Oriented Metrics for Bottom-Up Enumerative Synthesis

ROLAND MEYER, TU Braunschweig, Germany
JAKOB TEPE, TU Braunschweig, Germany

In syntax-guided synthesis, one of the challenges is to reduce the enormous size of the search space. We observe that most search spaces are not just flat sets of programs, but can be endowed with a structure that we call an oriented metric. Oriented metrics measure the distance between programs, like ordinary metrics do, but are designed for settings in which operations have an orientation. Our focus is on the string and the bitvector domains, where operations like concatenation and bitwise conjunction transform an input into an output in a way that is not symmetric. We develop several new oriented metrics for these domains.

Oriented metrics are designed for search space reduction, and we present four techniques: (i) pruning the search space to a ball around the ground truth, (ii) factorizing the search space by an equivalence that is induced by the oriented metric, (iii) abstracting the oriented metric (and hence the equivalence) and refining it, and (iv) improving the enumeration order by learning from abstract information. We acknowledge that these techniques are inspired by developments in the literature. By understanding their roots in oriented metrics, we can substantially increase their applicability and efficiency. We have integrated these techniques into a new synthesis algorithm and implemented the algorithm in a new solver. Notably, our solver is generic in the oriented metric over which it computes. We conducted experiments in the string and the bitvector domains, and consistently improve the performance over the state-of-the-art by more than an order of magnitude.

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

Additional Key Words and Phrases: Syntax-Guided Synthesis, Bottom-Up Enumeration, Metrics

ACM Reference Format:
Roland Meyer and Jakob Tepe. 2026. Oriented Metrics for Bottom-Up Enumerative Synthesis. *Proc. ACM Program. Lang.* 10, POPL, Article 75 (January 2026), 30 pages. https://doi.org/10.1145/3776717

1 Introduction

The goal of inductive program synthesis is to synthesize a program from input-output examples that are given as the specification. A prime example for inductive synthesis is the FlashFill [9, 22, 37] feature in Microsoft Excel enabling millions of end-users to automate data manipulation tasks. Other applications include superoptimization [35], program deobfuscation [8, 13], and synthesizing database queries [30, 44]. In this context, the Syntax-Guided Synthesis (SyGuS) paradigm [2] received considerable attention. Given a grammar $\mathcal{G}$ and a specification Φ, the goal is to find a program $\mathsf{p} \in L(\mathcal{G})$ that satisfies Φ. Numerous tools [2, 3, 6, 15, 16, 24, 25, 27, 28, 38, 48] have been developed to tackle SyGuS problems.

Most successful SyGuS solvers implement a bottom-up enumeration, which constructs larger programs from smaller ones until a solution has been found. While conceptually similar, the algorithms differ drastically when it comes to two parameters: the search space of programs that are considered as possible solutions, and the enumeration order in which the search space is explored. Most works aim to improve the enumeration order so as to find a solution quickly. Learning [6, 28] tries to understand which subprograms are likely to play a role in the solution, and therefore should

Authors' Contact Information: Roland Meyer, TU Braunschweig, Braunschweig, Germany, roland.meyer@tu-bs.de; Jakob Tepe, TU Braunschweig, Braunschweig, Germany, j.tepe@tu-braunschweig.de.

ACM 2475-1421/2026/1-ART75
https://doi.org/10.1145/3776717

be enumerated early on. Deduction [3, 15, 16, 27, 48] tries to guide the bottom-up enumeration by information about the target values that has been computed top-down from program sketches. Strategies to reduce the enormous search space have received less attention. The standard technique is to factorize the search space along observational equivalence [1, 2, 43]: if two programs have the same outputs on the given inputs, it suffices to keep one of them. The comparison has been weakened to an abstraction of the output values, and a refinement loop has been introduced to recover from imprecision and remain complete [47].

A drastically new approach to reduce the search space starts from the following consideration [21]. Observational equivalence is a discrete judgment: programs may or may not be equivalent. Metrics generalize this to a continuous notion of distance [42]: programs may be closer to each other or further apart. Having a metric on the search space suggests a reduction strategy that we refer to as pruning: only consider programs that are close enough to the ground truth. These programs form a ball around the ground truth whose radius is the threshold on the distance. Pruning deliberately gives up completeness and trades it for performance. The balance between completeness and performance, however, is under the control of the user: it is the radius of the ball. Pruning can be combined with the aforementioned factorization techniques.

While metric search spaces and pruning are attractive conceptually, their applicability has been limited so far. The problem is that metrics require symmetry, $m(i, o) = m(o, i)$. The operations on most data domains, however, are oriented. Consider strings and concatenation. If the string i is a substring of o, then i may help us produce o by concatenation. If i is a superstring, then there is no chance to produce o by concatenation. The situation is similar with bitvectors and the operations of bitwise conjunction, disjunction, and multiplication. In short, the symmetric metrics cannot measure in a meaningful way the impact of operations that are oriented.

Quasimetrics [42] have been proposed as a generalization of metrics that does not require symmetry. With quasimetrics, we can assign a meaningful distance to strings that are manipulated by concatenation: if i is a substring of o, we take as distance the difference in length $|o| - |i|$; if i is a superstring, we take infinity. Similar quasimetrics can be defined for the operations in the bitvector domain. Unfortunately, quasimetrics do not work with example-based specifications. When two programs have the same outputs on the given inputs, their distance should be zero under the quasimetric (lifted to the space of input-output functions). The definition of quasimetrics, however, requires equality for objects with distance zero. This will not hold: the fact that programs agree on a number of inputs does not mean they agree on all inputs.

Contribution. We define a new notion of *oriented metrics (orimetrics)* that targets data domains whose operations are not symmetric. Orimetrics only require reflexivity in the form of $m(a, a) = 0$. We do not require that $m(a, b) = 0$ implies $a = b$ as in quasimetrics. Orimetrics only require symmetry at zero, $m(a, b) = 0$ implies $m(b, a) = 0$. We do not require $m(a, b) = m(b, a)$ as in metrics and pseudometrics. We still have the triangle inequality, $m(a, b) \leq m(a, c) + m(c, b)$.

We present new orimetrics for the *string*, the *bitvector*, and the *function domain*. We give a principled way to derive these orimetrics that should carry over to other domains as well. The idea is to let the orimetric measure the complexity of the inverse semantics of the operator [27, 37].

Orimetrics are designed for pruning and factorization in bottom-up enumerative synthesis. For *pruning*, we still limit the search to a ball $\mathcal{B}_r(gt) = \{\mathsf{p} \mid m(\mathsf{p}, gt) < r\}$, as pioneered in [21]. The difference, however, is that the direction in which we measure the distance matters. For *factorization*, an important insight is that *every orimetric induces an equivalence relation* where $a \equiv_m b$ if $m(a, b) = 0$. This generalizes all factorization strategies discussed above. Depending on the orimetric, a distance of zero may mean the programs produce the same outputs on the inputs

from all examples (observational equivalence), produce the same abstract outputs on the inputs from all examples (the equivalence used by Blaze [47]), or produce similar outputs on some inputs.

Factorization makes it attractive to work with an orimetric that is rough in that it equates many programs. At the same time, the orimetric should faithfully represent the distance to the ground truth. To reconcile these desiderata, we work with *approximate orimetrics* $m^\sharp$ and introduce a refinement scheme. Approximate here means that $m^\sharp(\mathsf{p}, gt) = 0$ may hold although $\mathsf{p} \neq gt$. The reflexivity requirement for orimetrics plays a surprisingly important role for the refinement. We can refine $m^\sharp$ to any ${m_\mathsf{p}}^\sharp$ with ${m_\mathsf{p}}^\sharp(\mathsf{p}, gt) \neq 0$, and reflexivity will guarantee that the program will not be considered a candidate solution again. In short, we obtain a stronger factorization at the expense of potentially finding spurious programs and having to refine.

What turned out surprisingly challenging is to harmonize the factorization with the bottom-up enumeration. To see this, consider the set $\{\mathsf{x}, \mathsf{y}, \mathsf{op}(\mathsf{y}), \mathsf{op}(\mathsf{x})\}$ on which we have the equivalence $\mathsf{x} \equiv \mathsf{y}$ and $\mathsf{op}(\mathsf{x}) \equiv \mathsf{op}(\mathsf{y})$. Assume the bottom-up enumeration constructs the programs in the order given by the set, and the factorization maintains the first element in each equivalence class as a representative. Then we obtain $\{\mathsf{x}, \mathsf{op}(\mathsf{y})\}$, which is not bottom-up enumerable. We give natural and easy to satisfy conditions under which the factorized set remains bottom-up enumerable.

We implemented our approach in a tool called Merlin[1] and evaluated its performance against state-of-the-art SyGuS tools and domain-specific solvers. In the bitvector domain, Merlin is 27 times faster than DryadSynth [15], the current best SyGuS solver on the bitvector domain. On Blaze's [47] string benchmarks, it is 75 times faster than Blaze. Overall, Merlin is 42 times faster than a baseline implementation without our proposed techniques. We give a careful evaluation of the impact of pruning and refinement.

In total, we make three contributions:

(1) We define orimetrics for bottom-up enumerative synthesis. Orimetrics allow us to prune, factorize, and refine the search space (Sections 2 to 4).
(2) We define orimetrics for the string, the bitvector, and the function domain. Furthermore, we present a principled way of coming up with orimetrics (Section 5).
(3) We implemented our approach in a tool called Merlin and compared it to the state-of-the-art in the string and the bitvector domain. We win by over an order of magnitude (Section 6).

2 Overview

A SyGuS problem takes as input a specification in the form of a function gt and a grammar for programs $\mathcal{G}$. The task is to find a program $\mathsf{p} \in \mathcal{L}(\mathcal{G})$ that implements the function, $[\![\mathsf{p}]\!] = gt$. We call gt the ground truth and p a solution to the synthesis task. For simplicity, we assume gt is given as a finite set of input-output examples, but remark that our techniques carry over to more elaborate settings. There are various strategies of how to solve a SyGuS problem. The most successful solvers implement a form of bottom-up enumeration [2, 3, 15, 16, 27, 47, 48], where they try to find a solution by composing subprograms that have already been constructed. While every new solver contributes a new technique that makes it faster than the state-of-the-art, there are two parameters that play a role in all solvers. The *search space* P contains the programs that are considered relevant to solve the synthesis task (they may form solutions or occur as subprograms in solutions). The *enumeration order* $\preceq$ defines the order in which the search space should be explored.

Our first insight is that all search spaces of practical interest are not unstructured sets, but can be endowed with an *oriented metric* (orimetric) m that gives information about the distance between programs. The purpose is to deal with data domains whose operations have an orientation. Consider a grammar that supports `concat`, the concatenation of strings. If i is a substring of o, it is

[1] https://github.com/J4K0B/Merlin

easy to find a string i' so that $\texttt{concat}(i, i') = o$. This means $m(i, o)$ should be small. With o being a superstring of i, however, it is impossible to find a string o' so that $\texttt{concat}(o, o') = i$. The distance $m(o, i)$ should be infinity. These considerations prompted us to drop the symmetry requirement in metrics. The resulting object is an orimetric.

We present four techniques that capitalize on the information given by the orimetric to improve the efficiency of bottom-up enumerative synthesis. A remarkable aspect is that our techniques are generic: they only refer to the orimetric and the enumeration order, but do not make assumptions on how these are defined. This makes it possible to use the four techniques as enhancements in virtually any bottom-up enumerative solver.

Our first technique is called **pruning**. Pruning limits the search to a ball around the ground truth, meaning it tries to build a solution solely from the programs contained in this ball. Technically, the ball is the set of programs whose distance to the ground truth is smaller than a threshold r:

$$\mathcal{B}_{(P,m),r}(gt) \quad = \quad \{\, \mathsf{p} \in P \;\mid\; m(\llbracket \mathsf{p} \rrbracket, gt) < r \,\}\,.$$

We refer to r as the radius of the ball, and just write $\mathcal{B}_r(gt)$ when the orimetric (search) space is understood. The orimetric is required to have the mathematical properties described in the introduction (which will be made formal in Section 4). What is surprisingly important in the context of SyGuS is reflexivity of the orimetric:

$$\llbracket \mathsf{p} \rrbracket = gt \quad \Rightarrow \quad m(\llbracket \mathsf{p} \rrbracket, gt) = 0\,. \tag{reflexivity}$$

When read in contraposition, reflexivity says that only programs at distance zero to the ground truth can solve the synthesis task. This, however, does not mean we can just work with a ball of radius almost zero. The purpose of the ball is to constrain the subprograms that can be used to build candidate solutions. Reflexivity then applies to the candidate programs, but not to the subprograms.

The second insight is that the use of an orimetric not only allows us to limit the search to a ball, it also suggests identifying and removing duplicate elements from this ball. We call this **factorization**. Indeed, from the perspective of the orimetric, two programs are equivalent whenever they have a distance of zero. The orimetric thus induces the equivalence

$$\mathsf{p}_1 \equiv_m \mathsf{p}_2 \quad \text{if} \quad m(\llbracket \mathsf{p}_1 \rrbracket, \llbracket \mathsf{p}_2 \rrbracket) = 0\,.$$

We remove duplicates by factorizing the ball along this equivalence, that is, searching $\mathcal{B}_r(gt)/_{\equiv_m}$. In our implementation, we remove duplicates by considering representatives of equivalence classes. Concretely, we use as representatives the minimal elements wrt. the enumeration order $\preceq$. The reader will observe the similarity between factorization and observational equivalence [1, 43]. There, programs are considered equivalent when they return the same outputs on the inputs from all examples, in which case one program will be discarded. This significantly reduces the search space while leaving the search complete. We emphasize that orimetrics are only able to capture factorization because they do not require objects to be equal when they have a distance of zero.

As discussed in the introduction, it would be attractive to have a coarse equivalence that equates many programs. Our third technique is to work with *approximate orimetrics* $m^\sharp$. We say that an orimetric is *precise*, if the converse of reflexivity holds: $m^\sharp(\llbracket \mathsf{p} \rrbracket, gt) = 0$ implies $\llbracket \mathsf{p} \rrbracket = gt$. Otherwise, the orimetric is called *approximate*. In approximate orimetrics, we may have $m^\sharp(\llbracket \mathsf{p} \rrbracket, gt) = 0$ although $\llbracket \mathsf{p} \rrbracket \neq gt$. We thus have to explicitly check whether a program solves the synthesis task. If this is not the case — we call the program spurious — we have a **refinement** function that constructs from $m^\sharp$, p, and gt a new orimetric ${m_\mathsf{p}}^\sharp$. The idea is inspired by and can be combined with the abstraction-refinement approach from [47]. That work uses predicate abstraction on the output to equate programs.

Our fourth technique modifies the enumeration order on-the-fly by **learning** from spurious programs. When we find a spurious program p, we know that p was promising from the perspective of the previous orimetric. While it may not be a solution, p is likely to contain valuable subprograms. We therefore update the enumeration order to list these subprograms early on. To be precise, we only list them if they still reside in the ball that is formed with a precise orimetric m. It is useful to have a precise orimetric at this point to eliminate as many programs as possible. The approach is inspired by [6], where operators are preferred that occur frequently in solutions to single examples.

Another approach to modify the enumeration order are deductive methods [3, 15, 16, 27, 48]. We discuss them next, to make clear that they are orthogonal to the idea of using orimetrics and that the approaches can be combined. One deductive method is case-splitting [3]. When we enumerated a program for each of the given examples, we can build a decision tree to synthesize an if-then-else program that solves all examples. Another deductive method [15, 16, 27, 48] uses the inverse semantics of an operator. Consider the program sketch $\texttt{concat}(?, ?)$ and the output "POPL". Assume we already enumerated a program $\texttt{p}_1$ that outputs "PO". Given "POPL" and "PO", the inverse semantics is $\texttt{concat}^{-1}(\text{"POPL"}, \text{"PO"}) = \{\text{"PL"}\}$. We check whether we already enumerated another program $\texttt{p}_2$ that outputs "PL". If so, we can complete the sketch to $\texttt{concat}(\texttt{p}_1, \texttt{p}_2)$ and directly solve the synthesis problem. So instead of having to enumerate all programs up to the size of $\texttt{concat}(\texttt{p}_1, \texttt{p}_2)$, we only need to enumerate programs until we find $\texttt{p}_1$ and $\texttt{p}_2$.

We can leverage our understanding of an operator's inverse semantics to construct orimetrics. The idea is that the distance between an input and an output value should be inverse proportional to what may be considered the complexity of the inverse semantics. Continuing on the above example, if i is a superstring of "POPL", the inverse semantics is $\texttt{concat}^{-1}(\text{"POPL"}, i) = \emptyset$ and hence the distance $m(i, \text{"POPL"})$ should be infinity. As a less extreme case, a prefix "POP" would leave us with $\texttt{concat}^{-1}(\text{"POPL"}, \text{"POP"}) = \{\text{"L"}\}$, which is likely easier to generate than "PL", and therefore $m(\text{"POP"}, \text{"POPL"}) < m(\text{"PO"}, \text{"POPL"})$ should hold.

Figure 1 presents a generic bottom-up enumerative synthesis algorithm that incorporates our four techniques. Note that the algorithm is parametric in the enumeration order and in the initial orimetric. This means a tool will not have to hard code the enumeration order and the orimetric, but can take them as input, together with the grammar The algorithm implements the counterexample-guided abstraction refinement loop [11, Chapter 13] explained above. The input is the SyGuS task of interest. In the first step, we prune the search space to a ball P around the ground truth. Apart from the initial search space $\mathcal{L}(\mathcal{G})$, `Prune` takes as input the orimetric $m^\sharp$ and gt. In the second step, we factorize this ball and obtain Q. In the third step, we search Q in the order $\preceq$ for a program satisfying the specification. If we find a program p that $m^\sharp$ believes satisfies the specification, $m^\sharp(\llbracket \texttt{p} \rrbracket, gt) = 0$, we hand it over to the next step. Since $m^\sharp$ is approximate, we have to check whether p solves the SyGuS instance. If so, the loop stops and returns p. If p turns out to be spurious, we pass it to `Refine`. Using also $m^\sharp$ and gt, the refinement determines an updated approximate orimetric $m_\texttt{p}{}^\sharp$ with $m_\texttt{p}{}^\sharp(\llbracket \texttt{p} \rrbracket, gt) \neq 0$. In the final step, we update the enumeration order using function `Learn`. It takes as input the current enumeration order $\preceq$, all programs enumerated by `Search`, and the ground truth gt. By analyzing the enumerated programs, it learns a better enumeration order for the next iteration. Then the loop repeats.

Remarks. The above description is conceptual in that it separates the functions more than we do in our implementation. The workhorse of our implementation is the search for a candidate solution. Pruning and factorization run interleaved with it. Concretely, `Search` constructs the programs one by one: given a list of programs that have already been constructed, it is able to determine the program p that should be constructed next according to the enumeration order $\preceq$. If p does not belong to the ball of interest, $m^\sharp(\llbracket \texttt{p} \rrbracket, gt) \geq r$, it is discarded. The same holds if we already have

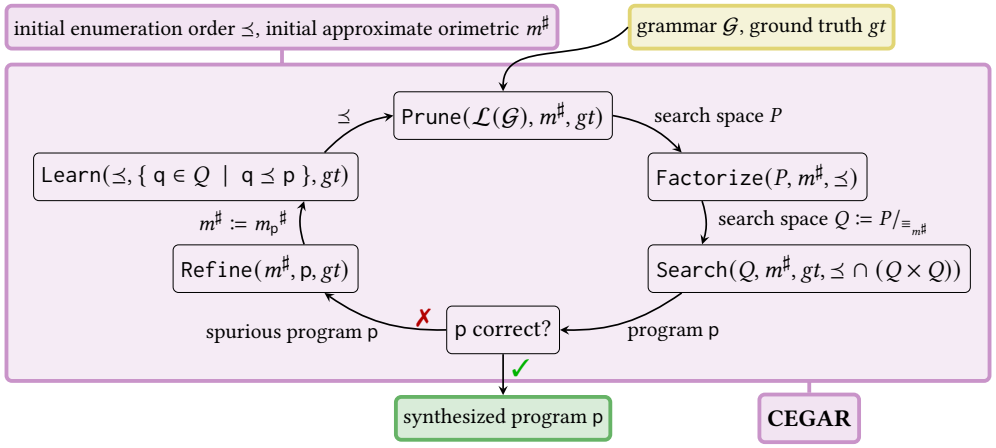


Fig. 1. CEGAR loop for synthesis.

another representative q in the list, meaning $m^\sharp([\![\mathsf{p}]\!], [\![\mathsf{q}]\!]) = 0$. We already know $\mathsf{q} \preceq \mathsf{p}$, and thus q should be the representative. If the program passes these tests, we append it to the list.

Why do we keep the enumeration order? An alternative would be to just have an orimetric and imitate the enumeration order by going through the programs in the order of their distance to the ground truth. First, we believe the enumeration order is such an integral part of the solving process that it deserves being a parameter on its own. Second, our metrics are very coarse: they are made to define the ball but do not distinguish much between the programs inside the ball. This will become clear in the next section where we illustrate our approach on an example. Third, the orimetrics are not as flexible as the enumeration order. An orimetric has to satisfy a few mathematical properties, whereas the enumeration order just has to be a total order on the search space. This flexibility makes it easier to adapt the enumeration order based on learned information.

One may also ask whether the concept of orimetrics is needed after all, or whether we could have based our algorithmic improvements on more elementary mathematical notions. One could try to prune the search space with a quasimetric and factorize the search space with an equivalence. Quasimetrics have the problem that a distance of zero should imply equality of the elements, which does not hold in example-based settings. The elements are, however, observationally equivalent. The discussion suggests we could endow the search space P with an equivalence $\equiv \subseteq P \times P$ that should be used for factorization and with a quasimetric on the now factorized space $q \subseteq P/_\equiv \times P/_\equiv$ that could be used for pruning. To our surprise, this alternative definition $(P, \equiv, q)$ is equivalent to our notion of orimetric search spaces (P, m): a combination of an equivalence and a quasimetric is enough to induce an orimetric, and vice versa. At the same time, the alternative definition has disadvantages that orimetrics overcome: (i) working with equivalence classes is cumbersome, we believe the symmetry at zero requirement for orimetrics is simpler, (ii) one has to understand refinement for two objects, the equivalence and the quasimetric, and maintain both objects during computation, (iii) there is no guidance on how to obtain the equivalence, while it is a derived concept for orimetrics. All this indicates that the notion of orimetric search spaces is somewhat fundamental to bottom-up enumerative synthesis.

2.1 Example

We illustrate our algorithm by solving an example problem in the string domain. We will repeatedly refer to Figure 3 to make the link to the conceptual development introduced above. Figure 2

$$\begin{array}{rcl} \mathcal{S} & ::= & \mathit{Init} \mid \texttt{replace}(\mathcal{S},\mathcal{S},\mathcal{S}) \mid \texttt{concat}(\mathcal{S},\mathcal{S}) \\ \mathit{Init} & ::= & \texttt{x} \mid \epsilon \mid \texttt{"␣Conference"} \mid \texttt{"␣City"} \end{array}$$

Fig. 2. Example grammar $\mathcal{G}$.

Table 1. Input-output examples (I, O).

n	Input	Output
1	"POPL␣Conference"	"POPL"
2	"Rennes␣City"	"Rennes"
3	"PLDI␣Conference"	"PLDI"
4	"Seoul␣City"	"Seoul"

depicts a context-free grammar $\mathcal{G}$ and Table 1 shows gt as four input-output examples (I, O). The nonterminal *Init* can be rewritten to the input variable x or to string constants, where ϵ is the empty string. The operator `replace` takes as arguments three strings: the first is the string in which the replacement should happen, namely the first occurrence of the second argument, should it exist, will be replaced by the third argument. The operator `concat` concatenates the strings that are given as arguments.

In Figure 4, we show the set $\mathcal{L}(\mathcal{G})$ of all programs as a collection of sets P_i that contain all programs of size i. The set P_1 contains all programs that can be derived from *Init*. Programs of size greater than one are constructed by combining programs of smaller size. We use an enumeration order $\preceq$ that orders programs by their size. Programs that have the same size are ordered from left to right in Figure 4. The set P_7 contains the solution to the synthesis task, highlighted in yellow. The first row in Table 2 gives the cardinality of each set P_i. Note how the number of programs grows exponentially with the size, and so eliminating programs early on is essential. In Figure 3, the set of programs is represented by the gray box and the enumeration order by the dashed line.

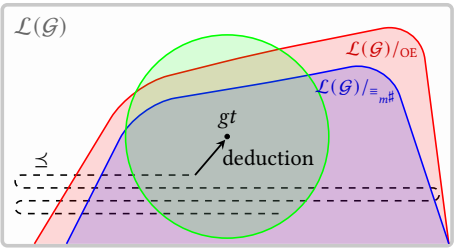


Fig. 3. Search space and enumeration order.

State-of-the-art bottom-up enumerative SyGuS solvers [2, 3, 15, 16, 27, 47, 48] factorize the search space, often using observational equivalence [1, 43]. In Figure 4, programs that are pruned because they are observationally equivalent to a program that was enumerated earlier are highlighted in red. The cardinality of the factorized sets is given in the second row in Table 2. In Figure 3, this search space is represented by the red area.

Using orimetrics, we can further reduce the search space. The first step is to define an orimetric on programs. The orimetric is chosen based on the data and data manipulations that should be supported. For strings with replacement, it is beneficial to reward programs that produce superstrings of the outputs given in the examples and punish programs that do not. We explain in a moment how this plays together with the fact that concatenation prefers substrings. We begin by defining an auxiliary quasimetric $\widetilde{m}$ on strings:

$$\widetilde{m}(i, o) = \begin{cases} len(i) - len(o) & \text{if } i \text{ is a superstring of } o \\ 100 + |len(i) - len(o)| & \text{otherwise .} \end{cases}$$

For example, $\widetilde{m}(\text{"PO"}, \text{"POPL"}) = 102$ but $\widetilde{m}(\text{"POPL"}, \text{"PO"}) = 2$. Note that $\widetilde{m}$ is not symmetric in general, but it is symmetric at distance zero. We now lift the auxiliary quasimetric $\widetilde{m}$ on strings to an orimetric m on functions over strings. The definition is as expected, we evaluate the given functions on the inputs from all examples and sum up the distances of the output values. To make

$P_1 = \{\mathtt{x}, \epsilon, \text{"␣Conference"}, \text{"␣City"}\}$

$P_2 = \emptyset$

$P_3 = \{\mathtt{x}.\mathtt{x}, \mathtt{x}.\epsilon, \mathtt{x}.\text{"␣Conference"}, \mathtt{x}.\text{"␣City"}, \epsilon.\mathtt{x}, \epsilon.\epsilon, \epsilon.\text{"␣Conference"}, \epsilon.\text{"␣City"},$
$\text{"␣Conference"}.\mathtt{x}, \text{"␣Conference"}.\epsilon, \text{"␣Conference"}.\text{"␣Conference"},$
$\text{"␣Conference"}.\text{"␣City"}, \text{"␣City"}.\mathtt{x}, \text{"␣City"}.\epsilon, \text{"␣City"}.\text{"␣Conference"},$
$\text{"␣City"}.\text{"␣City"}\}$

$P_4 = \{\ldots, \mathtt{r}(\mathtt{x}, \mathtt{x}, \text{"␣City"}), \mathtt{r}(\mathtt{x}, \epsilon, \epsilon), \mathtt{r}(\mathtt{x}, \text{"␣City"}, \text{"␣Conference"}), \ldots, \mathtt{r}(\mathtt{x}, \text{"␣Conference"}, \epsilon), \ldots\}$

$P_5 = \{\ldots\} \qquad P_6 = \{\ldots\} \qquad P_7 = \{\ldots, \mathtt{r}(\mathtt{r}(\mathtt{x}, \text{"␣Conference"}, \epsilon), \text{"␣City"}, \epsilon), \ldots\}$

Fig. 4. Example execution of a bottom-up enumeration algorithm with highlights for OE factorization, OE factorization with learning, orimetric pruning, partial correctness, and correctness. The concat operator is abbreviated by an infix "." and the replace operator is abbreviated by r.

this formal, let $\mathcal{J} \subseteq I$ and define

$$m_{\mathcal{J}}(f, g) \;=\; \sum_{i \in \mathcal{J}} \widetilde{m}(f(i), g(i)) \;.$$

Our orimetric is then $m = m_I$. We will use other instantiations of $\mathcal{J}$ in a moment. Note, that the lifting does not result in a quasimetric. If two different programs p and q produce the same outputs on the given inputs, their distance is 0. A quasimetric would require p and q to be equal. This need not be the case. In fact, allowing p and q to have a distance of zero while being different is what enables factorization in the first place.

With the orimetric at hand, we restrict the search to programs in the ball $\mathcal{B}_r(gt)$, with the radius r set to 100. This means we only keep programs that return a superstring of the output for every input example. We admit that this is discrete, and does not make use of the fact that an orimetric yields continuous values in $\mathbb{R}_{\geq 0}$. In our experiments, we will see more elaborate instantiations.

Note that focusing on superstrings does not mean we are not allowed to use the concat operator. We can still use concat, but on the programs inside the ball. The superstring ball is even closed under concatenation: concatenating superstrings yields a superstring. One may ask how this relates to the above argument that concat prefers substrings. This argument was made for a comparison with the final value. Since we use superstrings here, it means the final value cannot be produced by concatenation, and so concat will not be the topmost operator in the solution to the synthesis task. It can, however, still be a part of it.

Coming back to Figure 4, the programs that are additionally pruned by this orimetric are highlighted in green. Figure 3 depicts the resulting search space as the green ball around gt. The cardinality of the program sets is shown in Table 2. Using only pruning is not as effective as factorizing along observational equivalence, but combined we are only left with 81 programs to explore in P_7. In Figure 3, this search space is the intersection of the green ball and the red area.

With approximate orimetrics $m^\sharp$, we can reduce the search space even further. The approximate orimetrics should be rough so that the induced equivalence $\equiv_{m^\sharp}$ eliminates many programs. We will later recover precision by adding a refinement loop. In our example, we use $m^\sharp = m_{\mathcal{J}}$ with $\mathcal{J} = \{i_1\}$. This approximate orimetric only takes into account the first example when comparing functions, $m^\sharp(f, g) = \widetilde{m}(f(i_1), g(i_1))$. The induced equivalence equates programs at distance zero—actually, this is why we wanted the orimetric to be symmetric at distance zero. Here, we equate programs that have the same output on the first input. To give an example, $\mathtt{x} \equiv_{m^\sharp} \mathtt{r}(\mathtt{x}, \text{"␣City"}, \text{"␣Conference"})$.

Table 2. Number of programs at each size per solving method.

Method	P_1	P_2	P_3	P_4	P_5	P_6	P_7
No Pruning or Factorization	4	-	16	64	128	1280	4352
OE Factorization	4	-	9	6	27	56	119
Orimetric Pruning (OP)	4	-	7	18	56	323	929
OE Factorization + OP	4	-	5	6	19	50	81
$\equiv_{m^\sharp}$ Factorization + OP + Learning	4 / 5	-	5 / 12	3 / 10	-	-	-

$P_1 = \{\texttt{r(x, "␣Conference", }\epsilon\texttt{), x, }\epsilon\texttt{, "␣Conference", "␣City"}\}$

$P_2 = \emptyset$

$P_3 = \{\texttt{x.x, x.}\epsilon\texttt{, x."␣Conference", x."␣City", x.r(x, "␣Conference", }\epsilon\texttt{), ..., }\epsilon\texttt{."␣City", ...}\}$

$P_4 = \{\ldots, \texttt{r(x, x, "␣City")}, \texttt{r(x, }\epsilon\texttt{, }\epsilon\texttt{)}, \ldots, \texttt{r(r(x, "␣Conference", }\epsilon\texttt{), "␣City", }\epsilon\texttt{)} \ldots\}$

Fig. 5. Example execution of the second iteration of our algorithm with highlights for OE factorization and correctness. The concat operator is abbreviated by an infix "." and the replace operator is abbreviated by r.

The equivalence holds as "␣City" does not occur in "POPL␣Conference", and so no replacement happens. When we factorize the search space along $\equiv_{m^\sharp}$, the program in blue will be eliminated since we already have x. In Figure 3, the factorized search space is the blue area. When applying Prune as well as Factorize, we are left with the green ball intersected with the blue area. We now use Search to find a solution candidate for the synthesis problem in this search space.

Function Search enumerates the programs along the order $\preceq$. Observe that the output of the program p in cyan, when executed on the first example, is "POPL". Therefore, under $m^\sharp$, the distance to the ground truth is zero, and Search returns p as a candidate solution. Notably, to find the candidate solution, we only needed to enumerate programs of size up to 4.

Unfortunately, program p does not work on some of the other examples: the solution is spurious. For instance, on the second example the program yields "Rennes␣City" which is not equal to the specified output "Rennes". We use Refine to generate a new orimetric for the next iteration. To also take into account the second example, we set $m^\sharp$ to $m_{\{i_1,i_2\}}$. While the approximate orimetric and its refinement are simple in this example, our approach can also be instantiated with predicate abstraction as introduced in [47]. We discuss the details below.

We now use Learn to update the enumeration order for the next iteration. To do so, we use the precise metric m_I to analyze the candidate solution p. We see that p produces a superstring of the outputs for all examples. This means, $m_I(\llbracket \texttt{p} \rrbracket, gt) < 100$ and therefore the program lies within the ball around gt. We consider p valuable and promote it in the enumeration order. Recall that the enumeration order in our example is defined by the size. Function Learn redefines the size of p to be 1. Then p and programs that use p as a subprogram will be considered early on.

Figure 5 gives the program sets constructed in the second iteration of the refinement loop. Note how p has size one. With this change in size, we already find the solution in P_4. In the last row of Table 2, we see the cardinalities of the sets again. On the left of the slash are the cardinalities for the first iteration of the refinement loop, and on the right the cardinalities for the second iteration. In total, we only consider 39 programs to solve the problem.

Selecting an Orimetric. Coming up with a good orimetric is key to the success of our method. In our example, had we defined an orimetric that rewards functions producing substrings of the

Table 3. Comparison of enumerative SyGuS solvers.

Solver	Search Space Pruning / Factorization	Enumeration Order (Deduction)
ESolver [2]	Observational Equivalence (OE)	constant (✗)
EUSolver [3]	OE	constant (✓)
Blaze [47]	OE + Abstraction Refinement + Automata	constant (✗)
EUPhony [28]	weak OE	constant/offline learning (✓)
Probe [6]	OE	learning (✗)
Duet [27]	OE	constant (✓)
Simba [48]	OE	constant (✓)
DryadSynth [15]	OE	constant (✓)
Synthphonia [16]	OE	constant (✓)
Merlin	OE + Abstraction Refinement + Orimetrics	learning (✓)

output and punish those that produce superstrings, we would have failed to generate the solution using bottom-up enumeration.

To construct good orimetrics, we derive them from the semantics of the operators in the grammar. We already discussed how concatenation needs substrings to produce the desired output while replacement favors superstrings. In practice, we concurrently run a portfolio of solvers employing a different orimetric each. We developed three orimetrics for the string domain and four orimetrics for the bitvector domain. In Sections 5.3 and 5.4, we introduce a principled way of designing orimetrics.

Deduction. An aspect the above example did not highlight is the use of deduction. Conceptually, deduction accelerates the search by modifying the enumeration order, as depicted in Figure 3. Our tool Merlin implements the deduction technique from DryadSynth [15]. To explain it, consider the sketch `r(?, ?, ?)`. After we found the programs `"␣City"`, ϵ, and `r(x, "␣Conference"`, ϵ), we would enumerate `r(r(x, "␣Conference"`, ϵ), `"␣City"`, ϵ) next, which solves the synthesis problem. Deduction thus understands a program sketch. This relies on the inverse semantics of a sketch. If one finds arguments such that the sketch filled with these arguments is a solution to the synthesis problem, the solution will be enumerated next. This means, for any supported sketch, we only need to find the correct arguments. Most orimetrics we define in this paper optimize the search for viable arguments of a sketch by approximating its inverse semantics.

2.2 State-of-the-Art

We discuss to what extent the state-of-the-art SyGuS solvers [2, 3, 15, 16, 27, 28, 47, 48] can be seen as instances of the generic solver in Figure 1. The discussion shows that rather different techniques can be understood as (i) assuming an *orimetric* on the search space and (ii) improving an *enumeration order* through learning. These two ingredients are often left implicit in the related work. By making them explicit and giving them rigorous definitions, we provide a framework in which SyGuS technology can be developed.

The discussion of the solvers is summarized in Table 3. The second column describes the pruning technique that is applied to reduce the search space. The third column categorizes the enumeration order as constant or learning, and indicates whether deductive elements are used.

ESolver [2] is the basis of enumerative SyGuS solvers. It enumerates programs by size until it finds a program that works for all examples. ESolver applies observational equivalence to factorize the search space. The enumeration order is constant and does not use deduction. EUSolver [3] advances ESolver by adding deduction. If for every example a suitable program has been found,

EUSolver tries to construct an if-then-else program that solves all examples. Behind the construction is a decision tree classification of the examples.

Blaze [47] was the first solver based on abstraction refinement [11, Chapter 13]. The focus is on matrix transformation and string problems, and Blaze expects to have access to three pieces of information about the domain: the cost for each production in the grammar, a set of predicates that can be used to abstract data values [11, Chapter 15], and an abstract semantics for each operator. Blaze then solves the synthesis problem in the abstract, and refines the abstract domain by adding predicates if the solution turns out to be spurious. Blaze can be seen as an instance of the generic algorithm in Figure 1. For an orimetric based on predicate abstraction, let $f^\sharp$ be the abstraction of f and let $\alpha(i)$ be the abstraction of the input value i. Both can be computed with the information assumed by Blaze. One then defines

$$m_{Blaze}(f,g) \;=\; \sum_{i\in I} \widetilde{m}(f^\sharp(\alpha(i)), g^\sharp(\alpha(i))) \;.$$

Another novelty in Blaze is that the search space is represented and explored using automata-theoretic techniques. This important algorithmic aspect is not reflected in our generic solver, which is more on the semantic level. The enumeration order is constant, namely defined by the costs.

EUPhony [28] is the first solver that uses probabilistic information. It translates the given grammar into a probabilistic variant, and then traverses the language using A*. These probabilities have been learned from solutions to a large collection of synthesis tasks. The learning is offline, it happens before the translation and the probabilities are not adapted during the search. The enumeration order, which is determined by A* based on the probabilities, is therefore constant in our terminology. EUPhony applies a weak version of observational equivalence that works on sentential forms to factorize the search space. It adopts the deduction techniques from EUSolver to synthesize programs for benchmarks that require case-splitting.

Probe [6] uses just-in-time learning to train the grammar. It assigns costs to each production, and the cost of a program is then the sum of the costs of the productions needed to derive it. Probe enumerates programs in the order of increasing costs, and stops when a cost limit is reached or a solution is found. When the cost limit is reached, the costs are updated as follows. Operators which occur frequently in programs solving at least one example are assigned a lower cost than the other operators. The technique immediately fits, and actually inspired, our refinement of the enumeration order. Probe applies observational equivalence factorization and does not do deduction.

Duet [27], Simba [48], DryadSynth [15], and Synthphonia [16] are similar when it comes to the following characteristics: they all use observational equivalence to factorize the search space, have a constant enumeration order, enumerate based on the program size (with sharing in [15]), and use a variant of the deduction technique from EUSolver to deal with synthesis tasks that need case-splitting. What distinguishes them are their own new deduction techniques.

Duet [27] solves SyGuS problems in the theories of bitvectors, Booleans, strings, and integers. Its deduction is based on an inverse semantics for the operators in the grammar. When one argument for an operator has been fixed, the inverse semantics provides subproblems for the remaining arguments. These are solved by inserting an already enumerated term or by further decomposition.

Simba [48] concentrates on bitvectors and advances Duet's deduction process. The deduction computes necessary preconditions for the arguments of a program sketch. If a program satisfying the preconditions of an argument has been found, it is inserted at the corresponding place, the preconditions for the remaining arguments are refined, and the process repeats.

DryadSynth [15] solves SyGuS problems in the theory of bitvectors. DryadSynth keeps a set of sketches that are hardcoded into the algorithm. This results in a drastic increase in performance compared to Simba. DryadSynth enumerates programs in order of their size (with sharing). For

each enumerated program it checks whether there are other previously enumerated programs with which a sketch can be completed. To perform this search efficiently, DryadSynth maintains viable programs for each sketch in a separate data structure.

Synthphonia solves synthesis problems in the string domain and works on more expressive grammars than specified in the SyGuS format. Synthphonia introduces a framework to perform deduction and enumeration concurrently instead of in an interleaved fashion, and has specialized data structures for the communication between the threads. Moreover, the case-splitting deduction technique from EUSolver is also implemented in a concurrent fashion.

The general solver we propose in Section 4 can be instantiated to the above solvers. While the above techniques mostly manipulate the enumeration order, our tool Merlin focuses on the search space (but can also update the enumeration order). Merlin solves SyGuS problems in the bitvector and in the string domain. Furthermore, Merlin uses abstraction refinement to factorize the search space more effectively. Merlin also incorporates the deduction techniques from DryadSynth and extends them to more sketches as well as to sketches in the string domain. The basis for these deduction techniques is to have an inverse semantics. Orimetrics can capture the complexity of the inverse semantics and prune programs accordingly.

3 Preliminaries

Definition 3.1. A *SyGuS problem* $(\mathcal{G}, \Phi)$ consists of a context-free grammar $\mathcal{G}$ and a specification Φ. The task is to find a program $\mathsf{p} \in L(\mathcal{G})$ that satisfies the specification, $\mathsf{p} \models \Phi$.

We briefly recall the ingredients. A *context-free grammar* $\mathcal{G} = (\mathcal{N}, \mathcal{T}, \mathcal{P}, \mathcal{S})$ consists of finite sets of nonterminals $\mathcal{N}$, terminals $\mathcal{T}$, and productions $\mathcal{P} \subseteq \mathcal{N} \times (\mathcal{T} \cup \mathcal{N})^*$, together with a start nonterminal $\mathcal{S} \in \mathcal{N}$. The term language of a nonterminal, denoted by $\widetilde{L}(\mathcal{A})$, consists of all words over $\mathcal{T}$ and $\mathcal{N}$ that can be derived from $\mathcal{A}$ using the productions. The language of $\mathcal{A}$ is limited to the terminal words, $L(\mathcal{A}) = \widetilde{L}(\mathcal{A}) \cap \mathcal{T}^*$. The language of the grammar is $L(\mathcal{G}) = L(\mathcal{S})$. The complete language of the grammar considers all nonterminals, $\mathcal{L}(\mathcal{G}) = \bigcup_{\mathcal{A} \in \mathcal{N}} L(\mathcal{A})$. Similarly, the complete term language of the grammar is $\widetilde{\mathcal{L}}(\mathcal{G}) = \bigcup_{\mathcal{A} \in \mathcal{N}} \widetilde{L}(\mathcal{A})$.

In SyGuS, the terminals are variables or operators. Variables have arity zero, operators may have an arity greater than zero. We expect the grammar to respect the arity, meaning the productions have to provide the expected number of arguments. With this requirement, we can simply call terminal words programs, and write them as $\mathsf{p} \in L(\mathcal{S})$. A word over terminals and nonterminals is a program sketch $\mathsf{s} \in \widetilde{\mathcal{L}}(\mathcal{G})$. The holes of a sketch are the unresolved nonterminals. The arity of a sketch is the number of its holes, say n. The program that results from replacing the holes by programs p_1 to p_n is $\mathsf{s}(\mathsf{p}_1, \ldots, \mathsf{p}_n)$. We use $\sqsubseteq \subseteq \mathcal{L}(\mathcal{G}) \times \mathcal{L}(\mathcal{G})$ for the subprogram relation where $\mathsf{p} \sqsubseteq \mathsf{q}$, if there is a sketch s so that $\mathsf{q} = \mathsf{s}(\mathsf{p})$. We use $\mathsf{q}{\downarrow} = \{\mathsf{p} \in \mathcal{L}(\mathcal{G}) \mid \mathsf{p} \sqsubseteq \mathsf{q}\}$ for the set of all subprograms of q, including q itself. This is the downward closure wrt. $\sqsubseteq$.

A domain $(D, \mathcal{I})$ consists of a set of data values D and an interpretation $\mathcal{I}$. The interpretation assigns a function $\mathcal{I}(\mathsf{op}) = D^k \to D$ to each operator op of arity k. The domain gives rise to a semantics for programs $\mathsf{p} \in L(\mathcal{G})$. The semantics is the function $[\![\mathsf{p}]\!]$ of type $\mathcal{F} = (\mathit{Vars} \to D) \to D$. It takes as input a variable assignment $i : \mathit{Vars} \to D$ and returns a data value. The definition is as expected: $[\![\mathsf{x}]\!](i) = i(\mathsf{x})$ and $[\![\mathsf{op}(\mathsf{p}_1, \ldots, \mathsf{p}_k)]\!](i) = \mathcal{I}(\mathsf{op})([\![\mathsf{p}_1]\!](i), \ldots, [\![\mathsf{p}_k]\!](i))$. It is common in SyGuS to take the domain and the semantics of programs as defined by the SMT-LIB [7] standard. We follow this convention.

We consider example-based specifications where $\Phi \subseteq (\mathit{Vars} \to D) \times D$ consists of a finite set of input-output examples. A program satisfies the specification, $\mathsf{p} \models \Phi$, if $[\![\mathsf{p}]\!](i) = o$ for all $(i, o) \in \Phi$. The *ground truth* $GT \subseteq \mathcal{F}$ consists of all functions that satisfy the specification in this sense.

CEGIS. SyGuS problems that are not example based can still be solved with example-based techniques. The idea, known as counterexample-guided inductive synthesis [41], is to let an SMT solver generate new examples should a candidate program not yet satisfy the specification. With this argument, we focus on example-based specifications.

Equivalences. An equivalence $\equiv \subseteq S \times S$ on a set S is a relation that is reflexive, symmetric, and transitive. We write $[a]_\equiv = \{b \mid b \equiv a\}$ for the equivalence class of $a \in S$. We just write $[a]$ if the equivalence relation is understood. We lift the notation to sets $G \subseteq S$ and define $[G] = \bigcup_{g \in G} [g]$. We call this the closure of G under $\equiv$. The equivalence is precise wrt. G, if the closure does not add any elements, $[G] \subseteq G$. Note that the reverse inclusion always holds. If the equivalence is not precise wrt. G, it is called approximate. The equivalence is unambiguous wrt. G, if $[G] = [g]$ for all $g \in G$. This means all elements in G are equivalent. Let $f : S \to S$ be a transformer on S. The equivalence is a congruence wrt. f, if $a \equiv b$ implies $f(a) \equiv f(b)$. The definition generalizes to functions in several arguments in the expected way. Factorizing S along $\equiv$ yields the set of equivalence classes $S/_\equiv = \{[a] \mid a \in S\}$. A representative system for $S/_\equiv$ is a set $R \subseteq S$ that contains precisely one element $c \in C$ for every class $C \in S/_\equiv$.

4 Contribution I – Oriented Metric Search Spaces

We first define the main object of this paper, oriented metrics, and then turn to the search space and the enumeration order. Finally, we describe the components of our CEGAR loop in detail.

4.1 Oriented Metrics

It will be convenient to give the definition for arbitrary sets S.

Definition 4.1. A function $m : S \times S \to \mathbb{R}_{\geq 0}$ is an *oriented metric (orimetric)* if, for all $a, b, c \in S$,

$$m(a,a) = 0 \qquad \text{(reflexivity)}$$

$$m(b,a) = 0 \implies m(a,b) = 0 \qquad \text{(symmetry at zero)}$$

$$m(a,c) \leq m(a,b) + m(b,c)\,. \qquad \text{($\triangle$-inequality)}$$

The *equivalence induced by* m is the relation $\equiv_m \subseteq S \times S$ where $a \equiv_m b$ if $m(a,b) = 0$, for all $a, b \in S$. The orimetric is *precise*, *approximate*, resp. *unambiguous* wrt. $G \subseteq S$, if the equivalence $\equiv_m$ has these properties. The orimetric is a *congruence* wrt. $f : S \to S$, if this holds for $\equiv_m$. We generalize the congruence requirement to functions in several arguments where needed. The *quasimetric induced by* m is $q_m : S/_{\equiv_m} \times S/_{\equiv_m} \to \mathbb{R}_{\geq 0}$ with $q_m([a],[b]) = m(a,b)$.

We also suggested an alternative to orimetrics.

Definition 4.2. A *factorization and pruning structure* $(S, \equiv, q)$ consists of an equivalence relation $\equiv \subseteq S \times S$ and a quasimetric $q : S/_\equiv \times S/_\equiv \to \mathbb{R}_{\geq 0}$ on the factorized set. Recall that a quasimetric requires the triangle inequality and $q([a],[b]) = 0$ if and only if $[a] = [b]$ [42]. The *orimetric induced by the factorization and pruning structure* is $m_{\equiv,q} : S \times S \to \mathbb{R}_{\geq 0}$ with $m_{\equiv,q}(a,b) = q([a],[b])$.

The main finding is that these concepts are equivalent.

Theorem 4.3. *If (S, m) is an orimetric space, then $(S, \equiv_m, q_m)$ is a factorization and pruning structure. If $(S, \equiv, q)$ is a factorization and pruning structure, then $(S, m_{\equiv,q})$ is an orimetric space.*

We split the proof into two lemmas.

Lemma 4.4. *Let m be an orimetric on S. (i) $\equiv_m$ is an equivalence. (ii) The orimetric is invariant under the induced equivalence, $a_1 \equiv_m a_2$ and $b_1 \equiv_m b_2$ imply $m(a_1, b_1) = m(a_2, b_2)$, which means the induced quasimetric is well-defined. (iii) q_m is a quasimetric.*

Proof. (i) Reflexivity of the induced equivalence is by reflexivity for the orimetric. For symmetry, it suffices to have symmetry at zero. Transitivity is by the triangle inequality.

(ii) Recall that $a_1 \equiv_m a_2$ means $m(a_1, a_2) = 0 = m(a_2, a_1)$. We can therefore calculate as follows:

$$\begin{aligned} & m(a_1, b_1) \\ (\triangle\text{-inequality}) \quad & \leq m(a_1, a_2) + m(a_2, b_1) \\ (m(a_1, a_2) = 0) \quad & = m(a_2, b_1) \\ (\triangle\text{-inequality}) \quad & \leq m(a_2, b_2) + m(b_2, b_1) \\ (m(b_2, b_1) = 0) \quad & = m(a_2, b_2)\,. \end{aligned}$$

Repeating the argument with the indices swapped yields $m(a_2, b_2) \leq m(a_1, b_1)$. Together, the desired equality follows.

(iii) The triangle inequality immediately follows from the orimetric. For equality at zero, we have

$$q_m([a], [b]) = 0 \quad \Leftrightarrow \quad m(a, b) = 0 \quad \Leftrightarrow \quad a \equiv_m b \quad \Leftrightarrow \quad [a] = [b]\,. \qquad \square$$

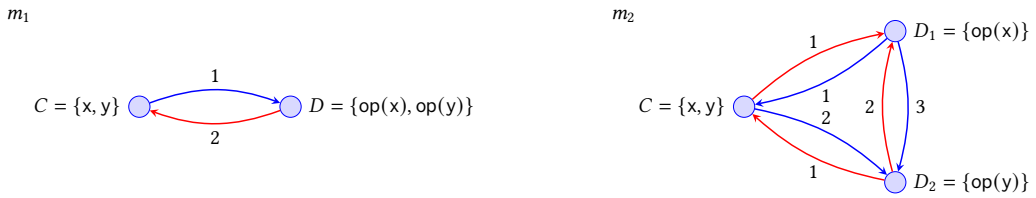
Lemma 4.5. *Let $(S, \equiv, q)$ be a factorization and pruning structure. Then $m_{\equiv,q}$ is an orimetric.*

Proof. Reflexivity and the triangle inequality carry over from the quasimetric. Symmetry at zero is a consequence of the equality at zero property of quasimetrics:

$$m_{\equiv,q}(a, b) = 0 \;\Rightarrow\; q([a], [b]) = 0 \;\Rightarrow\; [a] = [b] \;\Rightarrow\; q([b], [a]) = 0 \;\Rightarrow\; m_{\equiv,q}(b, a) = 0\,. \qquad \square$$

With this equivalence in place, we concentrate on orimetrics and illustrate the above definitions. We visualize an orimetric by a labeled and directed graph. The nodes are the classes in the induced equivalence. They are labeled by the elements they contain. The graphs are complete, meaning we have a directed edge between every pair of nodes. The edge is labeled by the distance between the nodes in the corresponding classes. Here we use Lemma 4.4 (ii). The orientation of the edges matters, which is what makes orimetrics more expressive than metrics. The triangle inequality says that one cannot reduce the distance from one class to another by traveling a detour.

Consider the set $R = \{\mathsf{x}, \mathsf{y}, \mathsf{op}(\mathsf{x}), \mathsf{op}(\mathsf{y})\}$. The following graphs define orimetrics m_1 and m_2 on R that will serve as running example:



The leftmost node in the graph defining m_1 represents the equivalence class $C = \{\mathsf{x}, \mathsf{y}\}$, which means $m_1(\mathsf{x}, \mathsf{y}) = 0$. The directed edge from C to the class $D = \{\mathsf{op}(\mathsf{x}), \mathsf{op}(\mathsf{y})\}$ is labeled by 1. This means $m_1(a, b) = 1$ for all $a \in C$ and $b \in D$. That this distance is the same for all elements in the two classes is Lemma 4.4 (ii). We do not have symmetry, the edge from D back to C has distance 2 instead of 1. The orimetric m_2 splits up the equivalence class D. Consider the blue edges. We have the distance $m_2(\mathsf{op}(\mathsf{x}), \mathsf{op}(\mathsf{y})) = 3$. The triangle inequality says that we cannot take a shortcut by traveling from $\mathsf{op}(\mathsf{x})$ to $\mathsf{op}(\mathsf{y})$ via C. If we assigned the blue edge from C to D_2 a distance of 1, the triangle inequality would fail and the graph would not represent an orimetric.

Orimetric m_1 is a congruence wrt. op, but m_2 is not. We have $\mathsf{x} \equiv_{m_2} \mathsf{y}$ but $\mathsf{op}(\mathsf{x}) \not\equiv_{m_2} \mathsf{op}(\mathsf{y})$. Consider the set $\{\mathsf{x}\}$. Neither m_1 nor m_2 is precise wrt. $\{\mathsf{x}\}$, because closing the set under the induced equivalence would add the element $\{\mathsf{y}\}$. Both orimetrics m_1 and m_2 are precise wrt. $\{\mathsf{op}(\mathsf{x}), \mathsf{op}(\mathsf{y})\}$, but only m_1 is unambiguous wrt. this set. Under m_2, the set falls apart into two equivalence classes.

We discuss the motivation behind the properties we require of orimetrics. Reflexivity is essential for the refinement. It says that a distance different from zero is enough to rule out a program as a candidate solution. Metrics and also the weaker pseudo-metrics are typically symmetric [42]. By avoiding symmetry, we can assign a meaningful distance to relations that are oriented, as explained in the overview. We need symmetry at zero for symmetry of the induced equivalence. The triangle inequality is important for transitivity of the induced equivalence and for Lemma 4.4 (ii). The three properties seem to be what is needed in the context of synthesis. We have not found this definition in the literature, and chose to name the object oriented metrics.

In the overview, we considered the ground truth to be a single function. This assumption is actually justified, but needs some discussion. For now, all we know is that GT is a set. The point is that an example-based specification defines the output only for some of the inputs, but leaves freedom for the remaining inputs. Consider now a candidate solution f and $gt_1, gt_2 \in GT$. It could be the case that $m(f, gt_1) = 0$ but $m(f, gt_2) \neq 0$. This means if we picked the wrong ground truth function, we would not be able to show that we solved the synthesis task. The problem disappears if we assume that the orimetric is *unambiguous* wrt. the ground truth. Not only will we have $m(f, gt_1) = 0$ if and only if $m(f, gt_2) = 0$. Lemma 4.4 (ii) even guarantees that the distance is the same, $m(f, gt_1) = m(f, gt_2)$, whether it is zero or not!

Remark. We expect all orimetrics we work with to be unambiguous wrt. the ground truth GT. With the above discussion, we can then concentrate on an arbitrary element $gt \in GT$.

4.2 Search Spaces and Enumeration Order

The two main parameters in bottom-up enumerative SyGuS solvers are the search space and the enumeration order. We now give them formal definitions.

Definition 4.6. A *search space* is a set of programs $P \subseteq \mathcal{L}(\mathcal{G})$. We call $\mathcal{L}(\mathcal{G})$ the full search space. The semantics of a program is a function from $\mathcal{F} = (Vars \to D) \to D$. If we have an orimetric m on this set $\mathcal{F}$, we speak of an *oriented metric search space*. We also write (P, m) to make both components explicit. Since we are interested in bottom-up enumerative solving, the search space should be *bottom-up enumerable*: for every program $\mathsf{p} \in P$ we should also find all subprograms in the set, $\mathsf{p}{\downarrow} \subseteq P$. We use $\mathsf{BU}(P)$ to denote the largest bottom-up enumerable set contained in P.

Lemma 4.7. *The largest bottom-up enumerable set contained in P is well-defined: the bottom-up enumerable sets are closed under arbitrary unions, and so $\mathsf{BU}(P) \subseteq P$ exists and is unique.*

To give an example, consider the set $R = \{\mathsf{x}, \mathsf{y}, \mathsf{op}(\mathsf{x}), \mathsf{op}(\mathsf{y})\}$ from above. This set is bottom-up enumerable, $\mathsf{BU}(R) = R$. If we remove the element y, we have $\mathsf{BU}(\{\mathsf{x}, \mathsf{op}(\mathsf{x}), \mathsf{op}(\mathsf{y})\}) = \{\mathsf{x}, \mathsf{op}(\mathsf{x})\}$. We fail to construct the program $\mathsf{op}(\mathsf{y})$ by bottom-up enumeration.

When is it sound to restrict the search to a subset of programs? The following definition gives a sufficient condition.

Definition 4.8. We call a search space Q *complete* wrt. another search space P and orimetric m, if for every program $\mathsf{p} \in P$ there is a program $\mathsf{q} \in Q$ with $[\![\mathsf{p}]\!] \equiv_m [\![\mathsf{q}]\!]$.

In Lemma 4.14, we give termination guarantees for working with complete search spaces. Pruning methods based on observational equivalence satisfy completeness. Also, the abstraction method in [47] is complete. We will deliberately use *incomplete* search spaces.

Definition 4.9. An *enumeration order* is a well-founded total order $\preceq \subseteq P \times P$ on the search space. It is a *bottom-up* enumeration, if it is compatible with the subprogram relation, for all $\mathsf{p}, \mathsf{q} \in P$, $\mathsf{p} \sqsubseteq \mathsf{q}$ implies $\mathsf{p} \preceq \mathsf{q}$. For the factorization, it will be important that the enumeration order is a

precongruence. For every operation op, say of arity 1, we expect that $\mathsf{p} \preceq \mathsf{q}$ implies $\mathsf{op}(\mathsf{p}) \preceq \mathsf{op}(\mathsf{q})$. The definition generalizes to higher arities.

We define three enumeration orders on the set R:

$$\mathsf{x} \preceq_0 \mathsf{op}(\mathsf{y}) \preceq_0 \mathsf{op}(\mathsf{x}) \preceq_0 \mathsf{y} \qquad \mathsf{x} \preceq_1 \mathsf{y} \preceq_1 \mathsf{op}(\mathsf{y}) \preceq_1 \mathsf{op}(\mathsf{x}) \qquad \mathsf{x} \preceq_2 \mathsf{op}(\mathsf{x}) \preceq_2 \mathsf{y} \preceq_2 \mathsf{op}(\mathsf{y})\ .$$

The enumeration order $\preceq_0$ is not bottom-up, because $\mathsf{op}(\mathsf{y}) \preceq_0 \mathsf{y}$. It is also not a precongruence, because $\mathsf{x} \preceq_0 \mathsf{y}$ but $\mathsf{op}(\mathsf{y}) \not\preceq_0 \mathsf{op}(\mathsf{x})$. The enumeration order $\preceq_1$ is bottom-up, but fails to be a precongruence for the same reason. The enumeration order $\preceq_2$ is bottom-up and a precongruence.

Given an enumeration order, we can construct (a subset of) the search space P algorithmically. We go through the programs as prescribed by the order, and only keep a program if the subprograms have already been listed. Let this procedure return the set $\mathsf{Enum}_{\preceq}(P)$. On the running example, this yields $\mathsf{Enum}_{\preceq_0}(R) = \{\mathsf{x}, \mathsf{op}(\mathsf{x}), \mathsf{y}\}$, $\mathsf{Enum}_{\preceq_1}(R) = R = \mathsf{Enum}_{\preceq_2}(R)$.

Lemma 4.10. *If $\preceq$ is bottom-up, then $\mathsf{Enum}_{\preceq}(P) = \mathsf{BU}(P)$.*

4.3 Components of the CEGAR Loop

`Prune`. The function computes a ball around the ground truth element we have chosen. We now define this ball. We again give the definition in the abstract for flexibility, meaning we consider an arbitrary set S with orimetric m. Let $gt \in S$ be the element that should serve as the center of the ball. Let $r \in \mathbb{R}_{\geq 0}$ be the radius. The *ball* and the *closed ball* around gt of radius r are defined by

$$\mathcal{B}_r(gt) \;=\; \{\ a \in S \mid m(a, gt) < r\ \} \qquad \text{resp.} \qquad \mathcal{B}_r[gt] \;=\; \{\ a \in S \mid m(a, gt) \leq r\ \}\ .$$

The ball is of course monotonic in the radius: the larger the more. To give an example, consider the set R with orimetric m_1 from above. Then $\mathcal{B}_0(\mathsf{x}) = \emptyset$, $\mathcal{B}_0[\mathsf{x}] = \{\mathsf{x}, \mathsf{y}\} = \mathcal{B}_1(\mathsf{x}) = \mathcal{B}_1[\mathsf{x}]$, and $\mathcal{B}_2[\mathsf{x}] = R$. Moreover, $\mathcal{B}_1(\mathsf{op}(\mathsf{x})) = \{\mathsf{op}(\mathsf{x}), \mathsf{op}(\mathsf{y})\}$ and $\mathcal{B}_1[\mathsf{op}(\mathsf{x})] = R$.

Function `Prune` constructs the elements in the ball bottom-up, using Lemma 4.10. This means the enumeration order should be bottom-up, and the result of the enumeration will not be the ball itself but $\mathsf{BU}(\mathcal{B}_r(gt))$, the largest bottom-up enumerable set that lives inside the ball.

`Factorize`. We factorize the search space $P = \mathsf{BU}(\mathcal{B}_r(gt))$ that we have just determined along the equivalence induced by the orimetric, $P/_{\equiv_m}$. This means programs are put into an equivalence class if their distance is zero. How do we represent the equivalence classes in a way that can be manipulated algorithmically? The idea is to use a representative system that only keeps the $\preceq$-least program from each equivalence class. Formally, the `Factorize` function is defined as follows:

$$\mathtt{Factorize}(P, m, \preceq) \;=\; \{\ \mathsf{p} \in P \mid \forall \mathsf{q} \in P : \mathsf{q} \prec \mathsf{p} \Rightarrow [\![\mathsf{p}]\!] \not\equiv_m [\![\mathsf{q}]\!]\ \}\ .$$

We return to the set $R = \{\mathsf{x}, \mathsf{y}, \mathsf{op}(\mathsf{x}), \mathsf{op}(\mathsf{y})\}$. Recall that we defined the orimetrics m_1 and m_2, but only the former is a congruence. We also have the enumeration orders $\preceq_1$ and $\preceq_2$, but only the latter is a precongruence. Now $\mathtt{Factorize}(R, m_1, \preceq_1) = \{\mathsf{x}, \mathsf{op}(\mathsf{y})\}$, $\mathtt{Factorize}(R, m_2, \preceq_2) = \{\mathsf{x}, \mathsf{op}(\mathsf{x}), \mathsf{op}(\mathsf{y})\}$, and $\mathtt{Factorize}(R, m_1, \preceq_2) = \{\mathsf{x}, \mathsf{op}(\mathsf{x})\}$. All sets are complete wrt. R and the given orimetric. However, only the last set is bottom-up enumerable. We see, the correct use of `Factorize` is intricate. If we use `Factorize` with an orimetric that is not a congruence or an enumeration order that is not a precongruence, we can lose bottom-up enumerability. The following lemma states sufficient conditions for `Factorize` to return a search space that is bottom-up enumerable.

Lemma 4.11. *(i) Let $\preceq$ be an enumeration order on P. Then $\mathtt{Factorize}(P, m, \preceq)$ is complete wrt. P under m. (ii) If P is bottom-up enumerable, $\preceq$ is additionally a precongruence, and m is a congruence, then $\mathtt{Factorize}(P, m, \preceq)$ is bottom-up enumerable.*

The examples we have just given show that Lemma 4.11 (ii) does not hold without the precongruence and the congruence requirements. The proof of the result can be found in the extended version [32]. For Lemma 4.11 (i), we use the well-foundedness of the enumeration order, which guarantees the existence of a least element in each equivalence class.

`Search`. The `Search` function iterates through the search space along the enumeration order until it finds the first program that, from the perspective of the orimetric, solves the synthesis task. It starts with the $\preceq$-minimal program $\mathsf{p} \in P$. Then it sets p to the successor of p until $[\![\mathsf{p}]\!] \equiv_m gt$ holds, upon which p is returned. To compute the successor of p, we use the procedure $\mathtt{Enum}_{\preceq}(P)$. Assuming that $\preceq$ is bottom-up, Lemma 4.10 shows that `Search` explores $\mathsf{BU}(P)$. Hence, to make sure we inspect the entire search space, P must be bottom-up enumerable, $\mathsf{BU}(P) = P$. This is where we will use Lemma 4.11(ii).

We illustrate how our theory of enumeration orders can be combined with deduction, more precisely, the deduction technique in DryadSynth [15]. Consider a program sketch s of arity $n+1$. If we find the last parameter of s that is needed to solve the synthesis task, then the instantiation of the sketch should be the next program to enumerate. To make this formal, assume we currently explore program p and we have already enumerated $\mathsf{p}_1, \ldots, \mathsf{p}_n \preceq \mathsf{p}$. If $\mathsf{q} = \mathsf{s}(\mathsf{p}_1, \ldots, \mathsf{p}, \ldots, \mathsf{p}_n)$ solves the synthesis task, $[\![\mathsf{q}]\!] = gt$, then it should be the immediate successor of p, meaning $\mathsf{q} = succ_{\preceq}(\mathsf{p})$. If we relax the requirement so that the filled sketch should be the successor of the last program of the same size as p, then we get the instantiations from Duet [27] and Simba [48]. Checking whether there are $\mathsf{p}_1, \ldots, \mathsf{p}_n \preceq \mathsf{p}$ so that $[\![\mathsf{q}]\!] = gt$ can be done efficiently for some operators. For the `xor` operator for example, only a lookup in a hashmap storing all programs with their values as keys is needed [15]. For the `ite` operator, a decision tree can be learned, as done in EUSolver [3].

This modification of the enumeration order may ruin the precongruence property, and one may be concerned that it jeopardizes the guarantees given by Lemma 4.11. The point is that we only see the subset of programs up to and including the first deduction step. On these programs, precongruence and the guarantees hold. Afterwards, `Search` terminates and so the guarantees are not needed for the remaining programs.

`Refine`. We leave the exact instantiation of the function to the user and present our instantiation in Section 5. Here, we only give two properties that the refinement scheme must satisfy. Let $m' = \mathtt{Refine}(m, \mathsf{p}, gt)$ be the refined orimetric. We want

$$\equiv_{m'} \subseteq \equiv_m \qquad \textit{and} \qquad [\![\mathsf{p}]\!] \not\equiv_{m'} gt\,. \tag{refinement}$$

The first property states that the new orimetric is more precise in that the induced equivalence relates fewer programs. The second says that the new orimetric differentiates between $[\![\mathsf{p}]\!]$ and gt.

Lemma 4.12 (Progress). *If function* `Refine` *guarantees the properties in* (refinement)*, then the algorithm will never explore the same program in two different loop iterations.*

`Learn`. The enumeration order is updated by `Learn` based on the programs seen in `Search`. Programs and operators which promise to be more valuable for a solution to gt are preferred in the updated enumeration order. The exact instantiation is left to the user, and we present our instantiation in Section 5.

Putting everything together, we obtain the above CEGAR loop. The following correctness guarantee immediately follows from the termination condition of the loop.

Lemma 4.13. *Consider the SyGuS problem $(\mathcal{G}, (I, O))$. If the CEGAR loop (Figure 1) terminates, it returns a program $\mathsf{p} \in \mathcal{L}(\mathcal{G})$ that satisfies the specification, $[\![\mathsf{p}]\!](I) = O$.*

We also have a termination guarantee. It puts together Lemmas 4.10, 4.11, and 4.4(ii).

Lemma 4.14. *Assume $(\mathcal{G}, (I, O))$ is solvable, $\preceq$ is bottom-up and a precongruence, m is a congruence and precise as well as unambiguous wrt. GT, and* `Prune` *returns a bottom-up enumerable search space that is complete wrt. $\mathcal{L}(\mathcal{G})$ and m. Then CEGAR will terminate in the first iteration.*

5 Contribution II – Constructing Oriented Metric Search Spaces

We define new orimetrics and explain how these orimetrics can be refined. Our focus will be on strings and bitvectors. We do not have new instantiations for the enumeration order, but work with a standard size-based definition. However, we enhance this order with new deduction strategies for strings and bitvectors. As it turns out, the definition of orimetrics and the design of deduction strategies are somewhat related, so we discuss deduction together with the orimetrics.

5.1 Lifting Oriented Metrics to Function Spaces

For SyGuS, we are interested in orimetrics on the function space $\mathcal{F} = (\mathit{Vars} \to D) \to D$. However, as we have seen in the overview, it is convenient to construct such orimetrics on functions by lifting orimetrics on the data domain. This lifting is actually the reason why we defined orimetrics for arbitrary sets. We now make the lifting explicit.

Let $\tilde{m}$ be an orimetric on the data domain D. Our goal is to lift $\tilde{m}$ to the function space $X \to D$. The set X should be understood as the set of (inputs from) all examples, although the definition does not rely on this understanding. We define the lifting parametric in a set $Y \subseteq X$. Intuitively, this is the subset of examples we currently consider, and the refinement loop will then make Y larger and larger. We define $m_Y : (X \to D) \times (X \to D) \to \mathbb{R}_{\geq 0}$ by

$$m_Y(f, g) \quad = \quad \sum_{y \in Y} \tilde{m}(f(y), g(y)) \; .$$

Lemma 5.1. *If $\tilde{m}$ is an orimetric on D, then m_Y is an orimetric on $X \to D$, for all $Y \subseteq X$.*

Besides the triangle inequality, a quasimetric satisfies $\tilde{m}(a, b) = 0$ if and only if $a = b$ [42]. When we lift from a quasimetric on the data domain using the set I of inputs from all examples, then the result will be precise and unambiguous wrt. GT.

Lemma 5.2. *For $\mathcal{J} \subseteq I$ and $\tilde{m}$ an orimetric on D, the orimetric $m_{\mathcal{J}}$ on $\mathcal{F} = (\mathit{Vars} \to D) \to D$ is unambiguous wrt. GT. If $\tilde{m}$ is a quasimetric and $\mathcal{J} = I$, the orimetric $m_{\mathcal{J}}$ is also precise wrt. GT.*

`Refine`. As explained in the overview, we use $m^\sharp = m_{\mathcal{J}}$ with $\mathcal{J} \subseteq I$ as our approximate orimetric. We refine the orimetric by adding an example for which the spurious program `p` does not return the expected value. Let $i \in I$ be an input example for which $[\![\mathsf{p}]\!](i) \neq gt(i)$ holds. We define $\mathtt{Refine}(m_{\mathcal{J}}, \mathsf{p}, gt) = m_{\mathcal{J} \cup \{i\}}$. This is a refinement indeed. Blaze's [47] abstraction refinement can also be instantiated in `Refine`. For this, we only need to update the abstraction α and the abstract transformers $f^\sharp$. Then we can calculate the orimetric as described in Section 2.2.

`Learn`. When $\mathtt{Learn}(\preceq, P, gt)$ is called, the spurious program `p` is $\preceq$-maximal in P. We use the precise metric m_I to judge all subprograms `q` of `p`. For every subprogram `q` whose semantics reside in $\mathcal{B}_{(\mathcal{F}, m_I), r}(gt)$, we update the enumeration order by pretending `q` has no children and size 1.

5.2 Recipe

All orimetrics we work with will be liftings from orimetrics on the data domain. We therefore only discuss orimetrics on strings and bitvectors, but not on functions over these data domains. Actually, the orimetrics on strings and bitvectors we define will be quasimetrics. This guarantees us preciseness of the orimetrics that result from lifting, Lemma 5.2. We stress, however, that the

lifted objects are no longer quasimetrics, they are orimetrics. The definitions in this section are guided by the following

> **Rule of Thumb:**
> If we understand deduction for the operators on the data domain, then we understand which values should be close in the orimetric for the data domain.

5.3 Oriented Metrics for Strings

For strings, we have three orimetrics, two of them defined with deduction in mind.

5.3.1 An Oriented Metric for `concat`. For deduction, we start from the sketch $\texttt{concat}(\texttt{p}, \texttt{p}')$ that consists of the concatenation operation. If p returns i and p′ returns i', then $\texttt{concat}(\texttt{p}, \texttt{p}')$ will return $o = i.i'$. Deduction infers missing arguments from already given arguments and target values. Concretely, if we want to obtain value o and we already have i, then the missing input must be i'. This has been captured by the inverse semantics in [27].

We define an orimetric that is optimized for concatenation. We think of the arguments of the orimetric as the values that are given to the deduction engine, namely the output o and the input i that is already known. The value $m_{\texttt{concat}}(i, o)$ should then capture how far i is away from o when concatenation is applied next. This amounts to judging what could be called the complexity of the inverse semantics. How many values does the inverse semantics contain, and how easy is it to generate these values. The distance should then be inverse proportional to this complexity. In particular, when there are no values left in the inverse semantics, the distance should be infinity. For concatenation, this would be the case when i is not a prefix or a suffix of o.

While these considerations are useful as guidance, implementing them strictly does not lead to a useful definition. First, we want to determine the ball not only for the concatenation operation, but it should also work for the remaining operations. Second, trying to define the complexity of the inverse semantics quickly gets out of hand mathematically. How do we make sure the definition still satisfies the triangle inequality?

Our definition relaxes the requirement that the input should be a prefix or a suffix to just an infix. We approximate the complexity of the inverse semantics as follows. The number of elements in the inverse semantics is always one, we are just missing one string. The string, however, is easier to generate the shorter it is. With this, we use the orimetric

$$m_{\texttt{concat}}(i, o) \;=\; \begin{cases} len(o) - len(i) & \text{if } i \text{ is an infix of } o \\ c + |len(o) - len(i)| & \text{otherwise .} \end{cases}$$

We select the constant $c \in \mathbb{R}$ to be larger than the sum over the lengths of all output values in the examples, and use this constant to define the ball. Alternatively, we could extend the reals with infinity and define $m_{\texttt{concat}}(i, o) = \infty$, if i is not an infix of o. However, this would add infinity as an undesirable technicality to many places.

5.3.2 An Oriented Metric for `substr`. The second sketch our deduction should support consists of the substring operation. It takes three arguments: the original string i, the starting index of the substring that should be determined as the output o, and the maximal length of that substring. Our orimetric focuses on the relationship between i and o. The only precondition we can derive is that the input must be a superstring of the output. We define $m_{\texttt{substr}}$ just like $m_{\texttt{concat}}$, but with the infix requirement flipped. Actually, within Merlin we have implemented a full new deduction strategy for the substring operation. It can be found in the extended version [32].

5.3.3 Levenshtein. Our synthesizer runs with a portfolio of different orimetrics, and it has turned out beneficial to have a fallback that is not optimized for a deduction strategy. We use the Levenshtein

distance for this purpose [29]. The Levenshtein distance between two strings is the number of substitutions, deletions, and insertions that is necessary to convert one string into the other. To efficiently check whether a program lies inside a ball, we use Levenshtein automata [39]. A Levenshtein automaton for a target string o and a radius r is a finite automaton that accepts all strings i with $lvst(i, o) < r$. In our implementation, we set the maximum radius to $r = 4$.

5.4 Oriented Metrics for Bitvectors

For bitvectors, we have three orimetrics tailored towards sketches, and one additional orimetric.

5.4.1 An Oriented Metric for and. We consider the bitwise conjunction $\mathsf{and}(\mathsf{p}, \mathsf{p}')$. It takes as input bitvectors i and i' and produces the bitvector $o = i \& i'$. We again derive an orimetric by working out a deduction method. The deduction method reasons over the output o and the input i that is already known. The first step is to check whether o is bitwise smaller than i, denoted by $o \sqsubseteq i$. If the check fails, the deduction aborts. If the check succeeds, we calculate the requirements on i', for each bit b, as follows. (R1) If $o[b] = 1$, then $i'[b] = 1$. (R2) If $o[b] = 0$ and $i[b] = 1$, then $i'[b] = 0$. There are no requirements if $o[b] = 0$ and $i[b] = 0$.

The goal of the orimetric $m_{\mathsf{and}}(i, o)$ is to estimate the complexity of the inverse semantics. The idea is to sum up all situations (R1) and (R2) in which the inverse semantics has no choice for the value of a bit: $\sum_{b\in[0,63]} i[b] \vee o[b]$. This function, however, will not be reflexive. If $i = o = 1^{64}$, it will return 64 rather than 0. The solution is to drop requirement (R1). The idea is to approximate the search for i' from above. We take for granted that we are in a space of values that are bitwise larger than the output. Only summing up situations (R2) then leads to the final definition:

$$m_{\mathsf{and}}(i, o) \;=\; \begin{cases} \sum_{b\in[0,63]} \neg o[b] \wedge i[b] & \text{if } o \sqsubseteq i \\ c & \text{otherwise .} \end{cases}$$

The large constant is again for emptiness of the inverse semantics. It is worth noting that under the assumption $o \sqsubseteq i$, the sum $\sum_{b\in[0,63]} \neg o[b] \wedge i[b]$ is just the Hamming distance. The Hamming distance is even a metric. A similar reasoning yields an orimetric for bitwise disjunction.

5.4.2 An Oriented Metric for mul. For the multiplication among bitvectors $\mathtt{mul}(\mathsf{p}, \mathsf{p}')$, deduction has to start from o and i and determine i' so that $o = i \times i'$. This amounts to finding the multiplicative inverse in the integer ring represented by the bitvectors. Note that the inverse does not exist if the output is smaller than the input.

The corresponding orimetric again approximates the complexity of the inverse semantics, here the multiplicative inverse element. We take this complexity to be the difference in the number of leading zeros between input and output:

$$m_{\mathtt{mul}}(i, o) \;=\; \begin{cases} 0 & \text{if } i = o \\ 1 + nlz(o) - nlz(i) & \text{if } nlz(i) \leq nlz(o) \\ c & \text{otherwise .} \end{cases}$$

5.4.3 Hamming. To relate two values directly, we use a modification of the Hamming distance. The Hamming Distance between two bitvectors is the number of bits where the two bitvectors do not match. Taking *HDist* directly as the quasimetric prunes programs that are good. Consider a 64-bit bitvector i with $HDist(i, o) = 64$. A single not operation would yield o. With that in mind, we define a new quasimetric m_{hd} that relates two bitvectors a and b as follows:

$$m_{hd}(a, b) = \begin{cases} HDist(a, b) & , HDist(a, b) \leq 32 \\ 64 - HDist(a, b) + 1 & , otherwise . \end{cases}$$

5.5 Hacks

Sharing Programs Between Concurrent Solver Instances. In our implementation, we run several solvers concurrently that utilize different orimetrics to prune the search space. When a solver with orimetric m finds a candidate solution, we also analyze it in the context of the other orimetrics m'. The goal is to find subprograms that are valuable wrt. m', and modify the enumeration order of the corresponding solver accordingly. This way, valuable programs can be shared between the concurrent solvers.

Keeping Programs up to a Size Threshold. We do not want to apply pruning to small programs, otherwise the bottom-up enumerable portion of the ball may become too small to be useful. We slightly modify the given orimetric m to create m', which keeps programs of size up to a threshold s in the open ball of radius r.

6 Contribution III – Implementation and Evaluation

We implemented our approach in a SyGuS solver called Merlin. It is written in C++ and uses Z3 [14] as an SMT solver backend for the CEGIS loop. We evaluate the performance of Merlin to answer the following research questions:

Q1: How does Merlin perform on SyGuS tasks of a variety of domains?
Q2: How does Merlin compare against state-of-the-art SyGuS and domain-specific solvers?
Q3: What is the effect of pruning using orimetrics?
Q4: Does abstraction and learning from spurious programs enhance performance?
Q5: What impact does the radius of the ball have?
Q6: Which orimetrics are successful?
Q7: What is the benefit of running multiple instances with different orimetrics concurrently?

We ran all experiments on an Apple M3 Max with 64 GB of RAM and used a 10-minute timeout.

6.1 Implementation Details

For strings, we use a size threshold s of 3 and start with orimetrics that consider one example only. To implement Levenshtein automata, we use the Mata finite automaton library [10]. For Bitvectors, we use a size threshold s of 7 and start with orimetrics that consider two examples. We concurrently run several instances of the solver using different orimetrics. We also run an instance that does not prune and will refer to it as m_∞. For each orimetric designed for a specific sketch, the solver only uses deduction for this sketch. The other threads apply deduction on sketches for which we did not design a specific orimetric, e.g. the `add(?, ?)` sketch. That means, for each benchmark featuring strings (bitvectors), Merlin runs solver instances for all string (bitvector) orimetrics in a portfolio. In particular, even when a benchmark features multiple oriented operations, e.g. `concat` and `substr`, Merlin runs four threads each employing one of $m_{\texttt{concat}}$, $m_{\texttt{substr}}$, *lvst*, or m_∞. If the learning mechanism finds a program that is valuable using a specific orimetric, we only change the enumeration order of the thread employing this orimetric and of the thread that does not prune.

The choice of the radius depends on the evaluation. For the comparison with other tools, we conduct experiments with varying radii for *lvst* and m_{hd}. For the remaining orimetrics, we use c, which means we evaluate the given conditions and prune if they fail. For the ablation studies, we use varying radii for all orimetrics.

6.2 Setup

Benchmarks. We use benchmarks from three domains: SyGuS bitvector benchmarks without conditionals, SyGuS string benchmarks without conditionals, and the Blaze string benchmark set.

In the Bitvector domain, we have 549 benchmarks: We include 44 Hacker's delight [26] benchmarks from the SyGuS competition suite and 5 additional Hacker's Delight benchmarks from Probe. The specification of these benchmarks is not in the form of examples, thus, we use a CEGIS loop. The CEGIS loop introduces non-determinism. Therefore, we ran the benchmarks 3 times and report the mean of the results. We also include the 500 deobfuscation benchmarks from Simba [48]. Here, the specification is given in the form of input-output examples.

In the String domain, we use 181 tasks from Duet [27] that return a string. These include 108 benchmarks from the SyGuS competition, 32 benchmarks designed from Stack Overflow questions, and 41 benchmarks designed from Exceljet articles. We also use 108 benchmarks from the Blaze string benchmark set [47]. Blaze uses a custom DSL which is not in the SyGuS format.

Since we did not implement the case-splitting deduction methods from EUSolver [3], we omit this class of benchmarks. However, it is well-understood how to handle conditionals, namely with a decision tree construction. This was first proposed by EUSolver [3] and adapted by several other SyGuS solvers. Moreover, we stress that the decision tree construction *also fits into our framework*: it is another method of deduction that changes the enumeration order. To allow for a fair comparison of the core solving strategy, we remove conditionals from the String benchmark set and from 5 benchmarks of the benchmark set Hacker's Delight. The other benchmarks of the benchmark set Hacker's Delight, as well as the Blaze and Deobfuscation benchmark sets do not have conditionals.

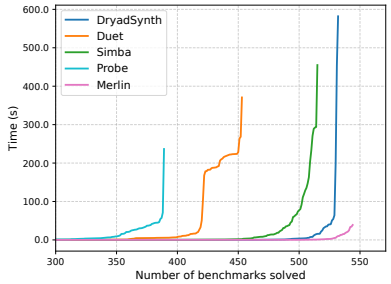
Operators Featured in the Benchmarks. Apart from two benchmarks of the Hacker's Delight benchmark set, *all* benchmarks use oriented operators for which we defined orimetrics. We give a detailed overview on the operators in each benchmark set. The benchmark set String has the following string operators: `str.++` (concatenation), `str.replace`, `str.at` (selecting a character by index), `int.to.str` (converting an integer to a string), and `str.substr`. The Blaze benchmarks use the following string operators: `Concat` and `SubStr`. The benchmark set Deobfuscation uses the following bitvector operators: `bvnot` (flips all bits), `bvxor` (bitwise xor), `bvand` (bitwise and), `bvor` (bitwise or), `bvneg` (negation), `bvadd` (addition), `bvmul` (multiplication), and `bvsub` (subtraction). The Hacker's Delight benchmark set has three categories that reflect the difficulty of the benchmarks. The most difficult benchmarks contain all of the following operators: the operators from Deobfuscation, `bvudiv` (unsigned division), `bvurem` (unsigned remainder), `bvlshr` (logical shift right), `bvashr` (arithmetic shift right), `bvshl` (shift left), `bvsdiv` (signed division), and `bvsrem` (signed remainder). The category with medium difficulty typically features 5-10 operators from above. The simplest benchmarks have 2-4 of the above operators.

Baseline Solvers. We compare Merlin against the general SyGuS tools Probe [6] and Duet [27]. In the bitvector domain, we additionally compare Merlin against Simba [48] and DryadSynth [15]. In the string domain, we add the recent Synthphonia [16] tool for comparison. For the Blaze benchmarks, we only compare against Blaze [47] since their DSL is not directly compatible with the other solvers. Section 2.2 offers a description on how all mentioned tools work. For DryadSynth, we did not enable the ChatGPT feature, which configures an initial enumeration order using ChatGPT.

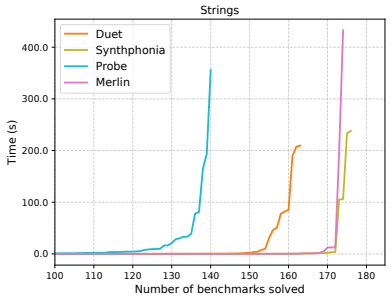
6.3 Effectiveness of Merlin

To answer **Q1** and **Q2**, we evaluate Merlin on all benchmarks.

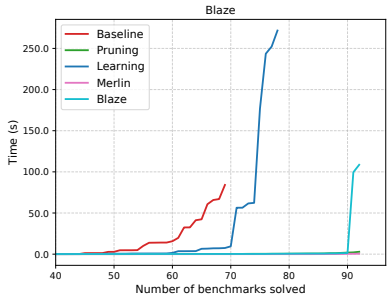
For the bitvector domain, Figure 6a summarizes the solving times of each solver. Note, that the x-axis starts at 300. The benchmarks before take negligible time. Probe solves 389 benchmarks. Duet solves 453 benchmarks, Simba finds solutions for 515 benchmarks, and DryadSynth solves 532 benchmarks. Lastly, our new tool, Merlin, solves 544 of the total 549 bitvector benchmarks. If we compare the running time of DryadSynth and Merlin on all benchmarks where at least one tool has a solution, Merlin is 27 times faster than DryadSynth.



(a) Time comparison for the bitvector domain.



(b) Time comparison for the string domain.



(c) Time comparison for the Blaze benchmarks.

Benchmark, r	#Bench	Time	$\lvert P\rvert$
Blaze, 25	48 (4)	55.00	4126
Blaze, 50	59 (8)	58.51	8264
Blaze, 75	81 (17)	68.42	39283
Blaze, 100	89 (23)	16.72	36433
Deobfusc, 25	275 (0)	4.32	42
Deobfusc, 50	288 (1)	8.00	60
Deobfusc, 75	322 (1)	4.11	24
Deobfusc, 100	324 (0)	3.37	6
Strings, 25	130 (4)	2.32	2
Strings, 50	119 (2)	2.12	2
Strings, 75	116 (6)	45.46	2
Strings, 100	108 (2)	16.01	3

(d) Ablation studies for different r.

Fig. 6. Solving time for all benchmarks and ablation studies for different r.

We want to note again that we tested DryadSynth without its ChatGPT feature. This feature applies to the 49 Hacker's Delight problems. While the running time reported in their paper [15] with ChatGPT enabled is comparable to the running time without it in most benchmarks, there are 4 benchmarks where the ChatGPT feature has a great effect: DryadSynth is able to solve these only with ChatGPT. Merlin is not able to solve these benchmarks. However, we stress that the ChatGPT feature can be seen as a different initial enumeration order.

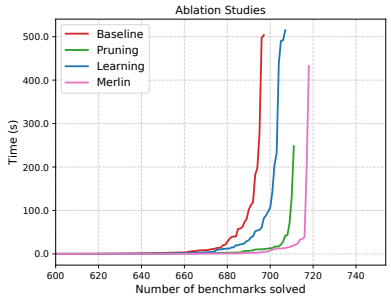
For the string domain, Figure 6b summarizes the solving times for each solver. Again, note that the x-axis starts at 100. Probe solves 140 benchmarks. Duet solves 163 benchmarks. Merlin can almost compete with the newly proposed, domain-specific string synthesis tool Synthphonia: Merlin solves 174 benchmarks while Synthphonia finds a solution for 176 benchmarks.

For the benchmarks from Blaze, Figure 6c summarizes the solving times for Blaze and Merlin. Again, note that the x-axis starts at 40. Both tools solve 92 benchmarks and have nearly the same running time across the majority of benchmarks. Still, Merlin is the fastest on all benchmarks. In the hardest benchmarks, Merlin is vastly superior. Overall, Merlin is 75 times faster than Blaze.

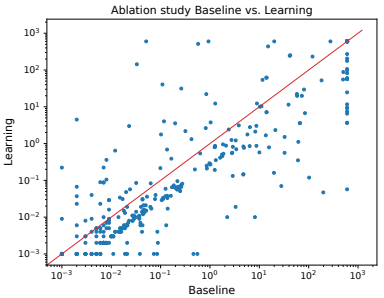
6.4 Ablation Studies

To answer **Q3** and **Q4**, we now evaluate the effect of pruning and learning on the synthesis performance. For this, we implemented three additional different versions of Merlin:

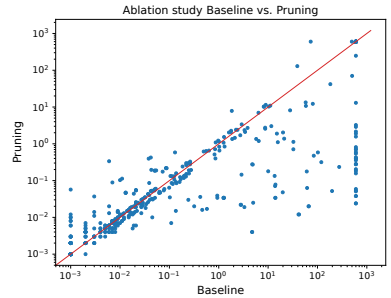
- Pruning: An implementation only using pruning. Here, learning and abstraction is disabled, and therefore we do not use `Refine` and `Learn` functions.
- Learning: An implementation that uses the orimetrics only for abstraction and learning.



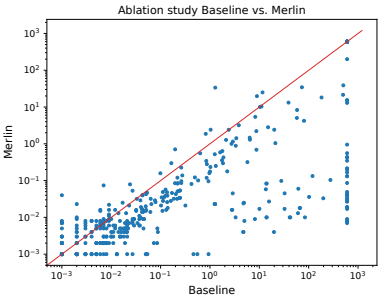
(a) Solving time across all 730 SyGuS benchmarks.



(b) Scatter plot Baseline vs Learning.



(c) Scatter plot Baseline vs Pruning.



(d) Scatter plot Baseline vs Merlin.

Fig. 7. Ablation Studies.

- Baseline: This implementation uses neither the pruning nor the learning features.

All implementations use the same deduction methods and have the same initial enumeration order. Moreover, all implementations at least factorize the search space using observational equivalence.

Figure 7 summarizes the ablation studies for the SyGuS benchmarks. Figure 7a shows the running time across all SyGuS benchmarks. Note, that the x-axis starts at 600. The benchmarks before take negligible time. The Baseline solver solves 697 benchmarks and is the slowest overall. The Pruning solver solves 711, and the Learning solver solves 707 benchmarks. Merlin solves 718 benchmarks and is the fastest overall.

Figure 6c shows the running time across the Blaze benchmarks. Again, note that the x-axis starts at 40. Baseline solves 69 benchmarks and is the slowest overall, while Learning solves 78 benchmarks. Pruning and Merlin both solve 92 benchmarks. Remarkable is that our abstraction alone, represented by Learning, is inferior to the abstraction of Blaze. However, our pruning method is so powerful that it compensates for the quality of our abstraction and even surpasses Blaze's performance. In future work we want to adapt Blaze's abstraction.

Figures 7b to 7d compare Baseline against Learning, Pruning, and Merlin in a scatter plot. The axes show the solving time in seconds. Note that the axes are in log-scale. Figure 7b compares Baseline against Learning. It shows that our learning method has a positive effect on the majority of benchmarks. However, there are also benchmarks where learning negatively impacts performance. Since the enumeration order is altered, learning might shift the solution to a later point in the enumeration order. The positive effect is two-fold. First, due to the approximate orimetrics, factorization has a greater effect. Second, learning from previously solved examples enables solving some

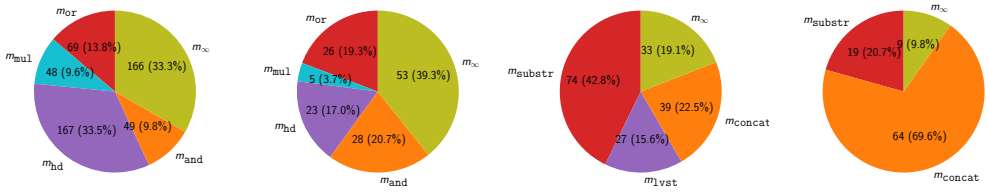
complex benchmarks where Baseline times out. Figure 7b compares Baseline against Pruning. For trivial benchmarks, the baseline solver outperforms the solver that uses pruning. The reason may be the overhead due to concurrency. For more complex benchmarks, Baseline takes significantly longer or even times out. This behavior is expected because we only start pruning the search space after a certain size threshold. For benchmarks that can be solved below or just above the threshold, the search space is not pruned enough to observe a noticeable improvement.

Figure 7d compares Baseline against Merlin. Combining learning and pruning in Merlin results in a significant speed up on almost all instances. Merlin also solves more instances than Pruning and Learning. This shows that learning and pruning both have a significant impact on their own and can be combined to provide an even greater benefit. The negative impact of learning on some instances that we observed in Figure 7b is also averted. This is because when using multiple threads, we are less likely to learn a bad program: We only update the enumeration order of the thread with the corresponding orimetric. Thus, the enumeration order of the other threads is not impacted. In the Learning implementation, we only had one thread and one enumeration order; we were at risk of learning a bad program. If we compare the total running time of Baseline and Merlin on all benchmarks where at least one tool has a solution, Merlin is 42 times faster than Baseline.

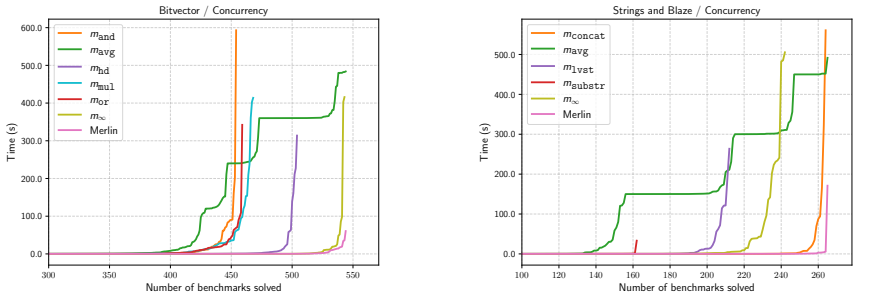
To answer **Q5**, we conduct experiments on the deobfuscation, strings, and Blaze benchmarks. We did not see a great effect of pruning on the Hacker's Delight benchmarks, thus we omit these here. We take 100, 75, 50, and 25 percent of the maximum distance as the radius for each orimetric. By maximum distance, we refer to the maximum distance an orimetric can return given an output o, excluding c. For example, if o is a 4-bit bitvector with two bits set, the maximum distance excluding c that m_{and} can return is 2. With several outputs, we take the sum. For the orimetric $m_{\texttt{substr}}$, we use discrete pruning at 100%. Since for the orimetrics $m_{\texttt{concat}}$ and $m_{\texttt{mul}}$ values smaller than 50% do not make sense, we use 100, 87.5, 75, and 62.5 percent instead. For the orimetric *lvst*, 100% corresponds to allowing a distance of 4 for each output, 75% corresponds to 3, and so on. Figure 6d compares Baseline with instances of Pruning. The second column shows the number of benchmarks that were solved using an underapproximation thread. In parentheses is the number of these benchmarks, that the baseline solver did not solve. The third column describes the gained speed-up factor on the benchmarks that were solved by an underapproximation thread and by the baseline. The last column compares the size of the explored search space. For example, the first row states that when instantiating Merlin with a 25% radius and running the Blaze benchmarks, 48 benchmarks were solved by a thread that prunes the search space. 4 of these benchmarks were not solved by the baseline solver. On the benchmarks that were also solved by the baseline solver, the 25% instantiation was 55 times faster and considered 4126 times fewer programs than the baseline solver. The Blaze benchmarks show the biggest gain in performance. This is not surprising because the top operator of the grammar is concatenation. Thus, concentrating on substrings is a very good heuristic. There also seems to be a sweet spot for taking 75% of the maximum radius, although on the blaze benchmarks this is still too coarse for many benchmarks. A prime example is the following benchmark: Convert the string `"Launa Withers"` into `"L. Withers"`. Extracting the `"L"` is essential for this benchmark, but the output falls outside the 75% radius. We leave fine-tuning the radius up to future work.

6.5 Suitability of the Presented Orimetrics and Impact of Concurrency

To answer **Q6**, we analyze *how* Merlin solved the benchmarks. Figures 8a to 8d show for how many benchmarks a solver instance was the fastest. For example, in Deobfuscation (Figure 8a), the pie chart says that in 49 benchmarks, which make up 9.8% of the solved instances of the benchmark set, the m_{and} solver was faster than the solvers with the other orimetrics. The entries m_∞ refer to the solver instance that does not prune. All in all, the orimetric solvers were successful on the



(a) Deobfuscation. (b) Hacker's Delight. (c) String. (d) Blaze.



(e) Time comparison for the bitvector domain. (f) Time comparison for the string domain.

Fig. 8. Suitability of presented orimetrics and impact of concurrency.

majority of benchmarks, although they prune away a part of the search space. The results are better for the string than for the bitvector benchmarks. For bitvectors (Figures 8a and 8b), the success of the m_{∞} solver indicates that pruning may have a negative impact, and there is room for better orimetrics. Furthermore, it very much depends on the benchmark set which orimetric will fit. For example, the orimetric $m_{\texttt{mul}}$ fits better the deobfuscation benchmark set than Hacker's Delight. The orimetrics designed for strings cover a broader spectrum of the benchmarks. Here, only 19% of the String benchmark set and 10% of the Blaze benchmark set were solved fastest by the m_{∞} instance. Remarkably, the Levenshtein distance was less suitable for the String and Blaze benchmarks than the Hamming distance was for the bitvector benchmarks.

Next, we study the impact of concurrency. Before execution, we do not know which orimetric will prune the search space best. For this reason, we run the solver instances in parallel. Figures 8e and 8f show the runtime for each instance in isolation, and for Merlin. Here, m_{avg} represents the average runtime across the instances. For bitvectors, no instance performs as well as Merlin. This means committing to one orimetric does not work well. Picking an orimetric at random will, on average, result in m_{avg}. While this random choice solves almost as many benchmarks as Merlin, the runtime is 115x slower. Always using m_{∞} is 6x slower than Merlin. For strings, the $m_{\texttt{concat}}$ instance comes closest to Merlin while being 10x slower. To answer **Q7**, concurrently running multiple solver instances employing different orimetrics is crucial for Merlin's performance.

7 Related Work

Metrics in Synthesis. SyMetric [21] uses distance metrics for approximate observational equivalence. They cluster programs whose output is within a given radius inside an equivalence class. SyMetric also prunes the search space using a ball around *gt*. Their resulting equivalence is not a congruence. They rely on similarity between the inputs and the outputs for most operators. Using this similarity, they are able to repair a spurious program given a predefined set of rewrite rules.

Syntia [8] incorporates metrics in top-down search. For a given sketch, they randomly fill its holes and execute the resulting program. Using metrics they assign each sketch a score based on similarity of the program's output and the specified output. This score is then used to guide search.

Pruning and Factorization. Most tools employ some form of OE factorization [1, 43]. For this, Version Spaces Algebras are used in FlashFill [9, 22, 37] and finite tree automata are used by Dace [46]. Absynthe [23] enables the use of OE for programs with local variables. Morpheus [19] and Neo [18] use logical reasoning to prune parts of the search space for which they can prove that it does not contain a solution. For this, Morpheus requires over-approximative specifications of program components. It then generates sketches and uses an SMT solver to discard infeasible program sketches. Neo extends this approach with conflict driven learning. Having ruled out one sketch, it extracts the root cause of its infeasibility. Then other sketches for which this root cause applies can also be pruned. FlashFill++[9] proposes cuts to enable middle-out synthesis. A cut creates a set of subproblems whose solutions can be combined to solve the whole synthesis problem. This set of subproblems may be incomplete, i.e. not every possible way to dissect the synthesis problem is explored. This effectively prunes the search space.

Approximation. To further prune the search space, a common approach is to use abstractions or type information to represent programs. With this, one can decide the feasibility of sketches in order to prune infeasible ones. NOSDAQ [30] synthesizes database queries. Through abstraction of database collections, they can prove that a sketch cannot be completed to yield the correct output. Similarly, Synquid [36], its extension [17], and λ^2 [20] use type information to prune infeasible partial programs. Scythe [44] first searches for sketches that satisfy the specification in the abstract setting. Then, it uses this set of sketches to search for an instantiation satisfying the specification in the concrete setting. Absynthe [23] is similar in that it uses user-defined abstract semantics to create viable sketches. For each a viable sketch, it fills the holes and executes the concrete program using a given interpreter. If the program satisfies the specification, it is returned as the solution.

Learning. Similar to DryadSynth [15], HySynth [5] and DeepCoder [4] use Large Language Models to instantiate an enumeration order. LaSy [34] and ReGuS [12] blend upfront and just-in-time learning. They solve a suite of related synthesis tasks. The solutions for easier tasks can then be used as a callable component to solve harder tasks. In our framework, we can simulate these learning approaches by appropriately setting the initial enumeration order. Atlas [45] extends Blaze [47] by learning useful predicates for abstraction from a set of training problems upfront. As we have shown for Blaze, predicate abstraction can be seen as an instantiation of oriented metrics. Bester [33] motivates just-in-time learning by showing that for complex benchmarks, partial solutions are often part of the intended solution. FrAngel [40] modifies partial solutions to solve the synthesis problem.

8 Conclusion and Future Work

We presented oriented metrics as a foundation for pruning, factorization, refinement, and learning in syntax-guided synthesis. We defined a synthesis algorithm that has these features and is parametric in the enumeration order and the initial orimetric. We invented new orimetrics for the string and the bitvector domain that occur frequently in SyGuS problems. We implemented our approach in a tool called Merlin, and obtained a speed-up of an order of magnitude compared to the state-of-the-art. In the future, we will further explore the design of orimetrics and the choice of when to prune.

Data-Availability Statement

A software artifact to reproduce the experiments is available on Zenodo [31].

Acknowledgments

We would like to thank the POPL'26 reviewers for their valuable comments and suggestions.

References

[1] Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. 2013. Recursive Program Synthesis. In *CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 8044)*, Natasha Sharygina and Helmut Veith (Eds.). Springer, 934–950. doi:10.1007/978-3-642-39799-8_67

[2] Rajeev Alur, Rastislav Bodík, Eric Dallal, Dana Fisman, Pranav Garg, Garvit Juniwal, Hadas Kress-Gazit, P. Madhusudan, Milo M. K. Martin, Mukund Raghothaman, Shambwaditya Saha, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2015. Syntax-Guided Synthesis. In *Dependable Software Systems Engineering*, Maximilian Irlbeck, Doron A. Peled, and Alexander Pretschner (Eds.). NATO Science for Peace and Security Series, D: Information and Communication Security, Vol. 40. IOS Press, 1–25. doi:10.3233/978-1-61499-495-4-1

[3] Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. 2017. Scaling Enumerative Program Synthesis via Divide and Conquer. 10205, TACAS (2017), 319–336. doi:10.1007/978-3-662-54577-5_18

[4] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2017. DeepCoder: Learning to Write Programs. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=ByldLrqlx

[5] Shraddha Barke, Emmanuel Anaya Gonzalez, Saketh Ram Kasibatla, Taylor Berg-Kirkpatrick, and Nadia Polikarpova. 2024. HYSYNTH: Context-Free LLM Approximation for Guiding Program Synthesis. In *NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, Amir Globersons, Lester Mackey, Danielle Belgrave, Angela Fan, Ulrich Paquet, Jakub M. Tomczak, and Cheng Zhang (Eds.). http://papers.nips.cc/paper_files/paper/2024/hash/1c9c85bae6161d52182d0fe2f3640512-Abstract-Conference.html

[6] Shraddha Barke, Hila Peleg, and Nadia Polikarpova. 2020. Just-in-time learning for bottom-up enumerative synthesis. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 227:1–227:29. doi:10.1145/3428295

[7] Clark Barrett, Aaron Stump, and Cesare Tinelli. 2010. The SMT-LIB Standard: Version 2.0. In *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*, A. Gupta and D. Kroening (Eds.).

[8] Tim Blazytko, Moritz Contag, Cornelius Aschermann, and Thorsten Holz. 2017. Syntia: Synthesizing the Semantics of Obfuscated Code. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, Engin Kirda and Thomas Ristenpart (Eds.). USENIX Association, 643–659. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/blazytko

[9] José Cambronero, Sumit Gulwani, Vu Le, Daniel Perelman, Arjun Radhakrishna, Clint Simon, and Ashish Tiwari. 2023. FlashFill++: Scaling Programming by Example by Cutting to the Chase. *Proc. ACM Program. Lang.* 7, POPL (2023), 952–981. doi:10.1145/3571226

[10] David Chocholatý, Tomás Fiedor, Vojtech Havlena, Lukás Holík, Martin Hruska, Ondrej Lengál, and Juraj Síc. 2024. Mata: A Fast and Simple Finite Automata Library. In *TACAS 2024 (LNCS, Vol. 14571)*, Bernd Finkbeiner and Laura Kovács (Eds.). Springer, 130–151. doi:10.1007/978-3-031-57249-4_7

[11] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem (Eds.). 2018. *Handbook of Model Checking*. Springer. doi:10.1007/978-3-319-10575-8

[12] Guofeng Cui, Yuning Wang, Wenjie Qiu, and He Zhu. 2024. Reward-Guided Synthesis of Intelligent Agents with Control Structures. *Proc. ACM Program. Lang.* 8, PLDI (2024), 1730–1754. doi:10.1145/3656447

[13] Robin David, Luigi Coniglio, and Mariano Ceccato. 2020. QSynth - A Program Synthesis based approach for Binary Code Deobfuscation. In *Binary Analysis Workshop (BAR) 2020*. doi:10.14722/bar.2020.23009

[14] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS 2008*. Springer, 337–340. doi:10.1007/978-3-540-78800-3_24

[15] Yuantian Ding and Xiaokang Qiu. 2024. Enhanced Enumeration Techniques for Syntax-Guided Synthesis of Bit-Vector Manipulations. *Proc. ACM Program. Lang.* 8, POPL (2024), 2129–2159. doi:10.1145/3632913

[16] Yuantian Ding and Xiaokang Qiu. 2025. A Concurrent Approach to String Transformation Synthesis. *Proc. ACM Program. Lang.* PLDI (2025). doi:10.1145/3729336

[17] Shingo Eguchi, Naoki Kobayashi, and Takeshi Tsukada. 2018. Automated Synthesis of Functional Programs with Auxiliary Functions. In *APLAS 2018, Wellington, New Zealand, December 2-6, 2018, Proceedings (LNCS, Vol. 11275)*, Sukyoung Ryu (Ed.). Springer, 223–241. doi:10.1007/978-3-030-02768-1_13

[18] Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. 2018. Program synthesis using conflict-driven learning. In *PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 420–435. doi:10.1145/3192366.3192382

[19] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017. Component-based synthesis of table consolidation and transformation tasks from examples. In *PLDI 2017, Barcelona, Spain, June 18-23, 2017*, Albert

Cohen and Martin T. Vechev (Eds.). ACM, 422–436. doi:10.1145/3062341.3062351

[20] John K. Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing data structure transformations from input-output examples. In *PLDI 2015, Portland, OR, USA, June 15-17, 2015*, David Grove and Stephen M. Blackburn (Eds.). ACM, 229–239. doi:10.1145/2737924.2737977

[21] John K. Feser, Isil Dillig, and Armando Solar-Lezama. 2023. Inductive Program Synthesis Guided by Observational Program Similarity. *Proc. ACM Program. Lang.* 7, OOPSLA2 (2023), 912–940. doi:10.1145/3622830

[22] Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. In *POPL 2011, Austin, TX, USA, January 26-28, 2011*, Thomas Ball and Mooly Sagiv (Eds.). ACM, 317–330. doi:10.1145/1926385.1926423

[23] Sankha Narayan Guria, Jeffrey S. Foster, and David Van Horn. 2023. Absynthe: Abstract Interpretation-Guided Synthesis. *Proc. ACM Program. Lang.* 7, PLDI (2023), 1584–1607. doi:10.1145/3591285

[24] Qinheping Hu, Jason Breck, John Cyphert, Loris D'Antoni, and Thomas W. Reps. 2019. Proving Unrealizability for Syntax-Guided Synthesis. In *CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I (LNCS, Vol. 11561)*, Isil Dillig and Serdar Tasiran (Eds.). Springer, 335–352. doi:10.1007/978-3-030-25540-4_18

[25] Qinheping Hu, John Cyphert, Loris D'Antoni, and Thomas W. Reps. 2020. Exact and approximate methods for proving unrealizability of syntax-guided synthesis problems. In *PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 1128–1142. doi:10.1145/3385412.3385979

[26] Henry S. Warren Jr. 2013. *Hacker's Delight, Second Edition.* Pearson Education. http://www.hackersdelight.org/

[27] Woosuk Lee. 2021. Combining the top-down propagation and bottom-up enumeration for inductive program synthesis. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–28. doi:10.1145/3434335

[28] Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. 2018. Accelerating search-based program synthesis using learned probabilistic models. *Proc. ACM Program. Lang.* PLDI (2018), 436–449. doi:10.1145/3192366.3192410

[29] V. I. Levenshtein. 1966. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady* 10 (Feb. 1966), 707.

[30] Qikang Liu, Yang He, Yanwen Cai, Byeongguk Kwak, and Yuepeng Wang. 2025. Synthesizing Document Database Queries Using Collection Abstractions . In *ICSE 2025*. IEEE Computer Society, Los Alamitos, CA, USA, 476–488. doi:10.1109/ICSE55347.2025.00152

[31] Roland Meyer and Jakob Tepe. 2025. Artifact for "Oriented Metrics for Bottom-Up Enumerative Synthesis". Zenodo. doi:10.5281/zenodo.17345358

[32] Roland Meyer and Jakob Tepe. 2025. Oriented Metrics for Bottom-Up Enumerative Synthesis. *CoRR* abs/2511.02491 (2025). doi:10.48550/arXiv.2511.02491 arXiv:2511.02491

[33] Hila Peleg and Nadia Polikarpova. 2020. Perfect Is the Enemy of Good: Best-Effort Program Synthesis. In *34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference) (LIPIcs, Vol. 166)*, Robert Hirschfeld and Tobias Pape (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2:1–2:30. doi:10.4230/LIPICS.ECOOP.2020.2

[34] Daniel Perelman, Sumit Gulwani, Dan Grossman, and Peter Provost. 2014. Test-driven synthesis. In *PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 408–418. doi:10.1145/2594291.2594297

[35] Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodík, and Dinakar Dhurjati. 2016. Scaling up Superoptimization. In *ASPLOS 2016, Atlanta, GA, USA, April 2-6, 2016*, Tom Conte and Yuanyuan Zhou (Eds.). ACM, 297–310. doi:10.1145/2872362.2872387

[36] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program synthesis from polymorphic refinement types. In *Proceedings of the 37th ACM SIGPLAN, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, Chandra Krintz and Emery D. Berger (Eds.). ACM, 522–538. doi:10.1145/2908080.2908093

[37] Oleksandr Polozov and Sumit Gulwani. 2015. FlashMeta: a framework for inductive program synthesis. In *OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, Jonathan Aldrich and Patrick Eugster (Eds.). ACM, 107–126. doi:10.1145/2814270.2814310

[38] Andrew Reynolds, Morgan Deters, Viktor Kuncak, Cesare Tinelli, and Clark W. Barrett. 2015. Counterexample-Guided Quantifier Instantiation for Synthesis in SMT. In *CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II (LNCS, Vol. 9207)*, Daniel Kroening and Corina S. Pasareanu (Eds.). Springer, 198–216. doi:10.1007/978-3-319-21668-3_12

[39] Klaus U. Schulz and Stoyan Mihov. 2002. Fast string correction with Levenshtein automata. *Int. J. Document Anal. Recognit.* 5, 1 (2002), 67–85. doi:10.1007/S10032-002-0082-8

[40] Kensen Shi, Jacob Steinhardt, and Percy Liang. 2019. FrAngel: component-based synthesis with control structures. *Proc. ACM Program. Lang.* 3, POPL (2019), 73:1–73:29. doi:10.1145/3290386

[41] Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodík. 2008. Sketching concurrent data structures. In *PLDI 2008, Tucson, AZ, USA, June 7-13, 2008*, Rajiv Gupta and Saman P. Amarasinghe (Eds.). ACM, 136–148. doi:10.1145/1375581.1375599

[42] Lynn Arthur Steen and J Arthur Seebach. 1978. *Counterexamples in Topology*. Vol. 18. Springer. doi:10.1007/978-1-4612-6290-9

[43] Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M. K. Martin, and Rajeev Alur. 2013. TRANSIT: specifying protocols with concolic snippets. In *PLDI '13, Seattle, WA, USA, June 16-19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 287–296. doi:10.1145/2491956.2462174

[44] Chenglong Wang, Alvin Cheung, and Rastislav Bodík. 2017. Synthesizing highly expressive SQL queries from input-output examples. In *PLDI 2017, Barcelona, Spain, June 18-23, 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 452–466. doi:10.1145/3062341.3062365

[45] Xinyu Wang, Greg Anderson, Isil Dillig, and Kenneth L. McMillan. 2018. Learning Abstractions for Program Synthesis. In *CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 10981)*, Hana Chockler and Georg Weissenbacher (Eds.). Springer, 407–426. doi:10.1007/978-3-319-96145-3_22

[46] Xinyu Wang, Isil Dillig, and Rishabh Singh. 2017. Synthesis of data completion scripts using finite tree automata. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 62:1–62:26. doi:10.1145/3133886

[47] Xinyu Wang, Isil Dillig, and Rishabh Singh. 2018. Program synthesis using abstraction refinement. *Proc. ACM Program. Lang.* 2, POPL (2018), 63:1–63:30. doi:10.1145/3158151

[48] Yongho Yoon, Woosuk Lee, and Kwangkeun Yi. 2023. Inductive Program Synthesis via Iterative Forward-Backward Abstract Interpretation. *Proc. ACM Program. Lang.* 7, PLDI (2023), 1657–1681. doi:10.1145/3591288

Received 2025-07-10; accepted 2025-11-06