# Program Synthesis with Equivalence Reduction

Calvin Smith(✉) and Aws Albarghouthi

University of Wisconsin–Madison, Madison, WI, USA
cjsmith@cs.wisc.edu

**Abstract.** We introduce *program synthesis with equivalence reduction*, a synthesis methodology that utilizes relational specifications over components of a given synthesis domain to reduce the search space. Leveraging a blend of classic and modern techniques from term rewriting, we use relational specifications to discover a canonical representative per equivalence class of programs. We show how to design synthesis procedures that only consider programs in *normal form*, thus pruning the search space. We discuss how to implement equivalence reduction using efficient data structures, and demonstrate the significant reductions it can achieve in synthesis time.

## 1 Introduction

Over the past few years, we have witnessed great strides in automated program synthesis, the process of automatic construction of programs that satisfy a given specification—for instance, a logical formula [3], an input-output example [16,24], a type [17], etc. While the underlying algorithmic techniques may appear different, ultimately, a majority of existing algorithms and tools implement a search through the space of programs, be it explicitly through careful enumeration or implicitly through constraint solving.

Of course, the search space in synthesis is enormous—likely infinite. But whenever we are encountered with a large search space, it is often the case that large fractions of the space are redundant. Here, we ask the following question: *How can we exploit operator semantics to efficiently explore large spaces of candidate programs?*

**Motivation.** Let us consider a generic learner–teacher model, where the learner (the synthesizer) proposes programs and the teacher (the verifier) answers with *yes*/*no*, indicating whether the learner has provided the correct program or not. Our goal is to make the learner *smarter*: we want to reduce the number of questions the learner needs to ask before arriving at the right answer.

Consider the following two string-manipulating programs:

$$p_1 : \ \lambda x.\, \texttt{swap}(\texttt{lower}(x)) \qquad\qquad p_2 : \ \lambda x.\, \texttt{upper}(x)$$

**Electronic supplementary material** The online version of this chapter (https://doi.org/10.1007/978-3-030-11245-5_2) contains supplementary material, which is available to authorized users.


C. Enea and R. Piskac (Eds.): VMCAI 2019, LNCS 11388, pp. 24–47, 2019.
https://doi.org/10.1007/978-3-030-11245-5_2

where `swap` turns all uppercase characters to lowercase, and vice versa; `lower` and `upper` turn all characters into lowercase or uppercase, respectively. A smart learner would know that turning all characters into lowercase and then applying `swap` is the same as simply applying `upper`. Therefore, the learner would only inquire about one of the programs $p_1$ and $p_2$. Formally, the learner knows the following piece of information connecting the three functions:

$$\forall x.\, \texttt{swap}(\texttt{lower}(x)) = \texttt{upper}(x)$$

One could also imagine a variety of other semantic knowledge that a learner can leverage, such as properties of specific functions (e.g., idempotence) or relational properties over combinations of functions (e.g., distributivity). Such properties can be supplied by the developer of the synthesis domain, or discovered automatically using tools like QuickSpec [6] or Bach [37].

**Equivalence Reduction.** Universally quantified formulas like the one above form *equational specifications* (equations, for short): they define some (but not all) of the behaviors of the *components* (functions in the synthesis domain), as well as relations between them. The equations partition the space of programs into equivalence classes, where each equivalence class contains all equivalent programs with respect to the equations. The learner needs to detect when two programs are in the same equivalence class and only ask the teacher about one *representative* per equivalence class. To do so, we make the observation that we can utilize the equations to define a *normal form* on programs, where programs within the same equivalence class all simplify to the same normal form. By structuring the learner to only consider programs in normal form, we ensure that no redundant programs are explored, potentially creating drastic reductions in the search space. We call this process *program synthesis with equivalence reduction*.

By constraining specifications to be equational (as in the above example), we can leverage standard *completion algorithms*, e.g., Knuth–Bendix completion [21], to construct a *term-rewriting system* (TRS) that is *confluent*, *terminating*, and *equivalent* to the set of equations. Effectively, the result of completion is a *decision procedure* that checks whether a program $p$ is the representative of its equivalence class—i.e., whether $p$ is in normal form. The difficulty, however, is that constructing such a decision procedure is an *undecidable process*—as equations are rich enough to encode a Turing machine. Nonetheless, significant progress has been made in completion algorithms and *termination proving* (e.g., [15,41,43]), which is used for completion.

Given a normalizing TRS resulting from completion, we show how to incorporate it in existing synthesis techniques in order to prune away redundant fragments of the search space and accelerate synthesis. We show how to incorporate equivalence reduction into salient synthesis algorithms that employ *bottom-up*, dynamic-programming-style search—e.g., [2,3,28]—and *top-down* search—e.g., [13,14,30,33].

**Our primary technical contribution is porting foundational techniques from term rewriting and theorem proving to a contemporary automated program synthesis setting.**

**Applicability.** While our proposed technique is general and orthogonal to much of the progress in program synthesis technology, it is important to note that it is not a panacea. For instance, a number of synthesizers, e.g., the enumerative SyGuS solver [3], prune the search space using *observational equivalence* with respect to a set of input–output examples, which effectively impose a coarse over-approximation of the true equivalence relation on programs. In such settings, equivalence reduction can be beneficial when, for instance, (*i*) evaluating examples is expensive, e.g., if one has to compile the program, simulate it, evaluate a large number of examples; or (*ii*) the verification procedure does not produce counterexamples, e.g., if we are synthesizing separation logic invariants, and one cannot prune through observational equivalence.

Our approach is beneficial in synthesis settings where observational equivalence is not an option or is difficult to incorporate, e.g., in functional program synthesis algorithms like $\lambda^2$ [13], Myth [14,30], SynQuid [33], Leon [20], and BIGλ [36]. A number of these tools employ a top-down type-driven search with which observational equivalence is not compatible. Additionally, some of these techniques decompose the problem into multiple subproblems, e.g., a process searching for *mappers* and another searching for *reducers* in BIGλ. In such case, different synthesis subproblems have no *input context* on which to employ observational equivalence. Thus, minimizing the search space is essential.

**Contributions.** This paper makes a number of contributions:

- ***Conceptual.*** We present *program synthesis with equivalence reduction*, where a synthesis problem is augmented with domain knowledge in the form of *equational specifications*.
- ***Algorithmic.*** We demonstrate how to utilize classical and modern techniques from theorem proving and the theory of TRSs to impose a normal form on programs. We demonstrate how to incorporate normal forms in bottom-up and top-down synthesis techniques.
- ***Practical.*** We implement our approach in an existing synthesis tool for functional, data-parallel programs. To fully exploit equivalence reduction, we discuss the importance of employing efficient data structures used by theorem provers—namely, *perfect discrimination trees* [27]—and fast algorithms for normality checking.
- ***Empirical.*** We apply our tool to synthesis of *reduction* functions—commutative and associative binary operators that are ubiquitous in modern data-parallel programming. Our thorough empirical evaluation investigates the following important aspects:
  - Speedups gained with equivalence reduction.
  - Overhead of applying equivalence reduction in different algorithms, in relation to program size.
  - Robustness of equivalence reduction to varying the number of equations used.
  - The impact of data structures (perfect discrimination trees) on efficiency.

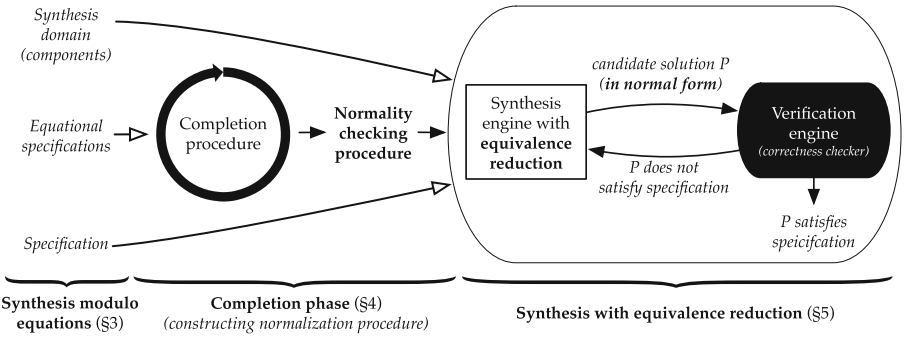



**Fig. 1.** Overview of synthesis with equivalence reduction

## 2 Overview and Illustration

### 2.1 Overview

Figure 1 provides an overview of our proposed synthesis technique. A *synthesis modulo equations problem* is defined by three inputs. First, we are given a *synthesis domain*, which is a set of components (operators) that define the search space of programs. Second, we expect *equational specifications*, which are equations over single components or combinations of components. For example, equations might specify that an operator $f$ is associative, or that two operators, $f$ and $g$, are inverses of each other. Finally, a synthesis problem also contains a *specification* of the desired program. Below, we describe the various components in Fig. 1 in detail.

### 2.2 Synthesis Modulo Equations Problem

**Synthesis Domain.** We will now illustrate the various parts of our approach using a simple example. Consider the synthesis domain shown in Table 1(a). The domain includes basic integer operations as well as a number of functions over strings and *byte arrays* (`utf8`) that form a subset of Python 3.6's string API.[1] We describe some of the non-standard components. `split(x,y)` splits string `x` into a list of strings using the *delimiter* string $y$, e.g.:

```
split("hizvmcaiz19","z") = ["hi", "vmcai", "19"]
```

The function `join(x,y)` concatenates a list of strings `x` using the delimiter string `y`. Functions `encode`/`decode` transform between strings and UTF-8 byte arrays.

**Equational Specifications.** Even for such a simple synthesis domain, there is a considerable amount of latent domain knowledge that we can exploit in the synthesis process. Table 1(b) provides a partial view of the equations that we can

[1] https://docs.python.org/3/library/stdtypes.html.

**Table 1.** (a) Left: synthesis domain; (b) Right: partial list of equations

| Component name | Description |
|---|---|
| *Integers* | |
| $+ : \texttt{int} \to \texttt{int} \to \texttt{int}$ | integer addition |
| $- : \texttt{int} \to \texttt{int} \to \texttt{int}$ | integer subtraction |
| $* : \texttt{int} \to \texttt{int} \to \texttt{int}$ | integer multiplication |
| $\texttt{abs} : \texttt{int} \to \texttt{int}$ | absolute value |
| *Strings and byte arrays* | |
| $+\!+ : \texttt{str} \to \texttt{str} \to \texttt{str}$ | str concatenation |
| $\texttt{len} : \texttt{str} \to \texttt{int}$ | str length |
| $\texttt{swap} : \texttt{str} \to \texttt{str}$ | swap upper/lowercase |
| $\texttt{split} : \texttt{str} \to \texttt{str} \to [\texttt{str}]$ | split str w/ delimiter |
| $\texttt{join} : [\texttt{str}] \to \texttt{str} \to \texttt{str}$ | concat. list w/ delimiter |
| $\texttt{encode} : \texttt{str} \to \texttt{utf8}$ | encode str as UTF-8 |
| $\texttt{decode} : \texttt{utf8} \to \texttt{str}$ | decode UTF-8 into str |

| Equational specifications |
|---|
| $x + y = y + x$ |
| $(x + y) + z = x + (y + z)$ |
| $x * (y + z) = (x * y) + (x * z)$ |
| $\texttt{abs}(\texttt{abs}(x)) = \texttt{abs}(x)$ |
| ... |
| $\texttt{len}(x +\!+ y) = \texttt{len}(x) + \texttt{len}(y)$ |
| $\texttt{swap}(\texttt{swap}(x)) = x$ |
| $\texttt{join}(\texttt{split}(x, y), y) = x$ |
| $\texttt{decode}(\texttt{encode}(x)) = x$ |
| $\texttt{encode}(\texttt{decode}(x)) = x$ |
| ... |

utilize for this synthesis domain. The variables $x, y, z$ are implicitly universally quantified. Consider, for instance, the following equation:

$$\forall x, y.\, \texttt{join}(\texttt{split}(x, y), y) = x$$

This connects `split` and `join`: splitting a string $x$ with delimiter $y$, and then joining the result using the same delimiter $y$, produces the string $x$. In other words, `split` and `join` are inverses, assuming a fixed delimiter $y$.

Other equations specify, e.g., that `abs` (absolute value of an integer) is idempotent ($\forall x.\, \texttt{abs}(\texttt{abs}(x)) = \texttt{abs}(x)$) or that the function `swap` is an *involution*—an inverse of itself ($\forall x.\, \texttt{swap}(\texttt{swap}(x)) = x$).

### 2.3 Completion Phase

**Completion Overview.** Two programs are equivalent with respect to the equations if we can use the equations to rewrite one into the other—just as a high-school student would apply trigonometric identities to make the two sides of a trigonometric equation identical. Given the set of equations, we would like to be able to partition the space of programs into equivalence classes, where two programs are in the same equivalence class if and only if they are equivalent with respect to the equations. By partitioning the space into equivalence classes, we can ensure that we only consider one representative program per equivalence class. Intuitively, without equations, each program is its own equivalence class. The more equations we add—i.e., the more domain knowledge we have—the larger our equivalence classes are.

Given the set of equations, the completion phase generates a TRS that transforms any program into its normal form—the representative of its equivalence class. It is important to note that the process of determining equality modulo equations is generally undecidable [29], since equations are rich enough to encode transitions of a Turing machine. Completion attempts to generate a decision procedure for equality modulo equations, and as such can fail to terminate.

Nevertheless, advances in automatic termination proving have resulted in powerful completion tools (e.g., [41,43]). Note that completion is a one-time phase for a given synthesis domain, and therefore can be employed offline, not affecting synthesis performance.

**The Term Rewriting System.** The TRS generated by completion is a set of *rewrite rules* of the form $l \to r$, which specify that if a (sub)program matches the *pattern* $l$, then it can be transformed using the pattern $r$. For instance, completion of the equations in our running example might result in a system that includes the rule $\texttt{swap}(\texttt{swap}(x)) \to x$. In other words, for any program containing the pattern $\texttt{swap}(\texttt{swap}(x))$, where $x$ is a variable indicating any completion of the program, we can rewrite it into $x$.

The above rule appears like a simple syntactic transformation (*orientation*) of the corresponding equation defining that `swap` is an involution. However, as soon as we get to slightly more complex equations, the resulting rules can become intricate. Consider, for instance, commutativity of addition. The completion procedure will generate an *ordered* rewrite system to deal with such *unorientable* rules. For example, one rule that completion might generate is $x + y \to_{>} y + x$, which specifies that a program of the form $x + y$ can be rewritten into $y + x$ only if $x + y > y + x$, where $>$ is a *reduction ordering*, which is a *well-founded* ordering on programs. (The difficulty in completion is finding a reduction order, just like finding a *ranking function* is the key for proving program termination.)

**Normality Checking.** Given the TRS generated by the completion procedure, checking whether a program $p$ is in normal formal is a simple process: If any of the rewrite rules in the TRS can be applied to $p$, then we know that the program is not in normal form, since it can be reduced.

### 2.4 Synthesis with Equivalence Reduction

Let us now discuss how a synthesis procedure might utilize the TRS generated by completion to prune the search space. For the sake of illustration, suppose our synthesis technique constructs programs in a bottom-up fashion, by combining small programs to generate larger programs, a strategy that is employed by a number of recent synthesis algorithms [2,3,28].

Consider the following simple program,

```
λs, count. len(s ++ "012") + count
```

where `s` is a string variable and `count` is an integer variable. The synthesizer constructs this program by applying integer addition to the two smaller expressions: `len(s ++ "012")` and `count`. To check if the program is in normal form, the synthesizer attempts to apply all the rules in the TRS generated by completion. If none of the rules apply, the program is *irreducible*, or in normal form. If any rule applies, then we know that the program is not in normal form. In the latter case, we can completely discard this program from the search space. *But what if the end solution uses this program as a subprogram?* By construction of the TRS, if a program $p$ is not in normal form, then all programs $p_s$, where $p$ appears in $p_s$ as a subprogram, are also *not* in normal form. Intuitively, we can apply the same rewrite rules to $p_s$ as those we can apply to $p$.

By ensuring that we only construct and maintain programs in normal form, we drastically prune the search space. Figure 2 shows the number of well-typed programs for fixed program size in our running synthesis domain, augmented with two integer and two string variables. The solid (blue) line shows the number of programs (normal forms) for increasing size of the abstract syntax tree (components and variables appearing in the program). When we include the equations in Table 1(b) that only deal with integer components, the number of programs per size shrinks, as shown by the dashed (green) line. Incorporating the full set of equations (over integer and string components) shrinks the number of normal forms further, as shown by the dotted (red) line. For instance, at 11 AST nodes, there are 21 million syntactically distinct programs, but only about 20% of them are in normal form with respect to the full set of equations.

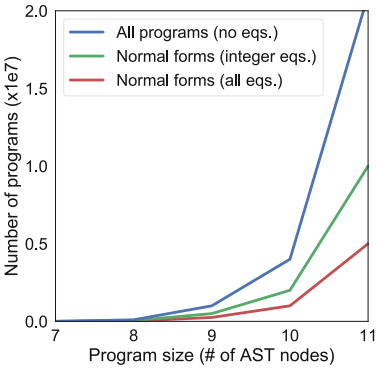



**Fig. 2.** #normal forms vs. prog. size (Color figure online)

While the number of programs explodes as we increase the size (unless the synthesis domain is fairly simple), utilizing the equations allows us to delay the explosion and peer deeper into the space of programs. In Sect. 5, we experimentally demonstrate the utility of equations on practical synthesis applications.

## 3 Synthesis Modulo Equations

We now define synthesis problems with equational specifications.

### 3.1 Formalizing the Synthesis Problem

**Synthesis Domain.** A *synthesis domain* $\mathcal{D}$ is a set of *components* $\{f_1, \ldots, f_n\}$, where each component $f_i$ is a function of arity $ar(f_i) \in \mathbb{N}$. The synthesis domain $\mathcal{D}$ induces a set of *candidate programs* $\mathcal{P}_\mathcal{D}$, where each $p \in \mathcal{P}_\mathcal{D}$ is defined as follows:

$$\begin{aligned} p &:= f && f \in \mathcal{D} \text{ and } ar(f) = 0 \\ &\mid f(p_1, \ldots, p_n) && f \in \mathcal{D} \text{ and } ar(f) = n > 0 \end{aligned}$$

When clear from context, we shall use $\mathcal{P}$ to refer to $\mathcal{P}_\mathcal{D}$. Components of arity $n$ model functions that take $n$ arguments and return some value; components of arity 0 model constants and input arguments of a program. For simplicity of presentation, we shall restrict our discussion to first-order components and elide types. While our approach can handle higher-order components, the equations we define below are restricted to first-order components.

**Synthesis Problems.** A *synthesis problem* $S$ is a tuple $(\mathcal{D}, \varphi)$, where $\mathcal{D}$ is a synthesis domain and $\varphi$ is a *specification*. A *solution* to a synthesis problem $S$ is a program $p \in \mathcal{P}_\mathcal{D}$ such that $p \models \varphi$, where $p \models \varphi$ specifies that the program $p$ satisfies the specification $\varphi$. We assume that $\varphi$ is defined abstractly—it can be a Hoare triple that $p$ should satisfy, a reference implementation that $p$ should be equivalent to, a set of input–output examples $p$ should satisfy, etc.

**Synthesis Modulo Equations Problems.** A *synthesis modulo equations problem* $S^\star$ is a tuple $(\mathcal{D}, \varphi, \mathcal{E})$, where $\mathcal{E}$ defines *equational specifications*. Formally, $\mathcal{E}$ is a set of *equations*, where each equation is a pair $(p_1, p_2) \in \mathcal{P}_\mathcal{D}(X) \times \mathcal{P}_\mathcal{D}(X)$ and $\mathcal{P}_\mathcal{D}(X)$ is the set of programs induced by the domain $\mathcal{D} \cup X$, where $X = \{x, y, z, \ldots\}$ is a special set of variables. An equation $(p_1, p_2)$ denotes the universally quantified formula $\forall X.\, p_1 = p_2$, indicating that programs $p_1$ and $p_2$ are semantically equivalent for any substitution of the variables $X$.

*Example 1 (Matrix operations).* Suppose that the synthesis domain is defined as follows: $\mathcal{D} = \{t, +_m, i\}$, where $t$ is a unary function that returns the transpose of a matrix, $+_m$ is (infix) matrix addition, and $i$ denotes an input argument. A possible set $\mathcal{E}$ is:

$$t(t(x)) = x \tag{$s_1$}$$

$$t(x +_m y) = t(x) +_m t(y) \tag{$s_2$}$$

where $x$ and $y$ are from the set of variables $X$. Formula $s_1$ specifies that transposing a matrix twice returns the same matrix; Formula $s_2$ specifies that transposition distributes over matrix addition. Using $\mathcal{E}$, we can infer that the following programs are semantically equivalent:

$$t(t(i) +_m t(i)) \quad =_{s_2} \quad t(t(i)) +_m t(t(i)) \quad =_{s_1} \quad i +_m i$$

**Equivalence Reduction.** Given a synthesis problem $S^\star$, the equations $\mathcal{E}$ induce an equivalence relation on candidate programs in $\mathcal{P}$. We shall use $p_1 =_\mathcal{E} p_2$ to denote that two programs are *equivalent modulo* $\mathcal{E}$ (formally defined in Sect. 3.2). We can partition the set of candidate programs $\mathcal{P}$ into a union of disjoint equivalence classes, $\mathcal{P} = P_1 \uplus P_2 \uplus \ldots$, where for all $p, p' \in \mathcal{P}$,

$$p =_\mathcal{E} p' \iff (\exists i \in \mathbb{N} \text{ such that } p, p' \in P_i)$$

For each equivalence class $P_i$, we shall designate a single program $p_i \in P_i$, called the representative of $P_i$. A program $p \in \mathcal{P}$ is in *normal form*, denoted $norm(p)$, *iff* it is a representative of some equivalence class $P_i$.

**Solutions of Synthesis Modulo Equations Problems.** A solution to a synthesis problem $S^\star = (\mathcal{D}, \varphi, \mathcal{E})$ is a program $p \in \mathcal{P}$ such that (1) $p \models \varphi$ and (2) $norm(p)$ holds. That is, a solution to the synthesis problem is in normal form.

### 3.2 Term-Rewriting and Completion

We now ground our discussion in the theory of term rewriting systems and discuss using *completion* to transform our equations into a procedure that detects

if a program is in normal form. We refer to Baader and Nipkow [4] for a formal exposition of term-rewriting systems.

**Rewrite Rules.** A *rewrite system* $R$ is a set of *rewrite rules* of the form $(l, r) \in \mathcal{P}_{\mathcal{D}}(X) \times \mathcal{P}_{\mathcal{D}}(X)$, with $vars(r) \subseteq vars(l)$. We will denote a rewrite rule $(l, r)$ as $l \to r$. These rules induce a *rewrite relation*. We say that $p$ rewrites to $p'$, written as $p \to_R p'$, *iff* there exists a rule $l \to r \in R$ that can transform $p$ to $p'$. We illustrate rewrite rules with an example.

*Example 2.* Consider the following rewrite rule, $f(x, x) \to g(x)$, where $f$ and $g$ are elements of $\mathcal{D}$ and $x$ is a variable. Consider the program $p = f(f(a, a), b)$, where $a$ and $b$ are two arguments. We can apply the rewrite rule to rewrite $p$ into $p' = f(g(a), b)$, by rewriting the subprogram $f(a, a)$ into $g(a)$.

We will use $\to_R^*$ to denote the reflexive transitive closure of the rewrite relation. The symmetric closure of $\to_R^*$, denoted $\leftrightarrow_R^*$, forms an equivalence relation. We shall drop the subscript $R$ when the TRS is clear from context.

**Normal Forms.** For a given TRS $R$, a program $p$ is *R-irreducible iff* there is no program $p'$ such that $p \to_R p'$. For a program $p$, the set of $R$-irreducible programs reachable from $p$ via $\to_R$ is its set of *normal forms*. We write $N_R(p) = \{p' \mid p \to^* p',\ p' \text{ is } R\text{-irreducible}\}$ for the normal forms of $p$.

We say that a TRS $R$ is *normalizing iff* for every program $p$, $|N_R(p)| \geqslant 1$. A TRS $R$ is *terminating iff* the relation $\to_R$ is *well-founded*; that is, for every program $p$, there exists $n \in \mathbb{N}$ such that there is no $p'$ where $p \to_R^n p'$ (i.e., no $p'$ reachable from $p$ through $n$ rewrites).

**Rewrite Rules and Equations.** Recall that equations are of the form $(p_1, p_2) \in \mathcal{P}_{\mathcal{D}}(X) \times \mathcal{P}_{\mathcal{D}}(X)$. It is often convenient to view an equation $(p_1, p_2)$ as two rules: $p_1 \to p_2$ and $p_2 \to p_1$. Let $R$ be the TRS defined by equations in $\mathcal{E}$, then for all programs $p, p' \in \mathcal{P}_{\mathcal{D}}(X)$, we have $p \leftrightarrow_R^* p' \iff p =_{\mathcal{E}} p'$.

$R$ is not terminating by construction, and so cannot be used for determining unique normal forms. For a terminating TRS equivalent to $\mathcal{E}$, we must be more cautious with how rules are generated. The process of generating these rules is known as a *completion procedure*.

**Completion Procedures.** For our purposes, we only need a declarative view of completion procedures. A completion procedure provides a term rewriting system $R_c$ such that $p \leftrightarrow_{R_c}^* p' \iff p =_{\mathcal{E}} p'$ and for any program $p$, applying the rules in $R_c$ will always lead to a unique normal form in finitely many rewrites, no matter what the order of application is. Formally, $R_c$ is *terminating and confluent*.

Completion is generally undecidable. Knuth and Bendix are responsible for the first completion procedure [21]; it repeatedly tries to *orient* equations—turn them into rewrite rules—through syntactic transformations. Knuth–Bendix completion, even if it terminates, can still fail to produce a result, as not all equations are orientable. Bachmair et al. neatly side-step this weakness by presenting a completion procedure that cannot fail, called *unfailing completion* [5]. In order to handle the unorientable rules, unfailing completion introduces *ordered rules*:

(a) BOTTOM-UP: Bottom-up synthesis

$$\frac{}{C \leftarrow \emptyset}\ \text{INIT} \qquad \frac{p \in C \qquad p \models \varphi}{p \text{ is a solution}}\ \text{VERIFY}$$

$$\frac{f \in \mathcal{D} \qquad \{p_1, \ldots, p_n\} \subseteq C \qquad p = f(p_1, \ldots, p_n) \qquad norm(p)}{C \leftarrow C \cup \{p\}}\ \text{EXPAND}$$

(b) TOP-DOWN: Top-down synthesis

$$\frac{}{C \leftarrow \{\bullet\}}\ \text{INIT} \qquad \frac{p \in C \qquad p \models \varphi}{p \text{ is a solution}}\ \text{VERIFY}$$

$$\frac{p \in C \qquad norm(\sigma p) \qquad \bullet \in vars(p) \qquad f \in \mathcal{D} \qquad \sigma = [\bullet \mapsto f(\bullet_1, \ldots, \bullet_n)] \qquad \{\bullet_i\}_i \text{ are fresh}}{C \leftarrow C \cup \{\sigma p\}}\ \text{EXPAND}$$

**Fig. 3.** Synthesis with equivalence reduction algorithms

let $>$ be a *reduction order*, and $r : u \to_> v$ be an ordered rule. (A reduction order is a *well-founded* order that ensures termination of the rewrite system.) Then $p_1 \to p_2$ by rule $r$ *iff* $p_1 \to p_2$ by the unordered rule $u \to v$ and $p_1 > p_2$.

Recall our matrix domain $\mathcal{D} = \{t, +_m, inp\}$ from Example 1, and suppose we have the equation $x +_m y = y +_m x$. Knuth–Bendix completion will not be able to orient this rule. Unfailing completion, when provided with a suitable reduction order $>$, would generate the ordered rule $x +_m y \to_> y +_m x$. Modern completion tools, such as omkbTT [43] and Slothrop [41], are able to simultaneously complete a set of rules and derive an appropriate reduction order.

**Knuth–Bendix Order.** The *Knuth–Bendix order* (KBO) is a standard family of reduction orders that we will use in our implementation and evaluation. The formal definition of KBO is not important for our exposition, and we thus relegate it to the supplementary material. We will denote a KBO as $>_{\text{KBO}}$, and note that naïvely computing KBO following its standard definition is polynomial in the size of the compared terms. We discuss our linear-time implementation in Sect. 5.3.

## 4 Synthesis Modulo Equations

We now describe how to incorporate equivalence reduction in *bottom-up* and *top-down* synthesis techniques, and highlight the subtleties involved. An example illustrating both scenarios is provided in the supplementary material.

*Bottom-up techniques* explore the space of programs in a bottom-up, dynamic-programming fashion, building larger programs from smaller ones. Examples include Escher [2], the enumerative solver of SyGuS [3], and the probabilistic search of Menon et al. [28].

*Top-down techniques* explore the space of programs in a top-down fashion, effectively, by unrolling the grammar defining the programs. A number of recent synthesis algorithms, particularly for functional programs, employ this methodology, e.g, Myth [30], Myth2 [14], Big$\lambda$ [36], $\lambda^2$ [13], and SynQuid [33].

We now present abstract algorithms for these techniques and show how to augment them with equivalence reduction.

### 4.1 Bottom-Up Synthesis Modulo Equations

We start by describing the bottom-up synthesis algorithm. We would like to find a solution to the synthesis problem $S^\star = (\mathcal{D}, \varphi, \mathcal{E})$. We assume that completion has resulted in a procedure $norm(p)$ that checks whether a candidate program $p$ is in normal form.

Figure 3(a) shows a bottom-up synthesis algorithm, BOTTOM-UP, as a set of guarded rules that can be applied non-deterministically. The only state maintained is a set $C$ of explored programs, which is initialized to the empty set in the initialization rule INIT. The algorithm terminates whenever the rule VERIFY applies, in which case a program satisfying the specification $\varphi$ is found.

The rule EXPAND creates a new program $p$ by applying an $n$-ary function $f$ to $n$ programs from the set $C$. Observe, however, that $p$ is only considered if it is in normal form. In other words, the algorithm maintains the invariant that all programs in $C$ are in normal form.

**Root-Normality.** The invariant that all programs in $C$ are normal can be used to simplify checking $norm(p)$ during the EXPAND step. In synthesizing $p = f(p_1, \ldots, p_n)$, we already know that the subprograms $p_1, \ldots, p_n$ are normal: no rule can apply to any subprogram. Therefore, if $p$ is not normal, it must be due to a rule applying at the root. Checking this property, called *root-normality*, simplifies rule application. Instead of examining all subprogram decompositions of $p$ to see if the rule $l \to r$ applies, it suffices to check whether there exists a substitution $\sigma$ such that $\sigma p = \sigma l$.

### 4.2 Top-Down Synthesis Modulo Equations

We now describe how to perform top-down synthesis with equivalence reduction. Top-down synthesis builds programs by unrolling the grammar of programs. We will assume that we have a countable set of variables $X = \{\bullet, \bullet_1, \bullet_2, \ldots\}$, called *wildcards*, which we use as placeholders for extending programs in $\mathcal{P}_\mathcal{D}(X)$.

Figure 3(b) shows the top-down synthesis algorithm, TOP-DOWN, a simplified version of the algorithm in BIG$\lambda$ [36]. The algorithm maintains a set $C$ of ground (with wildcards) and non-ground programs. $C$ is initialized to the program $\bullet$, using INIT. The rule EXPAND picks a non-ground program from $C$ and substitutes

one of its wildcards with a new program. The algorithm terminates when a ground program in $C$ satisfies the specification, as per rule VERIFY.

**Normality with Non-ground Programs.** The rule EXPAND checks whether $p$ is in normal form before adding it to $C$. However, note that TOP-DOWN maintains non-ground programs in $C$, and even if a non-ground program is normal, no ground programs derivable from it through EXPAND need be normal. Therefore, TOP-DOWN may end up exploring subtrees of the search space that are redundant. Deciding if a non-ground program has ground instances in normal form is known as checking *R-ground reducibility*, which is decidable in exponential time [7]. Our formulation avoids exponential checks at the cost of exploring redundant subtrees.

Soundness of both algorithms is discussed in the supplementary material.

## 5 Implementation and Evaluation

### 5.1 Implementation and Key Optimizations

We implemented our technique in an existing efficient synthesis tool, written in OCaml, that employs bottom-up and top-down search strategies. Our tool accepts a domain $\mathcal{D}$, defined as typed OCaml functions, along with a set of equations $\mathcal{E}$ over the OCaml functions. As a specification $\varphi$ for the synthesis problem, we utilize input–output examples (see Sect. 5.2 below).

The implementations of the bottom-up and top-down synthesis algorithms augment the abstract algorithms in Sect. 4 with a deterministic search strategy that utilizes types. Both algorithms explore programs by increasing size—a strategy used in many existing synthesis tools, e.g., [2,13,30], as smaller programs are considered more likely to *generalize*. Both algorithms are type-directed, enumerating only well-typed programs.

**Implementing Completion and Reduction Orders.** Completions of equations were found using the omkbTT tool [43]—which employs termination provers for completion. All completions used the KBO reduction order (see the supplementary material).

During synthesis, the reduction order can be a performance bottleneck, as we need to compute it for every candidate program. If we were to implement KBO directly from its formal definition (see the supplementary material), evaluating $s >_{\text{KBO}} t$ would be quadratic in $|s| + |t|$. However, program transformation techniques have given us an algorithm linear in the sizes of the terms [26]. In our tool, we implement Löchner's linear-time KBO computation algorithm. The performance impacts of the reduction order will be discussed in Sect. 5.3.

**Data Structures for Normalization.** Every time a candidate program is considered, we check if it is in normal form using $norm(\cdot)$ (recall algorithms in Fig. 3). More precisely, given a candidate program $p$, *norm* attempts to find a substitution $\sigma$ and a rule $l \to r \in R$ such that $\sigma(l) = p$. This is a *generalization* problem, which has been studied for years in the field of automated theorem

proving. A naïve implementation of *norm* might keep a list of rules in the TRS, and match candidate programs against one rule at a time. Instead, we borrow from the existing literature and use *perfect discrimination trees* [27] to represent our list of rules. Perfect discrimination trees are used in the Waldmeister theorem prover [18] to great effect; *the tree representation lets us match multiple rules at once, and ignore rules that are inapplicable.*

A perfect discrimination tree can be thought of as a *trie*. Figure 4 illustrates the construction for a set of unordered rules (ordered rules can be added analogously). First, rules are rewritten using De Bruijn-like indices [9]. Second, the left-hand side of every rule is converted into a string through a pre-order traversal. Finally, all string representations are inserted into the trie.

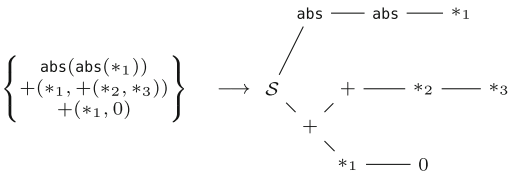
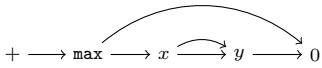



**Fig. 4.** Building the trie data structure from lhs of rules

To match a candidate program $p$ against the trie, we first convert $p$ to a *flatterm*, which is a linked-list representation of $p$ in pre-order with forward pointers to jump over subterms. For example, the term $+(\texttt{max}(x, y), 0)$ is converted to:




Now, matching the program against the trie is done using a simple backtracking algorithm, which returns a substitution (if one exists) that converts the left-hand side of a rule in our set to the query program. See [27] for details.

Using perfect discrimination trees in our normalization procedure has several immediate benefits, the most important of which is that unused rules do not impact the performance, as their paths are never followed. In Sect. 5.3, we will evaluate the performance overhead of normalization.

### 5.2 Synthesis Domain and Benchmarks

A primary inspiration for our work came from applying synthesis to the domain of large-scale, data-parallel programming, where a program is composed of data-parallel combinators, e.g., *map* and *reduce*, which allow programmers to write distributed programs without having to worry about low-level details of distribution. Popular *MapReduce-like* systems, e.g., Apache Spark [45], Hadoop [42], and Dryad [44], provide such high-level interfaces.

Here, we will focus on synthesizing *reducers* in the distributed, data-parallel programming context. Reducers are functions that allow us to *aggregate* large amounts of data by exploiting parallelism. Our long-term goal with synthesis of such aggregation functions from examples is to enable average computer users to construct non-trivial data analyses through examples. We will focus on evaluating our synthesis algorithms in this context.

```
let add-c q1 q2 =
  let real = (fst q1) + (fst q2) in
  let imaginary = (snd q1) + (snd q2) in
  pair real imaginary
```

**Fig. 5.** Addition of two complex numbers of the form $a + bi$, where $a$ and $b$ are represented as a `pair`

To synthesize deterministic data-parallel programs, tools like Big$\lambda$ ensure that reducers form a *commutative semigroup* (CSG) [36]. This guarantees determinism in the face of data reordering (e.g., *shuffles* [10]). To ensure we only synthesize CSG reducers, we employ the dynamic analysis technique from Big$\lambda$ [36].

**Synthesis Domain.** Our synthesis domain comprises four primary sets of components, each consisting of 10+ components, that focus on different types. These types—integers, tuples, strings, and lists—are standard, and appear as the subject of many synthesis works. See full list in the supplementary material.

**Equational Specifications.** We manually gathered a set of 50 equations for our synthesis domain. Each class of components has between 3 (lists) and 21 (integers) equations, with a few equations correlating functions over multiple domains (e.g., strings and integers interacting through `length`). Completions of the equations are a mix of ordered and unordered rules describing the interaction of the components. Some equations are described below—full list in the supplementary material.

- *Strings*: In addition to the equations relating `uppercase`, `swap`, and `lowercase` (as defined in Sect. 1), we include equations encoding, e.g., idempotence of `trim`, and the fact that many string operations distribute over concatenation. For instance, we have the equation $\forall x, y.\, \texttt{len}(x) + \texttt{len}(y) = \texttt{len}(x +\!\!+\, y)$.
- *Lists*: We provide equations specifying that operations distribute over list concatenation, as in $\forall x, y. \texttt{sum}(x) + \texttt{sum}(y) = \texttt{sum}(\texttt{cat}(x, y))$. In addition, we relate constructors/destructors, as in $\forall x, y. \texttt{head}(\texttt{cons}(x, y)) = x$.

**Benchmarks.** Our benchmarks were selected to model common reducers over our domain, and typically require solutions with 10–12 AST nodes—large enough to be a challenge for state-of-the-art synthesizers, as we see later in this Sect. 5.3. A few examples are given below—for a full list, refer to supplementary material.

- *Tuples and integers*: The tuple benchmarks expose several different uses for pairs in reducers—as an encoding for rational numbers (such as in `mult-q`), for complex numbers (in `add-c`), and for points on the plane (as in `distances`). We also treat pairs as intervals over integers (e.g., `intervals` synthesizes *join*

in the lattice of intervals [8]). Figure 5 shows the synthesized program for one of those benchmarks.
– *Lists and integers*: Lists are also an interesting target for aggregation, e.g., if we are aggregating values from different scientific experiments, where each item is a list of readings from one sensor. List benchmarks compute a value from two lists and emit the result as a singleton list. For example, `ls-sum-abs` computes absolute value of the sums of two lists, and then adds the two, returning the value as a singleton list.

Like many synthesis tools, we use input–output examples to characterize the desired solution. Examples are used to ensure that the solution (*i*) matches user expectations and (*ii*) forms a CSG.

### 5.3 Experimental Evaluation

Our experiments investigate the following questions:

**RQ1.** Does equivalence reduction increase the efficiency of synthesis algorithms on the domain described above?
**RQ2.** What is the overhead of equivalence reduction?
**RQ3.** How does the performance change with different numbers of equations?
**RQ4.** Are the data structures used in theorem provers a good fit for synthesis?

To address these questions, we developed a set of 30 synthesis benchmarks. Each benchmark consists of: (*i*) a specification, in the form of input–output examples (typically no more than 4 examples are sufficient to fully specify the solution); (*ii*) a set of components from the appropriate domain; (*iii*) a set of ordered and unordered rewrite rules generated from equations over the provided components.

For each algorithm, bottom-up (BU) and top-down (TD), we created three variations:

– BU and TD: equivalence reduction disabled.
– $\text{BU}_n$ and $\text{TD}_n$: equivalence reduction enabled.
– $\text{BU}_{\widetilde{n}}$ and $\text{TD}_{\widetilde{n}}$: equivalence reduction without ordered rules. By dropping ordered rules from the generated TRS, we get more normal forms (less pruning).

See Table 2 for the full results. For each experiment, we measure total time taken in seconds. Grey boxes indicate the best-in-category strategy for each benchmark—e.g., the winner of the `sub-c` benchmark is $\text{BU}_n$ in the bottom-up category, and $\text{TD}_{\widetilde{n}}$ in top-down. Values reported are the average across 10 runs.

**RQ1: Effects of Equivalence Reduction on Performance.** In 2 out of the 3 benchmarks where BU and TD do not terminate, adding equivalence reduction allows the synthesizer to find a solution in the allotted time. For bottom-up, in all benchmarks where BU terminates in under 1 s, both $\text{BU}_n$ and $\text{BU}_{\widetilde{n}}$ outperform the naïve BU, often quite dramatically: in `sum-to-second`, BU takes over 60 s, while $\text{BU}_n$ and $\text{BU}_{\widetilde{n}}$ finish in under 2 s. For top-down, $\text{TD}_{\widetilde{n}}$ outperforms TD in

**Table 2.** Experimental results (Mac OS X 10.11; 4 GHz Intel Core i7; 16 GB RAM). We impose a CPU timeout of 300 s and a memory limit of 10 GBs per benchmark. ✗ denotes a timeout.

| | Bottom-up variations | | | Top-down variations | | | Other tools | |
|---|---|---|---|---|---|---|---|---|
| **Benchmark** | BU | $\text{BU}_{\widetilde{n}}$ | $\text{BU}_n$ | TD | $\text{TD}_{\widetilde{n}}$ | $\text{TD}_n$ | $\lambda^2$ | SQ |
| *Integers* | | | | | | | | |
| `add` | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.01 | 0.18 |
| `max` | 0.01 | 0.01 | 0.01 | 0.03 | 0.05 | 0.02 | 0.4 | 0.6 |
| `min` | 0.01 | 0.01 | 0.01 | 0.05 | 0.02 | 0.04 | 0.01 | 0.62 |
| *Tuples & integers* | | | | | | | | |
| `add-4` | 0.05 | 0.05 | 0.11 | 0.12 | 0.14 | 1.29 | 5.35 | 8.86 |
| `mult-q` | 7.38 | 0.48 | 0.44 | 15.63 | 1.95 | 3.03 | ✗ | ✗ |
| `div-q` | 7.38 | 0.48 | 0.44 | 14.22 | 2.12 | 3.82 | ✗ | ✗ |
| `add-c` | 7.39 | 0.48 | 0.44 | 13.51 | 3.68 | 8.11 | ✗ | ✗ |
| `sub-c` | 7.35 | 0.48 | 0.43 | 31.63 | 4.51 | 9.28 | ✗ | ✗ |
| `add-q-long` | ✗ | 44.65 | 49.51 | ✗ | 57.43 | 95.55 | ✗ | ✗ |
| `max-pair` | 32.79 | 4.02 | 4.92 | 40.25 | 21.53 | 56.21 | ✗ | ✗ |
| `intervals` | 32.76 | 4.00 | 4.92 | 74.77 | 25.25 | 21.80 | ✗ | ✗ |
| `min-pair` | 32.72 | 4.03 | 4.95 | 81.88 | 19.55 | 67.49 | ✗ | ✗ |
| `sum-to-first` | 52.74 | 1.18 | 0.62 | 110.35 | 5.47 | 15.06 | ✗ | ✗ |
| `sum-to-second` | 68.93 | 1.51 | 0.70 | 107.88 | 5.27 | 11.22 | ✗ | ✗ |
| `add-and-mult` | 7.39 | 0.48 | 0.45 | 26.12 | 1.51 | 9.34 | ✗ | ✗ |
| `distances` | ✗ | 7.36 | 5.58 | ✗ | 50.31 | 100.66 | ✗ | ✗ |
| *Strings & integers* | | | | | | | | |
| `str-len` | 1.40 | 0.42 | 0.52 | 4.47 | 1.58 | 2.06 | NA | NA |
| `str-trim-len` | 26.29 | 6.79 | 7.14 | 219.50 | 52.21 | 218.61 | NA | NA |
| `str-upper-len` | 5.70 | 1.78 | 1.81 | 26.93 | 5.64 | 8.01 | NA | NA |
| `str-lower-len` | 3.86 | 1.23 | 1.26 | 22.48 | 6.93 | 4.63 | NA | NA |
| `str-add` | 0.05 | 0.02 | 0.03 | 0.26 | 0.08 | 0.07 | NA | NA |
| `str-mult` | 0.05 | 0.02 | 0.03 | 0.17 | 0.05 | 0.06 | NA | NA |
| `str-max` | 1.79 | 0.45 | 0.54 | 8.10 | 1.32 | 2.99 | NA | NA |
| `str-split` | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | NA | NA |
| *Lists & integers* | | | | | | | | |
| `ls-sum` | 0.00 | 0.02 | 0.03 | 0.01 | 0.02 | 0.11 | 0.01 | 10.43 |
| `ls-sum2` | 147.91 | 88.33 | 107.02 | 229.66 | ✗ | 254.87 | ✗ | ✗ |
| `ls-sum-abs` | 0.08 | 0.07 | 0.10 | 0.22 | 0.19 | 0.37 | 63.00 | ✗ |
| `ls-min` | 0.00 | 0.02 | 0.03 | 0.01 | 0.02 | 0.10 | 0.01 | NA |
| `ls-max` | 0.00 | 0.02 | 0.03 | 0.01 | 0.02 | 0.10 | 0.01 | NA |
| `ls-stutter` | 27.68 | 5.12 | 8.01 | 50.42 | 15.90 | 93.84 | ✗ | NA |

nearly all benchmarks that take TD more than 1 s (the exception being `ls-sum2`). With ordered rules, the exceptions are more numerous. The most egregious is `ls-stutter`, going from 50s with TD to 94 s with $\text{TD}_n$. There is still potential for large performance gains: in `sum-to-second`, we decrease the time from 108 s in TD to under 12 s for $\text{TD}_n$ and under 6 s for $\text{TD}_{\widetilde{n}}$.

**Equivalence Reduction Appears to Drastically Improve the Performance of Bottom-Up and Top-Down Synthesis.** In general, the unordered rules outperform the full ordered rules. In the bottom-up case, this performance gap is smaller than 5s: while the ordered rules are more costly to check, bottom-up synthesis only requires that we check them at the root of a program. In top-down, we must check rule application at all sub-programs. This magnifies the cost of the ordered rules and leads to significant performance differences between $\text{TD}_n$ and $\text{TD}_{\widetilde{n}}$.

**RQ2-a: Overhead of Equivalence Reduction.** Figure 6 provides a different look at the benchmarks in Table 2: for each benchmark where BU and TD do not terminate in less that 1 s, we compute (*i*) the *overhead*, the percentage of time spent in the normalization procedure *norm*; and (*ii*) the *reduction*, the percentage of programs visited compared to the un-normalized equivalent, BU or TD. The results are shown as density plots.

Figure 6a and c show the performance characteristics of $\text{BU}_{\widetilde{n}}$ and $\text{TD}_{\widetilde{n}}$, respectively. Both have consistent overhead—40% for $\text{BU}_{\widetilde{n}}$ and 25% for $\text{TD}_{\widetilde{n}}$—although $\text{TD}_{\widetilde{n}}$ has a more reliable reduction of over 85%, while $\text{BU}_{\widetilde{n}}$ ranges from 60% to 90% reduction. Both strategies boast large reductions in the number of candidate programs visited for reasonable overhead, although $\text{TD}_{\widetilde{n}}$ is the clear winner—$\text{BU}_{\widetilde{n}}$ dominates $\text{TD}_{\widetilde{n}}$ in Table 2, suggesting that normalization isn't enough to fully close the gap between BU and TD. In Fig. 6b and d, we see the performance characteristics of $\text{BU}_n$ and $\text{TD}_n$, respectively. Compared to Fig. 6a and c, we see a higher overhead with less consistent normalization. Both figures have secondary clusters of benchmarks outside the region of highest density: these contain the benchmarks from the strings and integers domain.

**This View of the Data Supports the Conclusion of Table 2 that Unordered Rules Outperform Ordered Rules.** While our implementation of KBO is optimized, evaluating the reduction order is still a bottleneck. Our implementation verifies candidate solutions quickly, but **the benefits of high reduction outweigh the large overhead as verification time increases.** For instance, when providing more input-output examples, the verification time increases but not the overhead. In the `ls-stutter` benchmark, $\text{BU}_{\widetilde{n}}$ visits 1,288,565 programs with an average overhead of 1.07 s, while $\text{BU}_n$ visits 792,662 programs with an average overhead of 5.6 s. Increasing the verification cost per program by only 0.0001 s will raise $\text{BU}_{\widetilde{n}}$'s time by 129 s, while $\text{BU}_n$'s time is only raised by 80 s—easily a large enough gap to out-scale the overhead. Indeed, when we instrument our tool with a synthetic delay, this behavior is visible.

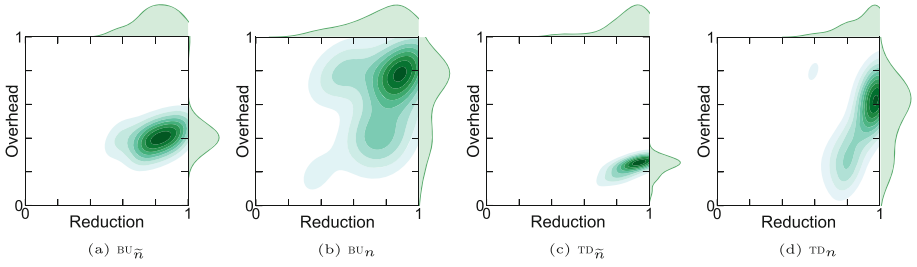
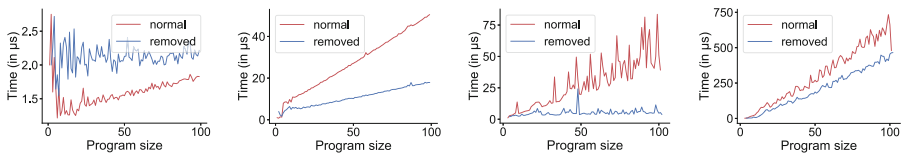



**Fig. 6.** Equivalence reduction overhead. Benchmarks are converted into (overhead, reduction) pairs and plotted using kernel density estimation (KDE), with marginal distributions projected on to the side. No points lie outside the bounding box (any appearance of such is an artifact of KDE).

**RQ2-b: Normalization Overhead w.r.t. Program Size.** Experience holds that normalization procedures don't scale as candidate programs become large. To explore how this behavior might impact the effectiveness of equivalence reduction, we instrumented our tool to ignore solutions and explore the space of programs depth-first, during which we record the average overhead of $norm(\cdot)$ at all program sizes. Figure 7 presents the data for the `sum-to-first` benchmark, although the figures are representative of the other benchmarks.




(a) $\text{BU}_{\widetilde{n}}$ (b) $\text{BU}_{n}$ (c) $\text{TD}_{\widetilde{n}}$ (d) $\text{TD}_{n}$

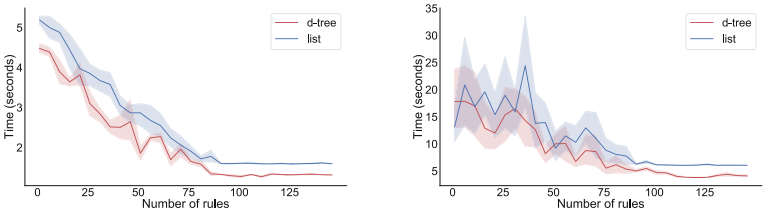
**Fig. 7.** Average performance of $norm(\cdot)$ w.r.t. the size of candidate programs. *Normal* graph represents executions of $norm(\cdot)$ that return `true`; *removed* represents executions that return `false`. Data is average of multiple executions of $norm(\cdot)$ per program size using the `sum-to-first` benchmark. Time is in microseconds—note the difference in scale between graphs.

**Unsurprisingly, $norm(\cdot)$ Scales Linearly with Program Size. This Linear Growth Appears Quite Sustainable.** Solutions with 100 AST nodes are beyond modern-day synthesis tools, and a 3x slowdown compared to programs of size 40 is manageable.

When we compare the performance of $\text{BU}_{\widetilde{n}}$ in Fig. 7a to that of $\text{BU}_{n}$ in Fig. 7b, we observe an order of magnitude loss in performance. This holds as well for $\text{TD}_{\widetilde{n}}$ and $\text{TD}_{n}$ in Fig. 7c and d, respectively. Checking KBO is clearly expensive, and so the observed performance in Table 2 of $\text{BU}_{n}$ and $\text{TD}_{n}$ indicate a large amount of search-space reduction occurring.

**RQ3 and RQ4: Impact of Rules and Perfect Discrimination Trees.** To determine how the number of rules impacts our tool's performance, we completed our entire set of 50 equations to produce 83 unordered rules that we randomly sample subsets from (the results from ordered results are similar). To test the effectiveness of perfect discrimination trees, we compare performance against a naïve algorithm that maintains a list of rules it checks against one by one on a representative benchmark: `str-len`. Not all rules apply to the components used—only 47 out of 83 describe components used for `str-len`. We plot the time taken for synthesis per number of randomly sampled rules, from 0 rules to 150 rules (to clearly show optimal performance). Results are presented in Fig. 8.

We see, for both benchmarks, nearly continuously decreasing graphs; the only exceptions are with low numbers of rules sampled, where it is likely we have mostly unusable rules. The performance levels off at 83 rules, when we are guaranteed to sample all applicable rules. These results are promising: completion




(a) str-len with $\text{BU}_{\widetilde{n}}$ (b) str-len with $\text{TD}_{\widetilde{n}}$

**Fig. 8.** Performance versus number of rules sampled for `d-tree` and `list` over 2 benchmarks. The line is the average of 10 samples per $x$-value, and the lighter band is a 95% confidence interval.

is undecidable, and so it is impossible to predict the rules that will be included from a given set of equations. However, the results in Fig. 8 indicate that—on average—**the more rules we provide the better the algorithm's performance, even when the rules might not be relevant**. Furthermore, we see immediately and clearly that **perfect discrimination trees outperform our list-based implementation.** Performance differences are magnified in the $\text{TD}_{\widetilde{n}}$ benchmarks, where checking normality includes checks on every subterm. On the rest of the benchmarks, the naïve implementation results in an average of an 11% increase in time for $\text{BU}_{\widetilde{n}}$ and a 144% increase for $\text{TD}_{\widetilde{n}}$, which strongly indicates that perfect discrimination trees are an important implementation choice.

**Gauging Benchmark Difficulty.** We considered related tools as a gauge of benchmark difficulty and a baseline for evaluation. The most similar tool—$\lambda^2$ [13]—is top-down, type-directed, uses input–output examples, and searches for programs from smallest to largest. SynQuid (SQ) [33] synthesizes Haskell programs from refinement types, using SMT-driven type-directed synthesis. When able, we encoded specifications of our benchmarks as refinement types.[2]

As seen in Table 2, $\lambda^2$ is either not applicable (strings are not supported, and so were ignored) or unable to solve most benchmarks. SQ exhibits similar behavior and performance. We stress that these results are meant as a indication of the difficulty of the benchmarks, and not a head-to-head comparison between our algorithms and those of $\lambda^2$ and SQ.

**Threats to Validity.** We identify two primary threats to the validity of our evaluation. First, we base our evaluation on a single tool in order to evaluate various algorithms and data structures. However, since our bottom-up and top-down strategies are (*i*) instances of standard synthesis techniques and (*ii*) comparable to existing implementations (as seen in Table 2), we believe our results can be beneficial to tools like Myth, SynQuid, and $\lambda^2$, modulo technical details.

[2] We also consider two other works: Big$\lambda$ [36] is implemented in Python and not competitive with our baseline, while Myth [30] expects data types to be specified from first principles, and does not have, e.g., integers or strings by default.

Second, the domains considered in our evaluation—integers, lists, etc.—operate over well-behaved algebraic structures. These domains form the core search space of many modern synthesis tools, but one could imagine domains that do not induce many equational specifications, e.g., GUI manipulation and stateful domains.

### 5.4 Further Discussion

**Constructing Equational Specifications.** In a large number of recent works on program synthesis, it is assumed that someone designs the synthesis domain by providing a set of components. We additionally assume that we are given a set of equational specifications over the components. In our evaluation, we manually crafted a set of equations for our domain. Alternatively, this process can be automated using tools like QuickSpec [6] and Bach [37].

**Rule Preprocessing.** The synthesis algorithms we consider search for a program over a regular tree grammar of components. Therefore, one could incorporate equations by rewriting the grammar so as to only generate normal forms. This can be done by encoding the TRS as a regular tree grammar and intersecting it with the search grammar. However, to express a TRS as a regular tree grammar, we require the TRS to be left-linear and unordered [31]. These conditions are too strong to be used as a general technique: most useful equations result in non-left-linear or ordered rules.

**Completion and Termination.** A key component in our approach is the completion tool that takes our equations and produces a TRS that can be used for pruning the search space. In our evaluation, we found that modern completion procedures were able to complete our equational specifications. In general, however, completion is an undecidable problem. In the supplementary material, we discuss a mechanism to work around this fact, by terminating the completion procedure at any point and salvaging a sub-optimal (non-confluent) TRS.

## 6 Related Work

**Program Synthesis.** We are not the first to use normal forms for pruning in synthesis. In type-directed synthesis, Osera and Zdancewic [30] and Frankle et al. [14] restrict the space by only traversing programs in *β-normal form*. Equivalence reduction can be used to augment such techniques with further pruning, by exploiting the semantics of the abstract data types defined. Feser et al. [13] mention that their enumeration uses a fixed set of standard rewrites, e.g., $x + 0 \rightarrow x$, to avoid generating redundant expressions. In contrast, our work presents a general methodology for incorporating equational systems into the search by exploiting completion algorithms.

Techniques that search for fast programs—e.g., *superoptimization* [32,35]—may not be able to directly benefit from equivalence reduction, as it may impose

inefficient normal forms. It would be interesting to incorporate a cost model into completion and coerce it into producing minimal-cost normal forms.

In SyGuS [3,39], the synthesizer generates a program encodable in a decidable first-order theory and equivalent to some logical specification. A number of solvers in this category employ a *counter-example-guided synthesis loop* (CEGIS) [38]: they prune the search space using a set of input–output examples, which impose a coarse over-approximation of the true equivalence relation on programs. In the CEGIS setting, equivalence reduction can be beneficial when, for instance, (*i*) evaluating a program to check if it satisfies the examples is expensive, e.g., if one has to compile the program, simulate it, evaluate a large number of examples; or (*ii*) the verification procedure does not produce counterexamples, e.g., if we are synthesizing separation logic invariants.

A number of works sample programs from a probabilistic grammar that imposes a probability distribution on programs [11,25,28]. It would be interesting to investigate incorporating equivalence reduction in that context, for instance, by *truncating* the distribution so as to only sample irreducible programs.

Recently, Wang et al. [40] introduced SYNGAR, where abstract transition relations are provided for each component of a synthesis domain. The synthesis algorithm over-approximates equivalence classes by treating two programs equivalent if they are equivalent in the abstract semantics. The abstraction is refined when incorrect programs are found.

**Completion and Term-Rewriting Systems.** A number of classic works [12, 34] used completion procedures to transform an equational specification into a program—a terminating rewrite system. Our setting is different: we use completion in order to prune the search space in modern inductive synthesis tools.

Kurihara and Kondo's multi-completion [23] sidesteps the issue of picking a reduction order by allowing completion procedures to consider a class of reduction orders simultaneously. Klein and Hirokawa's maximal completion algorithm [19] takes advantage of SMT encodings of reduction orders (such as Zankl et al.'s KBO encoding [46]) to reduce completion to a series of MAXSMT problems in which the parameters of the reduction order are left free. Completion tools like omkbTT [43] and SLOTHROP [41], rely on external termination provers [1,22].

**Acknowledgement.** This work is supported by the National Science Foundation CCF under awards 1566015 and 1652140.

## References

1. Alarcón, B., Gutiérrez, R., Lucas, S., Navarro-Marset, R.: Proving termination properties with MU-TERM. In: Johnson, M., Pavlovic, D. (eds.) AMAST 2010. LNCS, vol. 6486, pp. 201–208. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-17796-5_12
2. Albarghouthi, A., Gulwani, S., Kincaid, Z.: Recursive program synthesis. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 934–950. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_67

3. Alur, R., et al.: Syntax-guided synthesis. In: FMCAD (2013)
4. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press, Cambridge (1998)
5. Bachmair, L., Dershowitz, N., Plaisted, D.A.: Completion without failure. Resolut. Eqn. Algebraic Struct. **2**, 1–30 (1989)
6. Claessen, K., Smallbone, N., Hughes, J.: QuickSpec: guessing formal specifications using testing. In: Fraser, G., Gargantini, A. (eds.) TAP 2010. LNCS, vol. 6143, pp. 6–21. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-13977-2_3
7. Comon, H., Jacquemard, F.: Ground reducibility is EXPTIME-complete. In: Proceedings of the 12th Annual IEEE Symposium on Logic in Computer Science, LICS 1997, pp. 26–34. IEEE (1997)
8. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, pp. 238–252. ACM (1977)
9. De Bruijn, N.G.: Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. In: Indagationes Mathematicae, Proceedings, vol. 75, pp. 381–392. Elsevier (1972)
10. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. In: OSDI (2004)
11. Dechter, E., Malmaud, J., Adams, R.P., Tenenbaum, J.B.: Bootstrap learning via modular concept discovery. In: IJCAI (2013)
12. Dershowitz, N.: Synthesis by completion. Urbana **51**, 61801 (1985)
13. Feser, J.K., Chaudhuri, S., Dillig, I.: Synthesizing data structure transformations from input-output examples. In: PLDI (2015)
14. Frankle, J., Osera, P.M., Walker, D., Zdancewic, S.: Example-directed synthesis: a type-theoretic interpretation. In: POPL (2016)
15. Giesl, J., Schneider-Kamp, P., Thiemann, R.: AProVE 1.2: automatic termination proofs in the dependency pair framework. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS, vol. 4130, pp. 281–286. Springer, Heidelberg (2006). https://doi.org/10.1007/11814771_24
16. Gulwani, S., Harris, W.R., Singh, R.: Spreadsheet data manipulation using examples. CACM **55**(8), 97–105 (2012)
17. Gvero, T., Kuncak, V., Kuraj, I., Piskac, R.: Complete completion using types and weights. In: PLDI (2013)
18. Hillenbrand, T., Buch, A., Vogt, R., Löchner, B.: Waldmeister-high-performance equational deduction. J. Autom. Reason. **18**(2), 265–270 (1997)
19. Klein, D., Hirokawa, N.: Maximal completion. In: LIPIcs-Leibniz International Proceedings in Informatics, vol. 10. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2011)
20. Kneuss, E., Kuraj, I., Kuncak, V., Suter, P.: Synthesis modulo recursive functions. In: OOPSLA (2013)
21. Knuth, D.E., Bendix, P.B.: Simple word problems in universal algebras. In: Siekmann, J.H., Wrightson, G. (eds.) Automation of Reasoning. SYMBOLIC, pp. 342–376. Springer, Heidelberg (1983). https://doi.org/10.1007/978-3-642-81955-1_23
22. Korp, M., Sternagel, C., Zankl, H., Middeldorp, A.: Tyrolean termination tool 2. In: Treinen, R. (ed.) RTA 2009. LNCS, vol. 5595, pp. 295–304. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02348-4_21
23. Kurihara, M., Kondo, H.: Completion for multiple reduction orderings. J. Autom. Reason. **23**(1), 25–42 (1999)

24. Lau, T., Wolfman, S.A., Domingos, P., Weld, D.S.: Programming by demonstration using version space algebra. Mach. Learn. **53**(1–2), 111–156 (2003)
25. Liang, P., Jordan, M.I., Klein, D.: Learning programs: a hierarchical Bayesian approach. In: Proceedings of the 27th International Conference on Machine Learning, ICML 2010, pp. 639–646 (2010)
26. Löchner, B.: Things to know when implementing KBO. J. Autom. Reason. **36**(4), 289–310 (2006)
27. McCune, W.: Experiments with discrimination-tree indexing and path indexing for term retrieval. J. Autom. Reason. **9**(2), 147–167 (1992). https://doi.org/10.1007/BF00245458
28. Menon, A.K., Tamuz, O., Gulwani, S., Lampson, B.W., Kalai, A.: A machine learning framework for programming by example. In: ICML (2013)
29. Novikov, P.S.: On the algorithmic unsolvability of the word problem in group theory. Trudy Matematicheskogo Instituta imeni VA Steklova **44**, 3–143 (1955)
30. Osera, P., Zdancewic, S.: Type-and-example-directed program synthesis. In: PLDI (2015)
31. Otto, F.: On the connections between rewriting and formal language theory. In: Narendran, P., Rusinowitch, M. (eds.) RTA 1999. LNCS, vol. 1631, pp. 332–355. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48685-2_27
32. Phothilimthana, P.M., Thakur, A., Bodik, R., Dhurjati, D.: Scaling up superoptimization. In: Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 297–310. ACM (2016)
33. Polikarpova, N., Kuraj, I., Solar-Lezama, A.: Program synthesis from polymorphic refinement types. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 522–538. ACM (2016)
34. Reddy, U.S.: Rewriting techniques for program synthesis. In: Dershowitz, N. (ed.) RTA 1989. LNCS, vol. 355, pp. 388–403. Springer, Heidelberg (1989). https://doi.org/10.1007/3-540-51081-8_121
35. Schkufza, E., Sharma, R., Aiken, A.: Stochastic superoptimization. ACM SIGPLAN Not. **48**(4), 305–316 (2013)
36. Smith, C., Albarghouthi, A.: MapReduce program synthesis. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 326–340. ACM (2016)
37. Smith, C., Ferns, G., Albarghouthi, A.: Discovering relational specifications. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, pp. 616–626. ACM (2017)
38. Solar-Lezama, A., Tancau, L., Bodík, R., Seshia, S.A., Saraswat, V.A.: Combinatorial sketching for finite programs. In: ASPLOS (2006)
39. Udupa, A., Raghavan, A., Deshmukh, J.V., Mador-Haim, S., Martin, M.M., Alur, R.: TRANSIT: specifying protocols with concolic snippets. ACM SIGPLAN Not. **48**(6), 287–296 (2013)
40. Wang, X., Dillig, I., Singh, R.: Program synthesis using abstraction refinement. Proc. ACM Program. Lang. **2**(POPL), 63 (2017)
41. Wehrman, I., Stump, A., Westbrook, E.: Slothrop: Knuth-Bendix completion with a modern termination checker. In: Pfenning, F. (ed.) RTA 2006. LNCS, vol. 4098, pp. 287–296. Springer, Heidelberg (2006). https://doi.org/10.1007/11805618_22
42. White, T.: Hadoop - The Definitive Guide: Storage and Analysis at Internet Scale (2015)

43. Winkler, S., Middeldorp, A.: Termination tools in ordered completion. In: Giesl, J., Hähnle, R. (eds.) IJCAR 2010. LNCS, vol. 6173, pp. 518–532. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14203-1_43
44. Yu, Y., et al.: DryadLINQ: a system for general-purpose distributed data-parallel computing using a high-level language. In: OSDI (2008)
45. Zaharia, M., et al.: Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: NSDI (2012)
46. Zankl, H., Hirokawa, N., Middeldorp, A.: KBO orientability. J. Autom. Reason. **43**(2), 173–201 (2009)