthesis times of 0.21s and 0.39s respectively. Figure 5 summarizes the synthesis time for all input programs broken down by language and whether or not they succeeded. Figure 6 summarizes the synthesis times for successful runs broken down by language and the number of components used.

[2]With one exception: We noticed bugs in Cobbler's Python post-processing (orthogonal to the synthesis algorithm) and fixed those after running Cobbler on the test set. The end result of this is that Cobbler succeeded on one fewer program.
[3]Cobbler reports 6 additional successes, but upon manual inspection, we deemed these runs as failures due to bugs in our implementation and omitted them from all timing experiments. Cobbler also reported success on a run but did not properly namespace one of the components it used, so its definition was shadowed; we counted this as a success in our analysis.

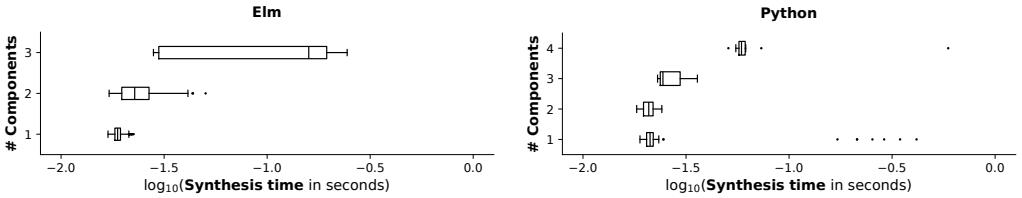


Fig. 6. **Cobbler's synthesis time on successful runs in the test set by language and number of components used.** For all such runs, Cobbler provides a synthesis result in <1s. Whiskers are ±1.5 × IQR.

Remark. Unsuccessful synthesis times can be arbitrarily inflated or deflated by adjusting the maximum enumeration depth, as an unsuccessful run means that Cobbler must consider the entire search space (except when Cobbler performs a lightweight, syntax-based early cutoff by checking that its library of functions is insufficient for the task). On a successful run, however, the maximum enumeration depth does *not* affect the synthesis time (other than possibly causing the successful run to become unsuccessful), as Cobbler will stop at the first solution in finds.

8.3 Experiment 2: Answering RQ2 (on Cobbler Improving Programs) for Elm

8.3.1 Setup. For Elm, we operationalize "improving programs" as increasing their subjective desirability to real statically-typed functional programmers, which we assessed via a user study approved by our institution's IRB. *However, we do not claim that combinator-style programs are always preferable to direct-style programs.* Consequently, a result that programmers always prefer the output of Cobbler to its input for Elm would be surprising.

To observe when Cobbler improves Elm programs, we randomly sampled the input-output program pairs from the test set of Experiment 1 whose outputs used more than one combinator. Within each of three categories—`Maybe`, `Result`, and `List` programs—we randomly sampled 200 such pairs. (In the case of `List`, the test set only included 62 such pairs, so we included all 62.) We then created a survey with randomized order that displayed a random sample of these pairs (equally distributed among the three types) and, for each pair, asked participants which program they preferred.[4] We did not refer to either program as "input," "output," "direct-style," "combinator-style" or any other descriptive label; we merely presented their source code in a random order.

We distributed this survey via email, Slack, and X (formerly Twitter) along with a request to share the survey more broadly. We did not compensate participants, and we requested that only self-identified statically-typed functional programmers fill out the survey. In total, we had 159 participants who overall answered 3,206 comparison questions.

8.3.2 Results. **Participants preferred combinators (Cobbler's output) in approximately one quarter of the `Maybe` and `Result` code (26% and 24% respectively) and approximately half of the `List` code (46%).** Although not necessarily indicative of causality, Figure 7 shows that participants more often preferred combinators for programs operating over a recursive datatype (`List`) than those operating over simpler non-recursive datatypes (`Maybe` and `Result`). Moreover, Figure 8 shows that participants tended to prefer combinators when the textual size of the program was much shorter than in the direct style, which was most often the case with `List` programs. We again stress that these trends do not necessarily indicate causality.

[4]The survey also asked participants to indicate which program they thought was more readable, but the results were nearly identical, so we do not report on them further here.

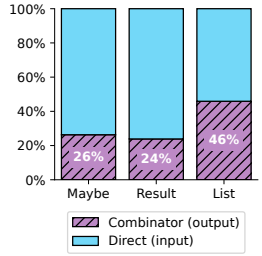


Fig. 7. **Preference by program type.** How often participants preferred the direct-style program vs. the equivalent combinator-style program.

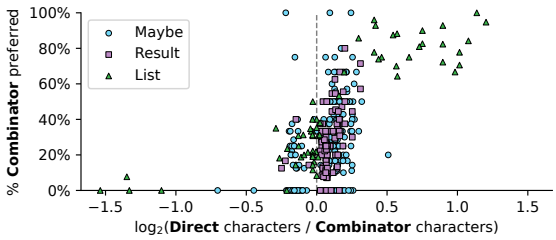


Fig. 8. **Preference by program size change.** How often participants preferred the combinator style for a given program vs. how much shorter the combinator style is compared to the direct style.

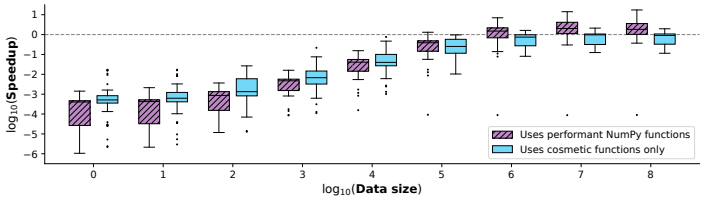


Fig. 9. **The speedup (accounting for synthesis time) of the synthetic Python program variants as their input data size increases.** The data is broken down by whether the refactored program uses NumPy functions intended to increase performance or only cosmetic functions (`list`, `np.vectorize`). By a data size of 10^6, the median speedup for refactored programs that use performant functions is >1×; by 10^8, this median is 1.95× and 42/56 programs are sped up >1×. Whiskers are ±1.5 × IQR.

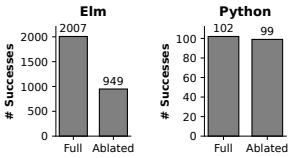


Fig. 10. **The effect of ablating Cobbler to use only syntactic unification.** Ablated Cobbler successfully synthesizes 949 Elm programs (compared to 2,007) and 99 Python programs (compared to 102).

8.4 Experiment 3: Answering RQ2 (on Cobbler Improving Programs) for Python

8.4.1 Setup. For Python, we operationalize "improving programs" as increasing their performance. Most input programs from Experiment 1 load input data from the authors' file systems (which we cannot access), load other external input data (such as online resources, also often inaccessible), do not define their input data, or operate on small data represented in the text of the program (for which the equivalent NumPy code is actually slower than the naïve Python). Thus, to observe when Cobbler could provide helpful speedup, we took the successfully-refactored Python programs, manually modified both the input and output programs to operate on data of sizes of $10^0, 10^1, \ldots, 10^8$, and recorded how long each of the sixteen variants took to terminate (median of 10 runs).

8.4.2 Results. **When using performant NumPy functions, Cobbler's outputs are more often faster (even including synthesis time) than its inputs with a data size of at least 10^6; at a data size of 10^8, the median speedup is 1.95×, and 42/56 programs exhibit speedups.** In Section 6.2, we discussed that not all components are intended to improve performance; some are merely cosmetic. We break down our performance results—which we visualize in Figure 9—based on whether or not Cobbler refactors the program to use at least one non-cosmetic function. To account for synthesis time, we define *speedup* as

$$\text{original program execution time} \div (\text{refactored program execution time} + \text{Cobbler synthesis time})$$

and plot this quantity over input data size in Figure 9.

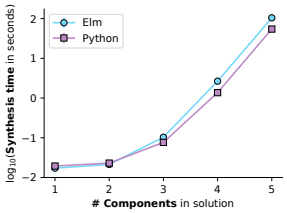


Fig. 11. **Cobbler's synthesis time on increasingly large synthetic problems.** Synthesis time is median over 10 runs (min-max error bars smaller than points).

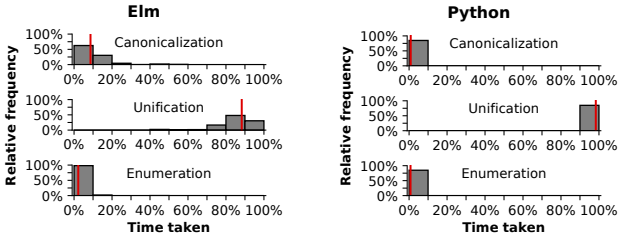


Fig. 12. **Cobbler's synthesis time broken down by time spent in canonicalization, unification, and enumeration.** The data includes successful and unsuccessful runs, and vertical bars represent medians.

We used only 100/102 of the transformed Python programs for this experiment because two input programs exponentiated integers to powers that scaled with the size of the input data; these two input programs did not terminate on all data sizes within one hour or crashed due to overflow.

8.5 Additional Experiments: Answering RQ3, RQ4, and RQ5

*8.5.1 Answering **RQ3** (on How Much Semantic Unification Helps Cobbler).* We ran Cobbler on the same programs as in Experiment 1, but instead of using higher-order unification for Elm and e-unification for Python, we used syntactic unification (modulo holes). We found that **semantic unification approximately doubles the number programs Cobbler can refactor for Elm, but has little effect for Python.** We plot this result in Figure 10.

*8.5.2 Answering **RQ4** (on Cobbler's Scalability).* We ran Cobbler on programs we constructed to require increasingly many components and measured how long Cobbler took to find the solutions. We found that **Cobbler scales at least exponentially in the number of components it uses for a solution**, as expected of an exhaustive enumerative synthesizer We plot this result in Figure 11.

*8.5.3 Answering **RQ5** (on Cobbler's Timing Breakdown).* We ran Cobbler on the same programs as in Experiment 1, but with more granular timing instrumentation. Specifically, we measured the time Cobbler spent (median of 10 runs) during (i) canonicalization, (ii) unification, and (iii) enumeration (not including canonicalization or unification). We found that **Cobbler spends most of its time performing unification**. We plot this result in Figure 12.

8.6 Threats to Validity

As we discuss in Section 7, Cobbler does not support the entirety of the Elm and Python languages, which may introduce sample bias for the programs used for all three experiments. Additionally, using only The Stack (which contains programs only from GitHub) may introduce sample bias. For Experiment 2, although the input code is real-world code drawn from The Stack, participants indicated their code style preference removed from a larger context or codebase, which may not be representative of real-world conditions. We also ran Cobbler on functions that programmers decided to commit to GitHub in a direct style, which may not be representative of the kinds of functions programmers would want to run Cobbler on to convert to combinator style. Moreover, survey respondents may not be representative of the statically-typed functional programming community as a whole. For Experiment 3, we generated synthetic variants of the real programs from The Stack with varying input data sizes, which may not be representative of real-world conditions.

9 DISCUSSION AND LIMITATIONS

9.1 When Does Cobbler Fail?

Cobbler refactors many real-world programs, but why does it sometimes fail? To answer this question, we took programs Cobbler failed to refactor and attempted to manually refactor them. For CBR-Elm, we took all 3 failed `Maybe` programs and randomly sampled 10 failed `Result` programs and 10 failed `List` programs. For CBR-Python, we randomly sampled 20 failed Python programs.

For CBR-Elm, we successfully refactored all 23/23 failed Elm programs and found that:

- (Elm$_1$) 3/3 failed `Maybe` programs failed due to the limitation of our unification algorithm that it cannot unify terms of the form $\lambda f.\ ?\ x$ with $\lambda f.\ f\ x$.
- (Elm$_2$) 10/10 failed `Result` programs and 5/10 failed `List` programs failed due to an insufficient component library—specifically, not having catamorphisms (as we discuss in Section 6.1) or the trivial pattern-matching combinator `List.uncons`. We verified that these failures were due to an insufficient library by temporarily adding these functions to the component library and observing that Cobbler could indeed refactor these 15 programs successfully.
- (Elm$_3$) 2/10 failed `List` programs failed due to Cobbler only supporting catamorphisms and not other recursion schemes; specifically, these programs relied on `foldl`.
- (Elm$_4$) 3/10 failed `List` programs failed due to requiring substantial non-syntactic reasoning (two for using an early-cutoff search idiom that is not possible to capture directly with a catamorphism, and one for requiring an entirely different approach to use a catamorphism).

For CBR-Python, we successfully refactored 19/20 failed Python programs; we were unable to refactor the last one because it required jagged arrays, which NumPy does not support.

For 11/20 failed Python programs, we found that a small semantics-preserving manual modification to the input program enabled Cobbler to refactor the program (we checked this by performing the modification, then re-running Cobbler). We do not suggest that programmers should do any of these transformations; rather, we showcase these as a demonstrations of particular failure modes.

- (Py$_1$) 4/20 failed programs failed due to our insufficient support for `np.vectorize` (e.g., not introducing lambdas to vectorize over method accesses). Our manual modification was to introduce the relevant lambdas ourselves (e.g., defining `f = lambda x: x.m()`).
- (Py$_2$) 3/20 failed programs failed due to working with dictionaries (which we consider out of scope in Section 7). Our manual modification was to introduce a single helper variable that encapsulated all the dictionary operations to a single line as a pre-processing step.
- (Py$_3$) 2/20 failed programs failed due to our insufficient support for multi-argument `range` calls. Our manual modification was to modify such calls to use the single-argument version.
- (Py$_4$) 1/20 failed programs failed due to using a helper variable in the body of a loop. Our manual modification was to inline this variable.
- (Py$_5$) 2/20 failed programs failed because Cobbler does not synthesize complex expressions as arguments to some functions. For example, Cobbler does not synthesize the filter operation `(x + x)[(x + x) > 0]`. Our manual modification—which was a bigger hint that the modifications above—was to introduce intermediate names for some expressions.

There were no small modifications to the remaining 8/20 failed Python programs that would enable Cobbler to refactor them successfully:

- (Py$_6$) 4/20 failed programs failed due to an insufficiently strong canonicalization function for compositions of `np.where` with other functions.
- (Py$_7$) 2/20 failed programs failed due to working with multi-dimensional arrays or lists of arrays (which we consider out of scope in Section 6.2).

($\textsc{Py}_8$) 1/20 failed programs failed due to requiring substantial non-syntactic reasoning; specifically, for searching for the last element of a list satisfying a predicate, which requires subtle indexing with `np.argwhere`.

Takeaways. Failure mode $\textsc{Elm}_2$ shows that **the library of combinators has a large impact on what code can be refactored**, and $\textsc{Elm}_3$ suggests that implementing additional recursion schemes (which should be straightforward given their similar fusion laws) could be useful in practice.

Failure modes $\textsc{Py}_1$–$\textsc{Py}_5$ show that **equivalence by canonicalization fails if the canonicalization function is not good enough in the sense of Definition 4.17**. Given a canonicalization function ϕ with respect to U and two programs P_1, P_2 with $[\![P_1]\!] = [\![P_2]\!]$ yet $\neg U(\phi(P_1), \phi(P_2))$, it is always possible to extend ϕ and U to a better canonicalization function $\phi' \succ \phi$ with respect to U' such that $U'(\phi'(P_1), \phi'(P_2))$. Extending these canonicalization functions thus becomes an engineering tradeoff: Which extensions are worth the human effort of implementing them?

Indeed, the modifications in $\textsc{Py}_1$–$\textsc{Py}_4$ could easily be incorporated into CBR-Python's canonicalization function, but *property-based testing for equality* (PBT)—which unsoundly checks the equality of two programs by checking that they have the same output on many different inputs—needs no domain-specific rules. Overall, **equivalence by canonicalization trades completeness for soundness** (which is most evident in $\textsc{Elm}_4$ and $\textsc{Py}_8$) and PBT makes the opposite tradeoff.

Lastly, one limitation this discussion does not capture is that **equivalence by canonicalization requires source code to be available**, whereas PBT, for example, does not.

9.2 Practical Tips for Component-Based Refactoring

Although authors of canonicalization functions can directly use off-the-shelf transformations like catamorphism fusion, knowing *which* transformations to employ requires domain expertise. Unlike PBT, for example, we do not expect end users to be able to define new canonicalization functions to check semantic equivalence without such domain expertise. To make implementing new instantiations of component-based refactoring easier, we provide a short sequence of steps for bootstrapping a canonicalization function and unification relation:

(1) **Identify a baseline notion of equivalence for the domain that is as simple as possible to serve as the unification relation.** Our timing breakdown analysis in Section 8.5.3 shows that Cobbler spends most of its time performing unification. Intuitively, canonicalization functions are typically syntax-directed and unification typically requires non-directional reasoning, so complex unification relations can be a bottleneck. However, our ablation analysis in Section 8.5.1 shows that it is possible for this relation to be too simple; in the case of CBR-Python, even though our performance analysis in Section 8.2 shows that Cobbler runs fast in practice, our e-graph unification takes most of the time of synthesis, yet does not substantially increase the number of programs Cobbler can refactor.
(2) **Craft the input and output program for a simple two-component synthesis problem.** We found it helpful to start with the input and output programs for the non-recursive Elm example in Section 2 and the dot product Python example in Section 6.2.
(3) **Inline the components in the output program of the previous step**. It can be helpful to consider inlining both components (as in CBR-Elm) or just one of them (as in CBR-Python).
(4) **Manually rewrite the resulting program of the previous step to be more idiomatic for the domain.** For example, for Elm, we rewrote the `case` expression in the scrutinee of another `case` expression as a single `case` expression, and, for Python, we rewrote `np.multiply(x, y)[i]` as `x[i] * y[i]`.
(5) **Identify relevant (existing) program transformations that perform this rewrite.** Both CBR-Elm and CBR-Python perform a type of *fusion* (combining multiple data passes into

one), suggesting that fusion transformations may be fruitful to investigate first. If no existing transformations exactly apply, the rewrite will need to be generalized to a new transformation.

The goal for Step (4) is to rewrite the output program to be in a form that unifies with the input program under the relation defined in Step (1). If this unification is achieved, then the transformation in Step (5) constitutes a canonicalization function (assuming it satisfies the required properties); this canonicalization function can then be further expanded by repeating this process for more complicated examples. If this unification is not achieved, it may be because the relation defined in Step (1) is too simple or the example in Step (2) is too complicated for a single transformation.

10 RELATED WORK

10.1 Program Equivalence Checking

Equivalence by canonicalization is a form of sound, automated program equivalence checking. Although proof assistants [87, 89, 108] provide facilities for semi-automatically proving program equivalence, we restrict our review of related work to automatic program equivalence checkers.

10.1.1 Sound Automated Techniques. Program equivalence checking is a type of *relational verification*, or verification of properties that include multiple programs. Many existing relational verification tools [25, 29, 30, 69, 99, 105, 111], techniques [11, 12, 27, 32, 48, 81, 100, 112, 113], and formalisms [7, 13, 15, 62, 65, 118] depend on logical specification and verification. Techniques based on equality saturation and e-graphs [107, 117] enable syntactic reasoning, but it can be challenging to represent arbitrary canonicalization functions using first-order syntactic rewrite rules.

Normalization by evaluation [16] is an approach to normalization in which programs are evaluated to a semantic domain and reified back to the syntactic domain. A sound (but not necessarily complete) semantic equivalence check can be done by checking the syntactic equality of these resultant terms, as in equivalence by canonicalization. However, this process relies on evaluation, and thus may not terminate for a Turing-complete language. Equivalence by canonicalization can be viewed as a variant of normalization by evaluation in which (i) evaluation is swapped out for a syntactic transformation that satisfies some (but not all) properties of evaluation while still guaranteeing termination, and (ii) reification is unnecessary and thus omitted. Interestingly, Shashidhar et al. [100] and Verdoolaege et al. [113] also compute a limited semantics-preserving normal form of associative operators in the context of program dependence graphs by flattening binary associative operator applications into multi-ary applications, indicating that it may be fruitful to consider canonicalization functions over the language of program dependence graphs.

10.1.2 Unsound Automated Techniques. Many verifiers in the previous section can be used in *bounded relational verification*, in which loops and recursion are unrolled up to a fixed depth and the relational logical property (such as program equivalence) is verified on the resulting program, as in bounded model checking [19]. Alternatively, random testing techniques such as property-based testing [28] and coverage-guided fuzzing [121] do not require or infer logical specifications about the programs at hand, but do require running the programs many times on a variety of inputs.

10.2 Program Synthesis

10.2.1 Component-Based Synthesis. Component-based refactoring is a form of component-based synthesis, where the goal is to synthesize compositions of components. This contrasts with recursive functional synthesis, where the goal is to synthesize functions using direct recursion [3, 33, 36, 37, 59, 66, 71, 79, 82, 88, 94, 120]. Component-based synthesizers sometimes rely on logical specifications [55] or types [21, 39, 46, 53, 61, 73]; many use input-output examples [45], including some for functional combinators [40, 51, 102, 106] and numerical computing combinators [14, 83, 101].

Component-based synthesizers sometimes target refactoring, including verified lifting [1, 18, 26, 58, 64] (in which logical summaries of provided programs are inferred and fed to a logic-based synthesizer to find an equivalent formulation in a domain-specific language) and NGST2 [74] (in which a neural-guided synthesizer translates imperative code to functional code annotated with custom logical specifications using a bounded verification check). NGST2's neural architecture is orthogonal to our idea of equivalence by canonicalization; an interesting avenue of future work could be to slot their neural-guided search in as an enumerator in our framework.

Although JLibSketch [75] does not target refactoring, it uses *algebraic specifications* [49] to synthesize code, which get compiled to logical specifications in JSketch [54] and ultimately Sketch [104]. Like our canonicalization functions, algebraic specifications are syntactic rewrites. However, unlike our canonicalization functions, these rewrites must be of the form *pattern* $\Rightarrow$ *result*, where *pattern* and *result* are opaque compositions of library functions. It is therefore not clear to us that these specifications could encode transformations like the non-constructive catamorphism fusion we described in Section 6.1. Moreover, our framework dispenses with logical specifications entirely: canonicalization functions can reuse off-the-shelf transformations developed by the programming languages community without any encoding into logical systems.

Lastly, Smith and Albarghouthi [103] introduce a generalization of the popular optimization in enumerative synthesis to consider only terms in normal form (such as β-normal η-long λ-terms) by enumerating normal forms of arbitrary term rewriting systems. They rely on syntactic transformations not as a form of equivalence checking of candidates sketches as we do, but as an optimization for enumerating candidates in the first place.

10.2.2 Using Unification for Synthesis. Although anti-unification is common in program synthesis [2, 5, 6, 24, 34, 53, 77, 95, 96], unification is rarer, but has been used in e-graph based synthesis approaches [84, 85] and to ensure input-output example satisfaction [21].

10.2.3 Library Learning. In a sense, the converse to our problem statement is the *library learning problem*: given a corpus of code, find a library of components to rewrite it [22, 24, 34, 35].

10.3 Program Compilation

Our particular canonicalization functions are inspired by compiler optimizations such as fusion [4, 76, 115] and partial evaluation [41, 56]. Our NumPy synthesizer could also be understood as a form of search-based compilation [9, 90–92, 98, 116, 119] or automatic parallelization [8, 17, 23, 38, 47, 63, 97].

11 CONCLUSION

In this paper, we introduced a sound, automated semantic equivalence check called *equivalence by canonicalization*. This technique (i) requires only the source code as a specification and (ii) can leverage decades of work from the programming languages community on syntactic transformations that were first developed for optimizing compilers. We use this technique in our *component-based refactoring* synthesis framework, which translates direct-style programs (like those that use pattern matching and recursion in Elm or `for` loops and lists in Python) into combinator-style programs (like those that use higher-order functional combinators in Elm or numerical computing combinators provided by NumPy). We found that this technique allowed us to synthesize combinator-style variants of thousands of real programs. Moreover, synthesis based on equivalence by canonicalization is fast; the median synthesis time was less than half a second. We applied our technique to synthesize Elm programs that programmers often preferred and Python programs that ran 1.95× faster, even accounting for synthesis time. The applicability of this synthesis technique for two differing purposes suggests we can use synthesis to accomplish refactoring across a variety of domains without requiring specification beyond the input program.

DATA AVAILABILITY STATEMENT

Cobbler is open source and freely available at https://github.com/justinlubin/cobbler. We also provide an archived snapshot of the Cobbler codebase and evaluation setup (including the input programs we used) for reproduction as a virtual machine hosted on Zenodo [72].

ACKNOWLEDGMENTS

We are indebted to our anonymous survey participants and are deeply grateful to the statically-typed functional programming community for extensively resharing it. We are also grateful to our anonymous POPL '24 and PLDI '24 reviewers as well as Joe Cutler, Gabriel Matute, Mae Milano, Chandrakana Nandi, and Max Willsey for their insightful feedback. We thank Aaron VonderHaar for developing the open source `elm-format` tool that greatly helped us with parsing and formatting, Naman Jain for showing us The Stack, and Maria Elena Grimmett for help with our explanatory figures. This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. 1752814 and is supported in part by NSF grants FW-HTF 2129008 and CA-HDR 2033558, as well as by gifts from Google and EPIC Lab sponsors G-Research, Adobe, Google, and Microsoft. Sarah E. Chasins is a Chan Zuckerberg Biohub Investigator.

REFERENCES

[1] M. B. S. Ahmad, J. Ragan-Kelley, A. Cheung, and S. Kamil. 2019. Automatically Translating Image Processing Libraries to Halide. In *TOG*. https://doi.org/10.1145/3355089.3356549
[2] T. Akiba, K. Imajo, H. Iwami, Y. Iwata, T. Kataoka, N. Takahashi, M. Moskal, and N. Swamy. 2013. *Calibrating Research in Program Synthesis Using 72,000 Hours of Programmer Time*. Technical Report. Microsoft Research.
[3] A. Albarghouthi, S. Gulwani, and Z. Kincaid. 2013. Recursive Program Synthesis. In *CAV*.
[4] F. E. Allen and J. Cocke. 1971. *A Catalogue of Optimizing Transformations*.
[5] R. Alur, P. Černý, and A. Radhakrishna. 2015. Synthesis Through Unification. In *CAV*.
[6] R. Alur, A. Radhakrishna, and A. Udupa. 2017. Scaling Enumerative Program Synthesis via Divide and Conquer. In *TACAS*. https://doi.org/10.1007/978-3-662-54577-5_18
[7] T. Antonopoulos, E. Koskinen, T. C. Le, R. Nagasamudram, D. A. Naumann, and M. Ngo. 2023. An Algebra of Alignment for Relational Verification. In *POPL*. https://doi.org/10.1145/3571213
[8] U. Banerjee. 1979. *Speedup of Ordinary Programs*. Ph. D. Dissertation. University of Illinois at Urbana-Champaign.
[9] S. Bansal and A. Aiken. 2006. Automatic Generation of Peephole Superoptimizers. In *OSR*.
[10] S. Barke, H. Peleg, and N. Polikarpova. 2020. Just-in-Time Learning for Bottom-up Enumerative Synthesis. In *OOPSLA*. https://doi.org/10.1145/3428295
[11] G. Barthe, J. Manuel Crespo, and C. Kunz. 2011. Relational Verification Using Product Programs. In *FM*. https://doi.org/10.1007/978-3-642-21437-0_17
[12] G. Barthe, J. Manuel Crespo, and C. Kunz. 2016. Product Programs and Relational Program Logics. In *JLAMP*.
[13] G. Barthe, B. Grégoire, and S. Zanella Béguelin. 2009. Formal Certification of Code-Based Cryptographic Proofs. In *POPL*. https://doi.org/10.1145/1480881.1480894
[14] R. Bavishi, C. Lemieux, R. Fox, K. Sen, and I. Stoica. 2019. AutoPandas: Neural-Backed Generators for Program Synthesis. In *OOPSLA*. https://doi.org/10.1145/3360594
[15] N. Benton. 2004. Simple Relational Correctness Proofs for Static Analyses and Program Transformations. In *POPL*. https://doi.org/10.1145/964001.964003
[16] U. Berger and H. Schwichtenberg. 1991. An Inverse of the Evaluation Functional for Typed Lambda-Calculus. In *LICS*. https://doi.org/10.1109/LICS.1991.151645
[17] A. J. Bernstein. 1966. Analysis of Programs for Parallel Processing. In *Transactions on Electronic Computers*. https://doi.org/10.1109/PGEC.1966.264565
[18] S. Bhatia, S. Kohli, S. A. Seshia, and A. Cheung. 2023. Building Code Transpilers for Domain-Specific Languages Using Program Synthesis. In *ECOOP*. https://doi.org/10.4230/LIPIcs.ECOOP.2023.38
[19] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. 1999. Symbolic Model Checking without BDDs. In *TACAS*. https://doi.org/10.1007/3-540-49059-0_14
[20] R. Bird and O. de Moor. 1997. *Algebra of Programming*.
[21] H. Botelho Guerra, J. F. Ferreira, and J. Costa Seco. 2023. Hoogle★: Constants and λ-Abstractions in Petri-net-based Synthesis Using Symbolic Execution. In *ECOOP*. https://doi.org/10.4230/LIPIcs.ECOOP.2023.4

[22] M. Bowers, T. X. Olausson, L. Wong, G. Grand, J. B. Tenenbaum, K. Ellis, and A. Solar-Lezama. 2023. Top-Down Synthesis for Library Learning. In *POPL*. https://doi.org/10.1145/3571234

[23] M. Burke and R. Cytron. 1986. Interprocedural Dependence Analysis and Parallelization. In *CC*. https://doi.org/10.1145/12276.13328

[24] D. Cao, R. Kunkel, C. Nandi, M. Willsey, Z. Tatlock, and N. Polikarpova. 2023. babble: Learning Better Abstractions with E-Graphs and Anti-unification. In *POPL*. https://doi.org/10.1145/3571207

[25] J. Chen, J. Wei, Y. Feng, O. Bastani, and I. Dillig. 2019. Relational Verification Using Reinforcement Learning. In *OOPSLA*. https://doi.org/10.1145/3360567

[26] A. Cheung, A. Solar-Lezama, and S. Madden. 2013. Optimizing Database-Backed Applications with Query Synthesis. In *PLDI*. https://doi.org/10.1145/2491956.2462180

[27] B. Churchill, O. Padon, R. Sharma, and A. Aiken. 2019. Semantic Program Alignment for Equivalence Checking. In *PLDI*. https://doi.org/10.1145/3314221.3314596

[28] K. Claessen and J. Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *ICFP*.

[29] E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. 2014. VeriMAP: A Tool for Verifying Programs through Transformations. In *TACAS*. https://doi.org/10.1007/978-3-642-54862-8_47

[30] E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. 2016. Relational Verification Through Horn Clause Transformation. In *SAS*. https://doi.org/10.1007/978-3-662-53413-7_8

[31] L. de Moura and N. Bjørner. 2007. Efficient E-Matching for SMT Solvers. In *CADE*.

[32] S. Dutta, D. Sarkar, A. Rawat, and K. Singh. 2016. Validation of Loop Parallelization and Loop Vectorization Transformations. In *ENASE*. https://doi.org/10.5220/0005869501950202

[33] S. Eguchi, N. Kobayashi, and T. Tsukada. 2018. Automated Synthesis of Functional Programs with Auxiliary Functions. In *APLAS*. https://doi.org/10.1007/978-3-030-02768-1_13

[34] K. Ellis, L. Morales, M. Sablé-Meyer, A. Solar-Lezama, and J. B. Tenenbaum. 2018. Library Learning for Neurally-Guided Bayesian Program Induction. In *NeurIPS*, Vol. 31.

[35] K. Ellis, C. Wong, M. Nye, M. Sablé-Meyer, L. Morales, L. Hewitt, L. Cary, A. Solar-Lezama, and J. B. Tenenbaum. 2021. DreamCoder: Bootstrapping Inductive Program Synthesis with Wake-Sleep Library Learning. In *PLDI*.

[36] A. Farzan, D. Lette, and V. Nicolet. 2022. Recursion Synthesis with Unrealizability Witnesses. In *PLDI*. https://doi.org/10.1145/3519939.3523726

[37] A. Farzan and V. Nicolet. 2021. Counterexample-Guided Partial Bounding for Recursive Function Synthesis. In *CAV*.

[38] P. Feautrier. 1996. Automatic Parallelization in the Polytope Model. In *The Data Parallel Programming Model*. https://doi.org/10.1007/3-540-61736-1_44

[39] Y. Feng, R. Martins, Y. Wang, I. Dillig, and T. W. Reps. 2017. Component-Based Synthesis for Complex APIs. In *POPL*. https://doi.org/10.1145/3009837.3009851

[40] J. K. Feser, S. Chaudhuri, and I. Dillig. 2015. Synthesizing Data Structure Transformations from Input-Output Examples. In *PLDI*. https://doi.org/10.1145/2737924.2737977

[41] Y. Futamura. 1971. Partial Evaluation of Computation Process–An Approach to a Compiler-Compiler. In *Systems, Computers, Controls*.

[42] W. D. Goldfarb. 1981. The Undecidability of the Second-Order Unification Problem. In *Theoretical Computer Science*. https://doi.org/10.1016/0304-3975(81)90040-2

[43] P. Graf. 1995. Substitution Tree Indexing. In *RTA*. https://doi.org/10.1007/3-540-59200-8_52

[44] P. Graf and C. Meyer. 1996. Advanced Indexing Operations on Substitution Trees. In *CADE*.

[45] S. Gulwani. 2011. Automating String Processing in Spreadsheets Using Input-Output Examples. In *POPL*.

[46] Z. Guo, M. James, D. Justo, J. Zhou, Z. Wang, R. Jhala, and N. Polikarpova. 2019. Program Synthesis by Type-Guided Abstraction Refinement. In *POPL*. https://doi.org/10.1145/3371080

[47] M. Gupta, S. Mukhopadhyay, and N. Sinha. 2000. Automatic Parallelization of Recursive Procedures. In *International Journal of Parallel Programming*. https://doi.org/10.1023/A:1007560600904

[48] S. Gupta, A. Saxena, A. Mahajan, and S. Bansal. 2018. Effective Use of SMT Solvers for Program Equivalence Checking Through Invariant-Sketching and Query-Decomposition. In *SAT*. https://doi.org/10.1007/978-3-319-94144-8_22

[49] J. V. Guttag and J. J. Horning. 1978. The Algebraic Specification of Abstract Data Types. In *Programming Methodology*.

[50] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, et al. 2020. Array programming with NumPy. In *Nature*. https://doi.org/10.1038/s41586-020-2649-2

[51] M. Hofmann and E. Kitzelmann. 2010. I/O Guided Detection of List Catamorphisms: Towards Problem Specific Use of Program Templates in IP. In *PEPM*. https://doi.org/10.1145/1706356.1706375

[52] G. P. Huet. 1975. A Unification Algorithm for Typed λ-Calculus. In *Theoretical Computer Science*.

[53] M. B. James, Z. Guo, Z. Wang, S. Doshi, H. Peleg, R. Jhala, and N. Polikarpova. 2020. Digging for Fold: Synthesis-Aided API Discovery for Haskell. In *OOPSLA*. https://doi.org/10.1145/3428273

[54] J. Jeon, X. Qiu, J. S. Foster, and A. Solar-Lezama. 2015. JSketch: Sketching for Java. In *ESEC/FSE*. https://doi.org/10.1145/2786805.2803189
[55] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. 2010. Oracle-Guided Component-Based Program Synthesis. In *ICSE*. https://doi.org/10.1145/1806799.1806833
[56] N. D. Jones, P. Sestoft, and H. Søndergaard. 1989. Mix: A Self-Applicable Partial Evaluator for Experiments in Compiler Generation. In *LISP and Symbolic Computation*. https://doi.org/10.1007/BF01806312
[57] S. P. Jones and S. Graf. 2023. Triemaps that match. arXiv:2302.08775 [cs.PL]
[58] S. Kamil, A. Cheung, S. Itzhaky, and A. Solar-Lezama. 2016. Verified Lifting of Stencil Computations. In *PLDI*. https://doi.org/10.1145/2908080.2908117
[59] T. Knoth, D. Wang, N. Polikarpova, and J. Hoffmann. 2019. Resource-Guided Program Synthesis. In *PLDI*. https://doi.org/10.1145/3314221.3314602
[60] D. Kocetkov, R. Li, L. B. Allal, J. Li, C. Mou, C. M. Ferrandis, Y. Jernite, M. Mitchell, S. Hughes, T. Wolf, D. Bahdanau, L. von Werra, and H. de Vries. 2022. The Stack: 3 TB of permissively licensed source code. arXiv:2211.15533 [cs.CL]
[61] J. Koppel, Z. Guo, E. de Vries, A. Solar-Lezama, and N. Polikarpova. 2022. Searching Entangled Program Spaces. In *ICFP*. https://doi.org/10.1145/3547622
[62] D. Kozen. 1997. Kleene Algebra with Tests. In *TOPLAS*. https://doi.org/10.1145/256167.256195
[63] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan. 2007. Effective Automatic Parallelization of Stencil Computations. In *PLDI*. https://doi.org/10.1145/1250734.1250761
[64] S. Laddad, C. Power, M. Milano, A. Cheung, and J. M. Hellerstein. 2022. Katara: Synthesizing CRDTs with Verified Lifting. In *OOPSLA*. https://doi.org/10.1145/3563336
[65] M. Lange, E. Lozes, and M. Vargas Guzmán. 2014. Model-Checking Process Equivalences. In *Theoretical Computer Science*. https://doi.org/10.1016/j.tcs.2014.08.020
[66] W. Lee and H. Cho. 2023. Inductive Synthesis of Structurally Recursive Functional Programs from Non-recursive Expressions. In *POPL*. https://doi.org/10.1145/3571263
[67] W. Lee, K. Heo, R. Alur, and M. Naik. 2018. Accelerating Search-Based Program Synthesis Using Learned Probabilistic Models. In *PLDI*. https://doi.org/10.1145/3192366.3192410
[68] Tomer Libal and A. Steen. 2016. Towards a Substitution Tree Based Index for Higher-order Resolution Theorem Provers. In *PAAR*. https://hal.science/hal-01424749
[69] N. P. Lopes and J. Monteiro. 2016. Automatic Equivalence Checking of Programs with Uninterpreted Functions and Integer Arithmetic. In *STTT*. https://doi.org/10.1007/s10009-015-0366-1
[70] J. Lubin and S. E. Chasins. 2021. How Statically-Typed Functional Programmers Write Code. In *OOPSLA*. https://doi.org/10.1145/3485532
[71] J. Lubin, N. Collins, C. Omar, and R. Chugh. 2020. Program Sketching with Live Bidirectional Evaluation. In *ICFP*. https://doi.org/10.1145/3408991
[72] J. Lubin, J. Ferguson, K. Ye, J. Yim, and S. E. Chasins. 2024. *Cobbler Virtual Machine*. https://doi.org/10.5281/zenodo.10802503
[73] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. 2005. Jungloid Mining: Helping to Navigate the API Jungle. In *PLDI*.
[74] B. Mariano, Y. Chen, Y. Feng, G. Durrett, and I. Dillig. 2022. Automated Transpilation of Imperative to Functional Code Using Neural-Guided Program Synthesis. In *OOPSLA*. https://doi.org/10.1145/3527315
[75] B. Mariano, J. Reese, S. Xu, T. Nguyen, X. Qiu, J. S. Foster, and A. Solar-Lezama. 2019. Program Synthesis with Algebraic Library Specifications. In *OOPSLA*. https://doi.org/10.1145/3360558
[76] E. Meijer, M. Fokkinga, and R. Paterson. 1991. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In *FPCA*. https://doi.org/10.1007/3540543961_7
[77] N. Meng, M. Kim, and K. S. McKinley. 2013. Lase: Locating and Applying Systematic Edits by Learning from Examples. In *ICSE*. https://doi.org/10.1109/ICSE.2013.6606596
[78] A. K. Menon, O. Tamuz, S. Gulwani, B. Lampson, and A. T. Kalai. 2013. A Machine Learning Framework for Programming by Example. In *ICML*.
[79] A. Miltner, A. T. Nuñez, A. Brendel, S. Chaudhuri, and I. Dillig. 2022. Bottom-up Synthesis of Recursive Functional Programs Using Angelic Execution. In *POPL*. https://doi.org/10.1145/3498682
[80] N. Mitchell. [n. d.]. *hlint: Source code suggestions*. https://hackage.haskell.org/package/hlint. Accessed: 2024-03-19.
[81] D. Mordvinov and G. Fedyukovich. 2017. Synchronizing Constrained Horn Clauses. In *LPAR*.
[82] N. Mulleners, J. Jeuring, and B. Heeren. 2023. Program Synthesis Using Example Propagation. In *PADL*.
[83] D. Nam, B. Ray, S. Kim, X. Qu, and S. Chandra. 2022. Predictive Synthesis of API-centric Code. In *MAPS*.
[84] C. Nandi, M. Willsey, A. Anderson, J. R. Wilcox, E. Darulova, D. Grossman, and Z. Tatlock. 2020. Synthesizing Structured CAD Models with Equality Saturation and Inverse Transformations. In *PLDI*. https://doi.org/10.1145/3385412.3386012
[85] C. Nandi, M. Willsey, A. Zhu, Y. R. Wang, B. Saiki, A. Anderson, A. Schulz, D. Grossman, and Z. Tatlock. 2021. Rewrite Rule Inference Using Equality Saturation. In *OOPSLA*. https://doi.org/10.1145/3485496

[86] T. Nipkow. 1993. Functional Unification of Higher-Order Patterns. In *LICS*. https://doi.org/10.1109/LICS.1993.287599
[87] U. Norell. 2007. *Towards a Practical Programming Language Based on Dependent Type Theory*. Ph. D. Dissertation. Chalmers University of Technology.
[88] P.-M. Osera and S. Zdancewic. 2015. Type-and-Example-Directed Program Synthesis. In *PLDI*. https://doi.org/10.1145/2737924.2738007
[89] L. C. Paulson. 1986. Natural Deduction as Higher-Order Resolution. In *Journal of Logic Programming*.
[90] P. M. Phothilimthana, T. Jelvis, R. Shah, N. Totla, S. Chasins, and R. Bodik. 2014. Chlorophyll: Synthesis-Aided Compiler for Low-Power Spatial Architectures. In *PLDI*. https://doi.org/10.1145/2594291.2594339
[91] P. M. Phothilimthana, A. Thakur, R. Bodik, and D. Dhurjati. 2016. GreenThumb: Superoptimizer Construction Framework. In *CC*. https://doi.org/10.1145/2892208.2892233
[92] P. M. Phothilimthana, A. Thakur, R. Bodik, and D. Dhurjati. 2016. Scaling up Superoptimization. In *ASPLOS*.
[93] B. Pientka. 2009. Higher-Order Term Indexing Using Substitution Trees. In *TOCL*.
[94] N. Polikarpova, I. Kuraj, and A. Solar-Lezama. 2016. Program Synthesis from Polymorphic Refinement Types. In *PLDI*. https://doi.org/10.1145/2908080.2908093
[95] M. Raza, S. Gulwani, and N. Milic-Frayling. 2014. Programming by Example Using Least General Generalizations. In *AAAI Conference on Artificial Intelligence*. https://doi.org/10.1609/aaai.v28i1.8744
[96] R. Rolim, G. Soares, L. D'Antoni, O. Polozov, S. Gulwani, R. Gheyi, R. Suzuki, and B. Hartmann. 2017. Learning Syntactic Program Transformations from Examples. In *ICSE*. https://doi.org/10.1109/ICSE.2017.44
[97] R. Rugina and M. Rinard. 1999. Automatic Parallelization of Divide and Conquer Algorithms. In *PPoPP*.
[98] E. Schkufza, R. Sharma, and A. Aiken. 2013. Stochastic Superoptimization. In *ASPLOS*.
[99] R. Sharma, E. Schkufza, B. Churchill, and A. Aiken. 2013. Data-Driven Equivalence Checking. In *OOPSLA*.
[100] K. C. Shashidhar, M. Bruynooghe, F. Catthoor, and G. Janssens. 2005. Functional Equivalence Checking for Verification of Algebraic Transformations on Array-Intensive Source Code. In *DATE*. https://doi.org/10.1109/DATE.2005.163
[101] K. Shi, D. Bieber, and R. Singh. 2022. TF-Coder: Program Synthesis for Tensor Manipulations. In *TOPLAS*.
[102] C. Smith and A. Albarghouthi. 2016. MapReduce Program Synthesis. In *PLDI*. https://doi.org/10.1145/2908080.2908102
[103] C. Smith and A. Albarghouthi. 2019. Program Synthesis with Equivalence Reduction. In *VMCAI*. https://doi.org/10.1007/978-3-030-11245-5_2
[104] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat. 2006. Combinatorial Sketching for Finite Programs. In *ASPLOS*. https://doi.org/10.1145/1168857.1168907
[105] M. Sousa and I. Dillig. 2016. Cartesian Hoare Logic for Verifying K-Safety Properties. In *PLDI*. https://doi.org/10.1145/2908080.2908092
[106] J. Swan, K. Krawiec, and Z. A. Kocsis. 2019. Stochastic Synthesis of Recursive Functions Made Easy with Bananas, Lenses, Envelopes and Barbed Wire. In *Genetic Programming and Evolvable Machines*.
[107] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner. 2009. Equality Saturation: A New Approach to Optimization. In *POPL*. https://doi.org/10.1145/1480881.1480915
[108] The Coq Development Team. 2022. *The Coq Proof Assistant*. https://doi.org/10.5281/zenodo.7313584
[109] The Pandas Development Team. 2020. *Pandas*. https://doi.org/10.5281/zenodo.3509134
[110] F. Theiss and C. Benzmüller. 2006. Term Indexing for the LEO-II Prover. In *IWIL*.
[111] E. Torlak and D. Jackson. 2007. Kodkod: A Relational Model Finder. In *TACAS*.
[112] H. Unno, T. Terauchi, and E. Koskinen. 2021. Constraint-Based Relational Verification. In *CAV*.
[113] S. Verdoolaege, G. Janssens, and M. Bruynooghe. 2009. Equivalence Checking of Static Affine Programs Using Widening to Handle Recurrences. In *CAV*. https://doi.org/10.1007/978-3-642-02658-4_44
[114] Verse Lab. [n. d.]. *Ego*. https://verse-lab.github.io/ego/ego/index.html. Accessed: 2023-11-14.
[115] P. Wadler. 1988. Deforestation: Transforming Programs to Eliminate Trees. In *Theoretical Computer Science*. https://doi.org/10.1016/0304-3975(90)90147-A
[116] Y. R. Wang, S. Hutchison, J. Leang, B. Howe, and D. Suciu. 2020. SPORES: Sum-Product Optimization via Relational Equality Saturation for Large Scale Linear Algebra. In *Proceedings of the VLDB Endowment*.
[117] M. Willsey, C. Nandi, Y. R. Wang, O. Flatt, Z. Tatlock, and P. Panchekha. 2021. egg: Fast and Extensible Equality Saturation. In *POPL*. https://doi.org/10.1145/3434304
[118] H. Yang. 2007. Relational Separation Logic. In *Theoretical Computer Science*. https://doi.org/10.1016/j.tcs.2006.12.036
[119] Y. Yang, P. M. Phothilimtha, Y. R. Wang, M. Willsey, S. Roy, and J. Pienaar. 2021. Equality Saturation for Tensor Graph Superoptimization. In *MLSys*.
[120] Y. Yuan, A. Radhakrishna, and R. Samanta. 2023. Trace-Guided Inductive Synthesis of Recursive Functional Programs. In *PLDI*. https://doi.org/10.1145/3591255
[121] M. Zalewski. [n. d.]. *American Fuzzy Lop*. https://lcamtuf.coredump.cx/afl/. Accessed: 2023-07-06.

Received 2023-11-16; accepted 2024-03-31