Phased Synthesis of Divide and Conquer Programs

Azadeh Farzan
Department of Computer Science
University of Toronto
Toronto, Canada
azadeh@cs.toronto.edu

Victor Nicolet
Department of Computer Science
University of Toronto
Toronto, Canada
victorn@cs.toronto.edu

Abstract

We propose a fully automated method that takes as input an iterative or recursive reference implementation and produces divide-and-conquer implementations that are functionally equivalent to the input. Three interdependent components have to be synthesized: a function that divides the original problem instance, a function that solves each sub-instance, and a function that combines the results of sub-computations. We propose a methodology that splits the synthesis problem into three successive phases, each with a substantially reduced state space compared to the original monolithic task, and therefore substantially more tractable. Our methodology is implemented as an addition to the existing synthesis tool Parsynt, and we demonstrate the efficacy of it by synthesizing highly nontrivial divide-and-conquer implementations of a set of benchmarks fully automatically.

CCS Concepts. • **Theory of computation → Program reasoning**; **Divide and conquer**; Parallel computing models; • **Software and its engineering → Automatic programming**;

Keywords. Program Synthesis, Divide and Conquer

ACM Reference Format:
Azadeh Farzan and Victor Nicolet. 2021. Phased Synthesis of Divide and Conquer Programs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21), June 20–25, 2021, Virtual, Canada.* ACM, New York, NY, USA, 26 pages. https://doi.org/10.1145/3453483.3454089

1 Introduction

A divide-and-conquer computation decomposes a problem instance into several smaller sub-problems, solves each *independently*, and then combines the results to solve the original problem. It may produce better solutions for algorithmic problems by (i) improving the asymptotic complexity of the computation, for all program inputs or for generic subsets of them, or (ii) creating the potential for leveraging parallelism, since independent subtasks can be easily parallelized with good speedups. Writing a good divide-and-conquer algorithm is often non-trivial and can sometimes be quite tricky. Every undergraduate algorithms textbook has a chapter on divide-and-conquer and attempts to teach computer science students how to do the task by example. There are sometimes several instances of a divide-and-conquer solution for a given problem (an example follows in Section 2), and the specific usage determines the preferred solution from the pool of candidates. Automated synthesis can therefore offer a lot of utility in this problem space.

This paper proposes a systematic and automatable way of inferring a divide-and-conquer algorithm from an input reference implementation. The target divide-and-conquer algorithms adhere to the diagram in Figure 1. The input is an existing (iterative or recursive) implementation of a function $f : \mathcal{S} \mapsto D$, which is a single pass function over a collection (of general type $\mathcal{S}$). The output is a divide-and-conquer implementation of the same function. More specifically, a triple of functional components $(\triangledown, \mathbf{f}, \odot)$ is synthesized where $\triangledown : \mathcal{S} \to \mathcal{S}^n$ is an n-way *divide operator* ($n = 2$ in the figure), $\odot : D^n \to D$ is the complementary *join operator*, and $\mathbf{f}$ computes f and potentially some extra information which is strictly required for $\odot$ to recover $f(l)$ from the partial computations of subtasks. $\mathbf{f}$ is a *lifting* (in the standard category theory sense) of f.

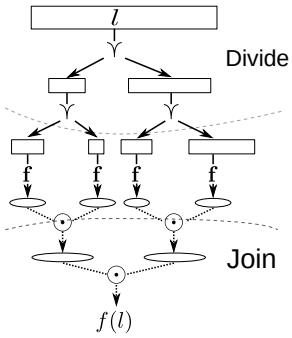


Figure 1. D&C Schema

The main restrictions of this family, compared to broadly understood divide-and-conquer algorithms, are: (i) the divide function $\triangledown$ has to be recursively applicable to an input collection, dividing it into smaller and smaller pieces, for an arbitrary number of calls, and (ii) the computation performed in each subproblem is a *lifting* of f. Yet, the model is very general and admits many interesting divide-and-conquer algorithms from the literature. In particular, it subsumes the dc1 category from [24] which proposes a manual methodology for producing such algorithms, and it is substantially

PLDI '21, June 20–25, 2021, Virtual, Canada

ACM ISBN 978-1-4503-8391-2/21/06…$15.00
https://doi.org/10.1145/3453483.3454089

more expressive than any class of divide-and-conquer considered for automated synthesis so far. MapReduce [7], for example, is a limited class of divide-and-conquer algorithms that has been targeted by automated synthesis successfully before [8, 9, 11, 22, 23]. The more general class we undertake is strictly more challenging to synthesize since three interdependent unknown code components $(\curlyvee, \mathbf{f}, \odot)$ need to be synthesized simultaneously. In [11, 22, 23], the target of synthesis is only the join operator $\odot$ (i.e. the *reduction* in MapReduce terminology). In [8, 9], the pair $(\mathbf{f}, \odot)$ was targeted through the simplifying assumption that $\curlyvee$ defaults to a simple sequence split operator, which is the inverse of sequence concatenation, i.e $\curlyvee(x \bullet y) = (x, y)$; this is the default assumption for all MapReduce frameworks.

We propose a *phased* synthesis procedure that breaks up the task of synthesizing $(\curlyvee, \mathbf{f}, \odot)$ into three separate synthesis subtasks. Figure 2 illustrates the workflow of our methodology. The first observation is that instead of synthesizing $\curlyvee$, one can synthesize a *divide predicate*, namely a specification for the acceptable outcome of $\curlyvee$. A predicate p is a valid divide predicate if and only if the results of the computations performed on the parts that satisfy p can be combined correctly. In other words, *there exists* a function $\odot$ such that, if a an input z is split into two parts x and y such that $p(x, y)$ holds, then $f(z) = f(x) \odot f(y)$. Our approach exploits the fact that $\odot$ has to only exist and need not be determined while p is being synthesized.

If p is successfully synthesized, then $\curlyvee$ is synthesized such that its output adheres to what p specifies. If the synthesis of p fails, then the reason for this failure may be that extra information, which is currently missing from what f computes, is required for the existence of $\odot$. Therefore, an attempt is made to *lift* f, and it is replaced by a new function $\mathbf{f}$, which additionally computes the missing information. Then, the attempt to synthesize p is repeated for the new function $\mathbf{f}$.

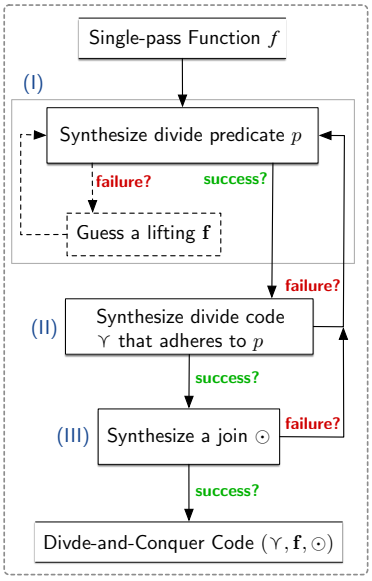


Figure 2. Phased Synthesis Schema

The synthesis of $\curlyvee$ may also fail. In general, it is not always possible or feasible to synthesize a piece of code (i.e. $\curlyvee$) that adheres to a given specification (i.e. p). In this case, a new p is requested with the hopes that a change in the predicate will lead to a successful step (II). Once $\curlyvee$ is synthesized, the algorithm proceeds to synthesize $\odot$ as the only remaining unknown, which is guaranteed to exist at this point. If this is successful, the procedure concludes and $(\curlyvee, \mathbf{f}, \odot)$ is returned.

The synthesis loop is further constrained to only explore implementations that are at least as efficient as the input reference implementation, so that no useless divide-and-conquer solutions are generated. One can iterate through the loop to find a first valid solution, and continue to enumerate several valid solutions.

Once the synthesis problem is decomposed as depicted in Figure 2, the synthesis of $\curlyvee$ and $\odot$ (boxes (II) and (III)) can be performed in a relatively standard way through syntax-guided synthesis, since each phase performs the synthesis of a single unknown code component (Section 7). We propose a novel algorithm for the synthesis of the *divide predicate* p (box (I)), which also predicts if a lifting of f will be required and produces the lifting. This algorithm, in the spirit of *deductive synthesis* [17], simultaneously infers p and any required lifting $\mathbf{f}$ that would guarantee the existence of a join implementation $\odot$, without the need to implement the dashed loop (for guessing $\mathbf{f}$) depicted in Figure 2.

In Figure 2, the input is a single-pass function f over a collection. In our technique, we accept an iterative or recursive implementation as input and produce an equivalent single-pass function f automatically. This step is described in [10]. In summary, in this paper:

- We lay out the theoretical foundations to reduce the problem of divide-and-conquer synthesis from the specification of Figure 1 to one more amenable to automation (Section 4).
- We propose a phased synthesis algorithm that synthesizes the triple of unknowns $(\curlyvee, \mathbf{f}, \odot)$ in three different stages employing both syntax-guided synthesis and deductive synthesis techniques (Section 5).
- We propose a novel algorithm based on deductive synthesis that can efficiently discover two unknowns, a divide predicate and a lifting of f (Section 6). Our proposed automated lifting algorithm surpasses previously known algorithms [8, 9] in that it can infer conditional accumulators to extend the signature of the function which were not possible before.
- We illustrate through a set of benchmarks that an implementation of our proposed approach can synthesize highly nontrivial divide-and-conquer solutions based on simple input implementations.

2 Motivating Example

We use an example to illustrate the types of programs that our approach can synthesize automatically. Additionally, the example underlines the following two observations: (i) there are often several acceptable divide-and-conquer implementations of a given function, and (ii) synthesizing a divide-and-conquer solution is not solely about discovering

the division and join operators, but may also require a lifting of the original input code.

Consider a set of (input) points on a 2D plane. A point is *Pareto optimal* if all the other points are either below or to the left of this point. Let $p.x$ and $p.y$ denote the coordinates of point p. Formally:

$$p \gg p' \iff p.x \geq p'.x \vee p.y \geq p'.y$$
$$\text{POP}(X) = \{p \in X \mid \forall p' \in X, p \gg p'\}.$$

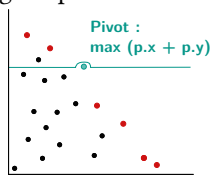
The code in Figure 3 computes POP(X) where the input X is a list of points. At each iteration, the set of optimal points is updated by maintaining the points that remain optimal with respect to the new input, and adding the new input if it is optimal with respect to all currently optimal points. If the divide operator is taken as the trivial split of the input list, then the correct join matching this divide has a quadratic time complexity. By the Master Theorem [5], the complexity of the resulting *naive* divide-and-conquer algorithm is $O(n^2)$, which matches that of the original input implementation.

```
List<Point> l = [];
List<Point> tmp = [];
for(i = 0; i < n; i++) {
  Point p = A[i];
  bool b = true;
  tmp = [];
  for(e in l) {
    if(e ≫ p)tmp.append(e);
    b = b && (p ≫ e); }
  if (b) tmp.append(p);
  l=tmp;
}
```

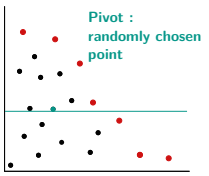
Figure 3. Single pass POP

Other *divide functions* yield algorithms with lower asymptotic complexities. We briefly introduce three such algorithms with a two-way divide, a two-way divide with *lifting* and a three-way divide, all of which our tool Parsynt can automatically synthesize.

The solution with a two-way divide is illustrated on the right: a (pivot) point p is chosen by taking the point with the maximum sum of coordinates, which is guaranteed to be Pareto optimal. The point set is then partitioned into two sets: the set of points (vertically) above and strictly below p. The optimal points of the original point set is then the concatenation of the lists of optimal points from each partition.



If the pivot is chosen at random, and therefore not guaranteed to be Pareto optimal, then the Pareto optimal points of the top partition all remain optimal. But of the ones in the bottom partition, those which are to the left of the rightmost point of the top partition have to be removed. This cannot be done without a *lifting*. Some additional information, for example the rightmost point of each partition, is needed so that the join can correctly combine the two Pareto optimal sets by pruning the result from the bottom partition.



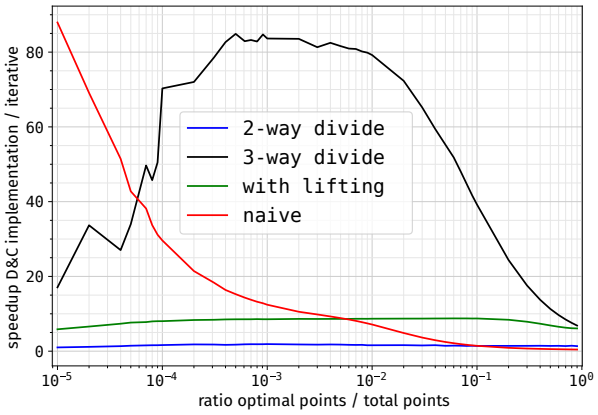
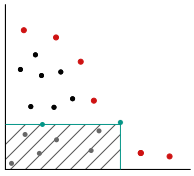


Figure 4. Speedups of sequential divide-and-conquer implementations of POP relative to implementation in Figure 3

Finally, in the implementation with a three-way divide, a point is chosen at random and the rightmost point above this point is chosen as the pivot. The point set is partitioned into three subspaces (as illustrated on the right): the points above, to the right, and below and to the left of the pivot. The third partition (hatched) does not contain any Pareto optimal point. The Pareto optimal points for the other two partitions are optimal for the original point set and therefore the join operator can simply concatenate the results from these two partitions.



All algorithms, except the naive one, partition the space in linear time, and join the results in constant or linear time, which puts them in the same asymptotic complexity class $O(n \log n)$. However, the performance of these solutions varies significantly depending on the composition of the input data; specifically, the ratio of the Pareto optimal points to the total number of points. The graph in Figure 4 illustrates the speedups of the different divide-and-conquer implementations relative to the input implementation of Figure 3. The horizontal axis is the ratio of optimal points in the input list (of size 2×10^5). When the ratio of optimal points is very small, the naive implementation performs significantly better than all other implementations, with speedups reaching 80x. When there are very few Pareto optimal points, the (quadratic) join operator defaults to a constant time complexity. As the ratio of optimal points increases, the performance of the naive implementation decreases to drop below all the other algorithms. Our tool produces all algorithms automatically and the user selects the best for their specific usage.

3 Background and Notation

Let *Sc* be a type that stands for any scalar type used in typical programming languages, such as `int` and `bool`, whenever the specific type is not important in the context. Scalars are

assumed to be of *constant* size, and conversely, any constant-size representable data type is assumed to be scalar. Consequently, all operations on scalars are assumed to have constant time complexity. Type $\mathcal{S}$ defines the set of all *sequences* of elements of type Sc. The concatenation operator $\bullet : \mathcal{S} \times \mathcal{S} \to \mathcal{S}$ defined over sequences is associative. The sequence type stands in for *arrays*, *lists*, or any collection data type that admits a linear iterator and an *associative* composition operator. A function $h : \mathcal{S} \to D$ is rightwards iff there exists a binary operator $\oplus : D \times Sc \to D$ such that for all $x \in \mathcal{S}$ and $a \in Sc$, we have $h(x \bullet [a]) = h(x) \oplus a$. A rightwards function can be defined by a left fold over a sequence: $h(x) = \textbf{foldl}\ \oplus\ h([])\ x$. A leftward function is defined analogously using the recursive equation $h([a] \bullet x) = a \otimes h(x)$. A function is *single-pass* if it is leftwards or rightwards.

4 Decomposing D&C Specification

The input to our approach is an implementation of a single-pass function $f : \mathcal{S} \to D$. To accommodate generic reference implementations, which may be a (nested) loop or a recursive function performing multiple passes over the input data, we propose a (source-to-source) translation in [10]. This translation converts an arbitrary iterative or recursive input implementation to a single pass recursive function f.

The goal of this section is to start with a generic specification for divide-and-conquer and transform it to a *tractable* specification for search-based synthesis. A general divide-and-conquer algorithm comprises of a *divide* function $\triangledown : \mathcal{S} \to \mathcal{S}^c$ and a *join* function $\odot : D^c \to D$ (with $c > 1$) that satisfy the specfication:

$$\Psi(\odot, \triangledown) \equiv \forall z \in \mathcal{S} : \\ f(z) = \odot\Big(f(\triangledown(z).1), f(\triangledown(z).2), \ldots, f(\triangledown(z).c)\Big) \quad (1)$$

Since $\odot$ and $\triangledown$ must be computable, the solution space for them is the set of recursive functions. To accommodate full automation, we focus on a slightly more limited universe of divide functions, namely those that *partition* the input space.

To simplify the formal notation, we restrict c to be precisely 2. All formal statements stated and proved in this paper generalize to any value for c, and therefore this does not cause a loss in generality. The stronger specification for *partition* divide-and-conquer algorithms is:

$$\Psi^\circ(\odot, \triangledown) \equiv \forall z \in \mathcal{S} :\ f(z) = f(\triangledown(z).1) \odot f(\triangledown(z).2) \\ \wedge\ \widetilde{z} = \widetilde{\triangledown(z).1} \cup \widetilde{\triangledown(z).2} \quad (2)$$

where $\widetilde{z}$ denotes the set of elements of sequence z.

Note that both Ψ and Ψ° admit trivial (useless) solutions. For example, a valid solution for $\triangledown$ is to divide a sequence s into the sequence s and the empty sequence [], and let $\odot$ return its first component. To rule these out, we add a constraint on the sizes of individual outputs generated by $\triangledown$ over a universe of inputs. It requires the existence of at least one input which would divide a list of length $m + k$ into two sublists of arbitrary sizes m and k. Formally:

$$\chi(\triangledown) \equiv \forall m, k \in \mathbb{N}, \exists z \in \mathcal{S} : |\triangledown(z).1| = m \wedge |\triangledown(z).2| = k$$

Note that this can be extended to multi-way divides in a straightforward way. Combining Ψ° from Equation 2 with χ will result in our first concrete specification with non-trivial solutions:

$$\Psi^\bullet(\triangledown, \odot) \equiv \Psi^\circ(\triangledown, \odot) \wedge \chi(\triangledown) \quad (3)$$

$\Psi^\bullet$ (a strict strengthening of Ψ) is the precise specification we aim to use for divide-and-conquer synthesis. Yet this specification defines a huge (intractable) search space for existing search-based program synthesis techniques. We propose a way to decompose this specification such that $\triangledown$ and $\odot$ can be synthesized *independently*, even though they are related through $\Psi^\bullet$.

Key insight. The most straightforward division operation is the inverse of sequence concatenation, that is, the sequence z is divided into any pair of sequences z_1 and z_2 such that $z = z_1 \bullet z_2$. The key observation is that a general divide $\triangledown$ satisfying $\Psi^\bullet$ can be defined as a composition of a sequence permutation function and this trivial divide. That is, if $\Psi^\bullet(\triangledown, \odot)$ then there exists a permutation function $\pi : \mathcal{S} \to \mathcal{S}$ such that $\forall z \in \mathcal{S} : z = \pi(\triangledown(z).1 \bullet \triangledown(z).2)$. This a simple consequence of the constraint $\widetilde{z} = \widetilde{\triangledown(z).1} \cup \widetilde{\triangledown(z).2}$.

The insight leads to the specification below, which makes use of a predicate $p : \mathcal{S} \times \mathcal{S} \to$ Bool and a permutation function π instead of the divide function $\triangledown$:

$$\Phi(\pi, p, \odot) \equiv \chi^\bullet(p) \wedge \forall (x, y) \in \mathcal{S}^2 : \\ p(x, y) \Rightarrow f(\pi(x \bullet y)) = f(x) \odot f(y) \quad (4)$$

where χ is reformulated for p as

$$\chi^\bullet(p) = \forall m, k \in \mathbb{N}, \exists x, y \in \mathcal{S}^2 : |x| = m \wedge |y| = k \wedge p(x, y)$$

The new specification is only a reframing of our problem, and as the following theorem states, $\Psi^\bullet$ can be used instead of Φ without a compromise.

Theorem 4.1. *The specifications $\Psi^\bullet$ (defined in Equation 3) and Φ (defined in Equation 4) are mutually realizable.*

So far, we have reformulated the problem of synthesizing a divide and a join operation to a different yet *equirealizable* problem of synthesizing a predicate p and a join operation. This is an intermediate step that facilitates the independent algorithmic synthesis of p and $\odot$, in contrast to the monolithic task that is put forward by the specification Φ. First, we discuss how this can be achieved through a specialization of the synthesis problem.

4.1 Permutation Invariance

If f is not sensitive to the order of elements in its input sequence, then π can be eliminated from Φ.

Definition 4.2 (Permutation invariant)**.** A function f is *permutation invariant* iff for all permutation functions π and all lists $x \in \mathcal{S}$, $f(x) = f(\pi(x))$.

If f is permutation invariant and $(\pi, p, \odot)$ is a solution for Φ, then $(\pi', p, \odot)$ for any permutation function π' is also a valid solution to Φ. Therefore, we can simply replace π with the identity permutation and simplify Φ to:

$$\Phi^\bullet(p, \odot) \equiv \forall x, y \in \mathcal{S}^2 : \\ p(x, y) \Rightarrow f(x \bullet y) = f(x) \odot f(y) \wedge \chi^\bullet(p) \quad (5)$$

Theorem 4.3. *If the function f is permutation invariant then the specifications $\Phi^\bullet$ and $\Psi^\bullet$ are mutually realizable.*

An insight from the proof (in Appendix B.2) is the necessary relation between p and $\curlyvee$, which must satisfy $\forall z \in \mathcal{S}, p(\curlyvee(z).1, \curlyvee(z).2)$.

The practicality of $\Phi^\bullet$ (over $\Psi^\bullet$) is that without π, one can synthesize p and $\odot$ in two independent (synthesis) steps as discussed in Sections 5 and 6.

4.2 Splitting Divides

The results in Section 4.1 are theoretically crisp, but seem restricted in the sense that they do not apply if the function is not permutation invariant. It turns out that solutions to specification $\Phi^\bullet$ go beyond divide functions for permutation invariant functions. Consider splitting divides as formally defined below.

Definition 4.4 (Splitting divide)**.** A divide function $\curlyvee : \mathcal{S} \to \mathcal{S} \times \mathcal{S}$ is a splitting divide if $\forall z \in \mathcal{S} : z = \curlyvee(z).1 \bullet \curlyvee(z).2$.

A splitting divide $\curlyvee$ can also be synthesized (indirectly) through the specification $\Phi^\bullet$. The reason goes as follows. The restriction of $\Psi^\bullet$ to splitting divides is:

$$\Psi^\star(\curlyvee, \odot) \equiv \chi(\curlyvee) \wedge (z) = f(\curlyvee(z).1) \odot f(\curlyvee(z).2) \\ \wedge \ \forall z \in \mathcal{S} : \curlyvee(z).1 \bullet \curlyvee(z).2 = z \quad (6)$$

Proposition 4.5. *If $\Psi^\star$ is realizable then so is $\Phi^\bullet$.*

Proposition 4.5 concludes that whenever a splitting divide solution exists, $\Phi^\bullet$ also presents a corresponding solution. The converse does not hold; there may exist a solution for $\Phi^\bullet$ while no solution for $\Psi^\star$ exists. Note that splitting divides provide valid divide-and-conquer implementations for many instances where f is not permutation invariant.

So far we have argued how $\Phi^\bullet$ can be used to generate two different generic class of divide-and-conquer algorithms. Theoretically, there is no guarantee that either f is a permutation invariant or a splitting divide for f exists. Practically, however, we have not come across a single case for this. Nevertheless, to close the theoretical gap, in Appendix B.4, we discuss a general construction that translates an arbitrary function f to a permutation-invariant implementation of it $\check{f}$. We prove that realizability of $\Phi^\bullet$ for f implies realizability of $\Phi^\bullet$ for $\check{f}$. Therefore, if f is not permutation invariant and a splitting divide does not exist, then one can theoretically synthesize a divide-and-conquer solution for $\check{f}$ instead.

5 Synthesizing Divide-and-Conquer

The intention with the design of a divide-and-conquer implementation is to either obtain a better performing sequential algorithm or a parallelizable algorithm. In both cases, the resulting algorithm should not have a worse computational complexity than the input reference implementation. Note that the specification(s) from Section 4 carry no guarantees about the computational complexity of the synthesized code. We first introduce sufficient conditions that guarantee a reasonable computational complexity for the synthesized program. Then, we provide a refinement of the schema in Figure 2 that incorporates these complexity constraints and makes explicit use of specifications introduced in Section 4.

5.1 Complexity of Divide-And-Conquer

The time complexity of synthesized divide-and-conquer algorithms can be measured through the Master Theorem [5]. Under the assumption that the divide operator is *balanced*, the time complexity is defined through the recurrence $T(n) = kT(n/c) + w(n)$, where $w(n)$ captures the combined complexities of $\odot$ and $\curlyvee$, and c and k respectively determines the number of subproblems created and recursively solved.

In our synthesis context, we assume $w(n)$ to have simple polynomial complexity, that is $O(n^m)$ for some $m \geq 0$, since it is difficult to relate logarithmic complexities to syntax. We use a triple (m, k, c) to denote the *complexity budget* for a divide-and-conquer algorithm, from which (through the Master Theorem) its asymptotic complexity can be calculated. In a typical divide-and-conquer algorithm, there is usually a tension between the complexity of divide and join functions. If the algorithm does more work upfront, to perform a favourable division, then the task of combining will become simpler. Conversely, if it performs a cheap division, a more elaborate join will be required. *Quick sort* and *merge sort* can be respectively considered instances of the two scenarios. We use this insight to enumerate different possible solutions to synthesis. The $O(n^m)$ total cost for divide and join operations is computed as the combination of the complexities $O(n^{m_\curlyvee})$ for division and $O(n^{m_\odot})$ for join.

5.2 Synthesis Paradigm

Figure 5 illustrates a precise instantiation of the schema presented earlier in Figure 2, incorporating a complexity budget and the specifications introduced in Section 4. Our synthesis algorithm maintains an internal budget for the join operation, which is initialized to the smallest possible value $\mathcal{B}_\odot = (m_\odot, k, c) = (0, 1, 2)$. The algorithm attempts to synthesize a solution within budget $\mathcal{B}_\odot$, and if it fails then it

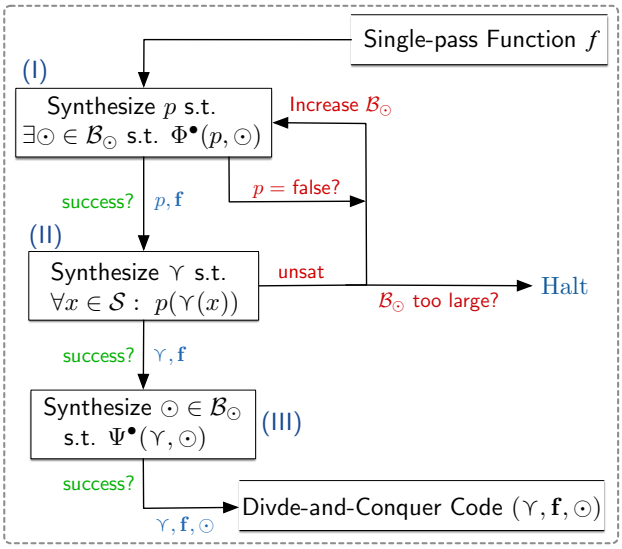


Figure 5. Phased Synthesis Schema

increases the budget until it reaches a limit, where the divide-and-conquer code will end up with higher computational complexity than the input implementation. At the high level, the algorithm proceeds in three phases:

(I) The *weakest divide predicate* p is synthesized that satisfies the specification $\exists \odot : \Phi^\bullet(p, \odot)$. At this stage, the algorithm does not synthesize an implementation for $\odot$ but rather guarantees its existence within budget $\mathcal{B}_\odot$. An algorithm for (I) is the key (algorithmic) contribution of this paper and appears in Section 6.

(II) A divide operation $\curlyvee$ *matching* p is synthesized. $\curlyvee$ is the lowest (time) complexity divide operation that satisfies $\forall x \in \mathcal{S} : p(\curlyvee(x))$. As a direct consequence of the conceptual contributions presented in Section 4, this phase can be efficiently implemented using a straightforward syntax-guided synthesis routine.

(III) An implementation of $\odot$ (within budget $\mathcal{B}_\odot$) is synthesized such that $\Psi^\bullet(\curlyvee, \odot)$ holds. Similar to (II), this step can be implemented using a straightforward syntax-guided synthesis routine.

The loop between steps (I) and (II) may succeed several times, for increasing values of $\mathcal{B}_\odot$, and therefore, it can enumerate many valid solutions when more than one exist.

If step (I) fails to discover a predicate p, the default value *false* is returned which triggers the step to be repeated with a higher budget $\mathcal{B}_\odot$. The fact that p is *the weakest* predicate implicitly guarantees that it satisfies $\chi^\bullet$, and therefore, it does not have to explicitly appear as part of $\Phi^\bullet(p, \odot)$.

The algorithm also produces a *lifting* of f to guarantee the existence of $\odot$ in step (I), if one is required. In other words, the f as it appears in $\Phi^\bullet(p, \odot)$ may be the original f or a lifting $\mathbf{f}$ (of f) that facilitates the existence of the $\odot$. Our proposed algorithm for synthesis of p in Section 6 also accommodates the computation of $\mathbf{f}$, if necessary.

In step (II), the algorithm attempts to synthesize $\curlyvee$ such that total complexity of divide-and-conquer algorithm based on the budget $(max(m_\odot, m_\curlyvee), k, c)$ is at most as computationally expensive as f. A failure in this step means that a divide function matching the predicate p cannot be synthesized. If step (II) fails, then $\mathcal{B}_\odot$ is increased so that a different predicate is produced in step (I).

$\mathcal{B}_\odot = (m_\odot, k, c)$ is increased first by incrementing k until $k = c$, and then by incrementing m until the complexity of f is reached, and finally by incrementing c. Note that there is no theoretical bound on c, but it is often a small constant and therefore a small bound is preset for it in this loop. The loop terminates when $\mathcal{B}_\odot$ reaches its limit.

By the end of step (II), the algorithm has synthesized a *divide* operation $\curlyvee$ that satisfies the specification $\exists \odot : \Psi^\bullet(\curlyvee, \odot)$ such that the combined cost of $\curlyvee$, known since it has been synthesized, and $\odot$, known through $\mathcal{B}_\odot$, does not surpass the asymptotic complexity of f. Therefore, step (III) is guaranteed to succeed.

The solution space of possible divide-and-conquer algorithms in each iteration subsumes that of the previous iteration since $\mathcal{B}_\odot$ is increased. Therefore, a predicate solution from an earlier iteration is a valid solution for a later iteration. However, a new predicate, admitted through a bigger $\mathcal{B}_\odot$, is strictly weaker than a predicate from an earlier iteration . This is precisely why to ensure progress, we actively seek the weakest predicate that satisfies the constraints in step (I). This also guarantees that upon termination, the algorithm explores all possible divide-and-conquer solutions with a join function of polynomial time complexity within acceptable range.

6 Deductive Recursion Synthesis

This section presents an algorithm for step (I) in Figure 5. The goal is to find a divide predicate p such that there is a join function $\odot$ within a given budget $\mathcal{B}_\odot = (m_\odot, k, c)$ that satisfies $\Phi^\bullet(p, \odot)$, that is (for $c = 2$):

$$\forall x, y \in \mathcal{S}^2 : p(x, y) \Rightarrow f(x \bullet y) = f(x) \odot f(y). \quad (7)$$

Key insight. In the above specification, f is the known recursive function, p is an unknown *recursive* predicate and $\odot$ is an unknown recursive function. The idea is to use the recursive definition of f to infer the recursive definition of p (and $\odot$). We do this by induction on the two f input parameters x and y, and the two $\odot$ input parameters $f(x)$ and $f(y)$ (unless they are scalars). Starting with empty lists, one can solve for p and $\odot$ for lists of increasing sizes, and then extrapolate a recursive definition for p (and for $\odot$). Since f is rightwards single-pass, it is sufficient to perform induction only on y and on $f(x)$, but not on x. We start with an intuitive explanation of the algorithm through an example, and then present the formal details.

Recall example POP from Section 2. We illustrate how a recursive definition of p may be discovered for the 2-way

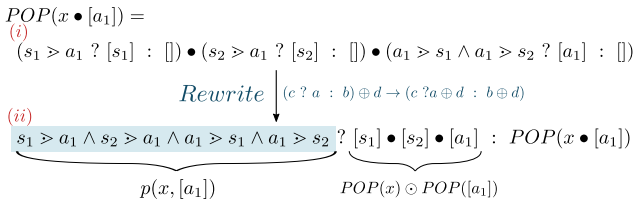


Figure 6. Expression of the unfolding POP($x \bullet [a_1]$)

divide in one of the divide-and-conquer instances discussed for POP. To simplify the presentation of this instance, we assume that we know $\odot = \bullet$ a priori, even though this is not generally the case. We start by doing induction on y and POP(x). We let $y = [a_1]$ and $y = [a_1, a_2]$ for two different instances, and POP(x) = $[s_1, s_2]$. Note that the list elements are symbolically denoted by variables a_1, a_2, s_1, and s_2.

Expression (i) in Figure 6 is the result of inlining the recursive definition of POP($x \bullet [a_1]$). Since x is fixed, one can view POP($x \bullet [a_1]$) as the *unfolding* of function POP starting from POP(x). Observe that expression (i) in no way resembles the format of Equation 7, and the expression of p cannot be guessed from it. Using a few simple rewrite rules, specifically $(c\ ?\ a\ :\ b) \oplus d \rightarrow c\ ?\ a \oplus d\ :\ b \oplus d$ and the factoring of conditionals, expression (i) can be rewritten to expression (ii) [1]. Note that in expression (ii), the then-expression is equal to POP(x) • POP($[a_1]$). Therefore, if the highlighted expression is true, then POP($x \bullet [a_1]$) = POP(x)• POP($[a_1]$), which makes the highlighted expression our best conjecture for $p(x, [a_1])$. Yet, one *unfolding* is not sufficient for deducing a recursive definition for p. Therefore, we repeat this process with y as a list of two elements, which will produce a conjecture for $p(x \bullet [a_1, a_2])$. If we compute the expression for POP($x \bullet [a_1, a_2]$) and rewrite it, we get:

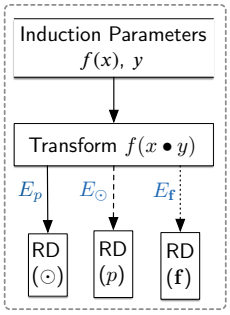
$$\begin{aligned}\text{POP}(x \bullet [a_1, a_2]) = &\Big(\neg(a_1 \succcurlyeq a_2) \vee p(x, [a_1])\Big) \wedge \Big(\neg(a_2 \succcurlyeq a_1) \vee \\ &(s_1 \succcurlyeq a_2 \wedge s_2 \succcurlyeq a_2 \wedge a_2 \succcurlyeq s_1 \wedge a_2 \succcurlyeq s_2)\Big)\ ? \\ &\text{POP}(x) \bullet \text{POP}([a_1, a_2]) : \text{POP}(x \bullet [a_1, a_2])\end{aligned}$$

The expression for $p(x, [a_1, a_2])$ is again identifiable from the conditional operator at the root. Observe that both expressions are instances of the more general pattern $\forall r \in \text{POP}(y), \forall s \in \text{POP}(x), s \succcurlyeq r \wedge r \succcurlyeq s$, which is precisely the recursive definition that our algorithm extrapolates from these two instances through a *recursion discovery* (RD) step. We say well-formed expressions like these are in *normal form*. $\Phi^\bullet$ can be transformed to:

$$\forall (x, y) \in \mathcal{S}^2 : f(x \bullet y) = p(x, y)\ ?\ f(x) \odot f(y) : f(x \bullet y) \quad (8)$$

where p appears as a conditional subexpression of the right-hand side, rather than as a precondition in Equation 7. This facilitates the identification of normal forms through the transformation of the expression of $f(x \bullet y)$.

We made the simplifying assumption that $\odot = \bullet$, but in general, it is unknown. Therefore, instead of knowing that POP(x) • POP($[a_1, a_2]$) is the join expression, we have to characterize the shape of valid join expressions. Then, based on Equation 8, a guess is made for a subexpression representing p based on the expression under the condition having the right shape.

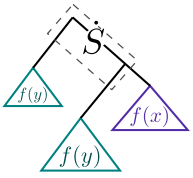


The diagram on the right summarizes the procedure. First, based on induction parameters of increasing size, the expressions of the unfoldings of $f(x \bullet y)$ are transformed to normal forms (see Section 6.1). The relevant sets of subexpressions for p, $\odot$, and a possible lifting $\mathbf{f}$ are extracted from the normal forms for successive unfoldings (see Section 6.2). Synthesizing p is the main goal of this procedure. In some instances, when the transformation works well, the set $E_\odot$ is precise enough so that $\odot$ can be discovered through recursion discovery. But, this is not always the case, and the only guarantee of this step is its existence within budget $\mathcal{B}_\odot$. Lifting is done when required, and the transformation produces the candidate set $E_\mathbf{f}$ (see Section 6.3). Finally, a recursion discovery (RD) subroutine extrapolates recursive definitions for p, $\odot$, and $\mathbf{f}$ respectively out of the sets of expressions E_p, $E_\odot$, and $E_\mathbf{f}$. (see Section 6.4).

6.1 Normal Forms

We first present a characterization of the expression (a normal form) of $\odot$ informed by a budget $\mathcal{B}_\odot$.

$\mathcal{B}_\odot$-normal form. The $\mathcal{B}_\odot$-normal form intuitively describes the shape of the expression of $f(x) \odot f(y)$. To characterize the unfolded expression of a join within budget $\mathcal{B}_\odot$ we define $\mathcal{B}_\odot$-normal forms parameterized by the budget, and the input expressions of the join $f(x)$ and $f(y)$. For a budget $\mathcal{B}_\odot = (m_\odot, k, c)$, the normal form illustrated on the right characterizes the expression of the join in the form of the expression skeleton $\dot{S}$. The leaves completing the skeleton are the inputs to $\odot$ which should be filled with $f(x)$ or $f(y)$. If $\dot{S}$ is meant to characterize a join within budget $\mathcal{B}_\odot$, then it can admit at most k inputs parameters. For example, since both $f(x)$ and $f(y)$ appear in the expression on the right, it is only in normal form for $k = 2$.



The join should be computable in $O(n^{m_\odot})$ time. Recall that normal forms are defined for expressions of fixed size resulting from unfoldings on inputs of fixed size, as the example in the beginning of this section suggests. We define a notion of *cost* for these expressions such that when a general recursive $\odot$ is synthesized using the normal form, we will have the guarantee that $\odot \in O(n^{m_\odot})$. An expression is in normal form if it adheres to a particular shape and has a particular *cost*.

An *expression skeleton* of degree k, denoted $\dot{S}^k$, is an abstract syntax tree (AST) described by the grammar on the

[1] Appendix C.1 spells out the rewriting steps for the interested reader.

right, where the leaves are constants or indexed holes $??_i$ with $1 \leq i \leq k$. Given a set of input expressions $E = \{e_i\}_{1 \leq i \leq k}$, $\dot{S}^k[E]$ denotes the expression constructed by replacing the hole $??_i$ in $\dot{S}^k$ by expression e_i, for all $1 \leq i \leq k$. In our algorithm, $\dot{S}^k$ characterizes the *shape* of a join of arity k, and the e_i's stand for the inputs to the join function, for instance $f(x)$ and $f(y)$. Note that skeletons have a fixed degree k, since k is the number of parameters of $\odot$ fixed by the budget.

$\dot{S}^k = \dot{S}^k \ominus \dot{S}^k \mid \dot{S}^k \lessgtr \dot{S}^k$
$\mid \neg \dot{S}^k \mid \dot{S}^k \bullet \dot{S}^k$
$\mid \dot{S}^k.m$
$\mid \dot{S}^k[j]$ where $j \in \mathbb{N}$
$\mid \dot{S}^k \; ? \; \dot{S}^k : \dot{S}^k$
$\mid ??_i$ where $0 < i \leq k$
$\mid$ true $\mid$ false $\mid n \in int$
$\ominus$: arithmetic or boolean operator.
$\lessgtr$: comparison operator.
$.m$: field acessor.

The cost associated to each skeleton is a vector $\vec{m}$ of length c (the number of partitions produced by $\curlyvee$). Semantically, it represents the conjecture that $\dot{S}^k[E]$ is computable in $O(n^{max(\vec{m})})$[2] time for arbitrary inputs E of size n. Note that G is a bounded expression, and the e_i that the algorithm considers are bounded, at each step of the inductive reasoning. Yet, the final join function has to have the right time complexity over arbitrary-sized inputs.

Intuitively, one can establish the complexity of the join operator by examining the candidate skeletons over different induction steps. The skeleton from the previous induction step and its associated cost forms a *context* that is used as a parameter to determine the cost of the new skeleton for the current induction step.

The *context* consists of a skeleton $\dot{S}^k_{prev}$, a cost conjecture $\vec{m}$, and an identifier $0 < i_c \leq c$ for the induction parameter expanded in the current induction step. Initially, $\vec{m} = \vec{0}$, and a skeleton of minimal size is assumed in the first induction step. The cost $\vec{m}$ of a skeleton $\dot{S}^k$ is determined based on the subexpression relation between $\dot{S}^k$ and $\dot{S}^k_{prev}$. If $\dot{S}^k$ is not changed in the current induction step, one can infer that its computation takes constant time with respect to the current induction parameter i_c. Otherwise, $\dot{S}^k_{prev}$ must appear as a subexpression of $\dot{S}^k$, since the target of synthesis is a recursive function. If there is a new hole $??_i$ in $\dot{S}^k$, which is not part of $\dot{S}^k_{prev}$, then the induction component $\mu(i)$ that corresponds to the hole is updated (μ maps the input of the skeleton to induction components). Otherwise, the current induction component is updated. Intuitively, since an extra subexpression appears in $\dot{S}^k$, there is an additional computation step required for one induction step, and the computation takes linear time. The algorithm is illustrated on the right, but it is missing a few cases, which seem to be uncommon in practice and did not occur in the synthesis of any of our benchmarks. The complete algorithm, listing all cases, appears in Appendix C.2.

if $\dot{S}^k = \dot{S}^k_{prev}$ **then**
  $\vec{m}[i_c] = 0$;
else if *No new hole in* $\dot{S}^k$ **then**
  $\vec{m}[i_c] = 1$;
else if *New hole* $??_i$ *in* $\dot{S}^k$ **then**
  $\vec{m}[\mu(i)] = 1$;
...;

[2] $max(\vec{v})$ is the maximum of the components of vector $\vec{v}$.

Definition 6.1 ($\mathcal{B}_\odot$-normal form)**.** An expression e is in $\mathcal{B}_\odot$-normal form in context C, for a budget $\mathcal{B}_\odot = (m_\odot, k, c)$, with respect to a family of expressions $E = \{e_i\}_{1<i\leq k}$, iff there exists a skeleton $\dot{S}^k$ such that $e = \dot{S}^k[E]$ and $max(\vec{m}) = m_\odot$, where $\vec{m}$ is the cost of $\dot{S}^k$ in context C.

We say that an expression is $\mathcal{B}_\odot$**-normalizable** in context C with respect to E if it can be rewritten to an expression in $\mathcal{B}_\odot$-normal form in context C with respect to E. The context is only mentioned explicitly if it is relevant.

Multi-way conditional expression. If $p(x, y) \equiv true$, that is if any division is acceptable, then $f(x \bullet y)$ is $\mathcal{B}_\odot$-normalizable with respect to $\{f(x), f(y)\}$. But, if a special division is necessary, then the shape of the expression $p(x, y)?f(x) \odot f(y) : f(x \bullet y)$ (from Equation 8) hints at the fact that only a subtree of the AST, after rewriting, is in $\mathcal{B}_\odot$-normal form. This is the subexpression that appears under the *then* branch of the conditional expression.

Definition 6.2. An expression $e = \{e_i \textbf{ if } b_i \mid i \in I\}$ is a multi-way conditional expression (MC-expression) with branch conditions $\{b_i\}_{i\in I}$, if branch expressions $\{e_i\}_{i\in I}$ do not contain any

***Example* 6.3.** Let $\uparrow$ denote the infix operator returning the maximum of two values. We use a computation of the longest increasing subsequence (LIS) of a list of integers as our running example in this section. The single-pass function $\mathsf{LIS} : [int] \to int \times int \times int$ with signature $(cl, ml, prev)$, where ml is the length of the longest increasing subsequence and cl is the length of the longest increasing suffix, is defined as (for any sequence x, state s, and integer a):

$$\mathsf{LIS}([]) = (0, 0, -\infty) \qquad \mathsf{LIS}(x \bullet [a]) = \mathsf{LIS}(x) \oplus a$$
$$s \oplus a = \textbf{let } cl = s.prev < a \; ? \; s.cl + 1 : 0 \textbf{ in } (cl, cl \uparrow s.ml, a)$$

We consider the second unfolding starting from $\mathsf{LIS}(x) = (cl_0, ml_0, prev_0)$, with input $[a_1, a_2]$. The expression of $\mathsf{LIS}(x \bullet [a_1, a_2]).ml$ is translated to a MC-expression with 4 branches:

$$\begin{array}{lll}
1: & ml_0 \uparrow cl_0 + 1 \uparrow cl_0 + 1 + 1 & \textbf{if } (a_1 < a_2) \wedge (prev_0 < a_1) \\
2: & ml_0 \uparrow cl_0 + 1 \uparrow 0 & \textbf{if } (a_1 \geq a_2) \wedge (prev_0 < a_1) \\
3: & ml_0 \uparrow 0 \uparrow 0 + 1 & \textbf{if } (a_1 < a_2) \wedge (prev_0 \geq a_1) \\
4: & ml_0 \uparrow 0 \uparrow 0 & \textbf{if } (a_1 \geq a_2) \wedge (prev_0 \geq a_1)
\end{array}$$

The expression of the divide predicate p intuitively corresponds to the subset of branches where the expressions under guards match the expressions of the function $\odot$. Formally, in a MC-expression $e = \{e_i \textbf{ if } b_i \mid i \in I\}$, a boolean expression b **isolates** the subset of branches $I' \subseteq I$ iff $\forall i \in I' : b_i \implies b$ and $\forall i \in I \setminus I' : b_i \implies \neg b$.

6.2 Expression Transformation

Given an unfolding of $f(x \bullet y)$, we first transform it into an MC-expression. Then we check which branches have expressions that are $\mathcal{B}_\odot$-normalizable by attempting to normalize them. The expression of p is exactly the expression isolating the subset of branches that are successfully rewritable to $\mathcal{B}_\odot$-normal forms, and the expression of $\odot$ is the skeleton defining the normal forms.

***Example* 6.4.** Recall Example 6.3 and suppose that we have a constant time budget for $\odot$. In the second induction step, the two inputs of the join are $\{(cl_0, ml_0, prev_0), \mathsf{LIS}([a1, a2])\}$. Branch 3 and 4 expressions can be rewritten to a $\mathcal{B}_\odot$-normal form, witnessed by the skeleton $\hat{S}^2 = ??_1.ml \uparrow ??_2.ml$, which can be computed in constant time with cost $(0, 0)$ (accounting for the context from previous step).

However, there is no normal form of cost $(0, 0)$ for the branches 1 and 2: the branches contain subexpressions of the form $cl_0 + 1 + \ldots$ that grow in size with each induction step, so the inferred cost in these branches is $(0, 1)$.

Branches (3,4) are normalizable for a constant-time budget, and the expression of the predicate is identified by isolating those branches: $p(\mathsf{LIS}(x), [a_1, a_2]) = a_1 \leq prev_0$. ⌟

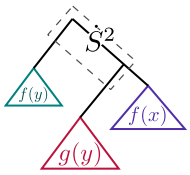
Any expression in the program can be transformed to an MC-expression. The first step consists in using a strongly normalizing rewrite system, with rules similar to the ones used in the introductory example of this section (complete list in Appendix A.3). The expression generated by the rewrite system is an MC-expression. Then a solver eliminates the branches that are infeasible; that is, each branch with index i such that $\neg b_i$ is valid is removed.

The $\mathcal{B}_\odot$-normal form generalizes the constant normal forms and recursive normal forms defined in [9]. Normalizing a symbolic expression to a $\mathcal{B}_\odot$-normal form can be done by small adjustments in the rewriting process from [9], which in turn is a relatively standard cost-based rewrite system. We list the rewrite rules used in Appendix A.2. Note that the context for $\mathcal{B}_\odot$-normalization depends on the branch of the MC-expression. Each branch has a matching branch at the previous induction step, from which the context is taken.

With an ideal normalization process, the predicate expressions synthesized are guaranteed to correspond to the expressions of the weakest predicate that ensures a join operator $\odot$ exists within budget $\mathcal{B}_\odot$, since at each unfolding stage, all the branches that are $\mathcal{B}_\odot$-normalizable are isolated. But, since reachability of an existing $\mathcal{B}_\odot$-normal form is undecidable [8, 9], the weakness of p cannot be theoretically guaranteed for all inputs. This also implies that the join cannot be always synthesized by our algorithm.

6.3 Automatic Lifting

During the process of identifying $\mathcal{B}_\odot$-normalizable subexpressions, instead of discovering a clean $\mathcal{B}_\odot$-normal form, the expression sometimes normalizes to a tree of the form illustrated on the right. There are leaves corresponding to $f(x)$ and $f(y)$ as before, but there is also a leaf that corresponds to subcomputations not already performed by f. The figure labels this as a new function $g(y)$. The normal form implies that the join operator needs access to the result of $g(y)$ to produce the overall result. Hence, f needs to lifted to compute g in addition to its original computation.



Normalization of a single branch of an MC-expression can have three possible outcomes: (i) success (i.e. no lifting required), (ii) lifting required, or (iii) failure, when the cost of the expression surpasses the budget. One can aim for a solution based on all branches in class (i) (with no lifting), or for one based on all branches in classes (i) and (ii) to produce the weakest predicate supported by the lifting.

***Example* 6.5.** Recall Example 6.4; a predicate p is discovered without a need for lifting (i.e. branches 3 and 4 belong in class (i)). Suppose now that we aim to identify *all* branches as normalizable; this will lead to $p \equiv true$, then $\vee$ being the random splitting, an instance of a MapReduce solution. For this a lifting is required, since subexpressions of the form $cl_0 + (1 + \ldots)$, which appear branches 1 and 2, have to be precomputed to maintain the possibility of a constant-time join. The exact expression depends on the condition $a_1 < a_2$, therefore we derive the auxiliary computation $g_1([a_1, a_2]) = a_1 < a_2 \ ? \ 1 + 1 \ : \ 1$. Additionally, the condition isolating branches (3,4), $a_1 \leq prev_0$ has to be available for join, the extra auxiliary $g_2([a_1, a_2]) = a_1$ is also required. Therefore, branches 1 and 2 belong in class (ii). ⌟

A similar deductive-style automated lifting was introduced for lists in [8] and extended to multidimensional lists in [9]. But with the aid of MC-expressions, we can infer strictly more expressive auxiliary computation in this paper, in particular we can synthesize *conditional auxiliary* computations. More details about the procedure and an example are presented in Appendix C.3.

6.4 Recursion Discovery

The goal of recursion discovery is to deduce the recursive definition of a function from its unfoldings. In [8, 9], a procedure is proposed for solving this exact problem. It operates by using subtree isomorphisms to identify different stages of a recursive computation in an input set of expressions. We apply their procedure as a black-box in two instances: the divide predicate and the lifting discovery.

At each step of the induction process, the unfoldings of the rightwards single-pass function f from an initial state $f(x)$ on sequence y are transformed to expressions of the form $p(f(x), y) \ ? \ f(x) \odot \mathbf{f}(y) \ : \ f(x \bullet y)$. With an ideal normalization process, this would allow to identify the unfoldings of $\odot$,

p and g, a function that computes the information required in addition to f in the lifting $\mathbf{f}$. As noted in Section 6.1 no such ideal procedure exists, and in practice we can only identify the expressions of $p(f(x), y)$ and $g(y)$, but not $\odot$, for the different values of $f(x)$ and y during the induction process.

Recursion discovery is used to produce a recursive definition of $p_0 : D \times \mathcal{S} \to \text{Bool}$ from the unfoldings of p_0, i.e. the expressions of the form $p_0(f(x), y)$, for different inputs $f(x)$ and y. The function p_0 is defined recursively by an operator $\otimes$ such that $p_0(f(x), y \bullet [a]) = p_0(f(x), y) \otimes (f(x), y, a)$. Note that $p_0(f(x), y)$ may be false, which means no corresponding divide can be discovered. Once p_0 is discovered, the divide predicate p is simply defined as $p(x, y) = p_0(f(x), y)$.

Recursion discovery is also used to discover a recursive definition of a lifting of f when necessary. It produces a function g from the expressions $g(y)$ (for different values of y) identified after normalization, which is then tupled with f to form a lifting $\mathbf{f}$.

***Example* 6.6.** In Example 6.5, we identified bounded expressions that correspond to the required lifting. From a set of such expressions, recursion discovery deduces a recursive definition for g_1, with signature $(cond, aux)$:

$$g_1(x \bullet [a]) = \textbf{let } c \ = \ g_1(x).cond \wedge (f(x).prev < a) \textbf{ in}$$
$$\textbf{let } b = g_1(x).aux \textbf{ in } (c, c \ ? \ b + 1 : b)$$

The final lifting of LIS is $\text{LIS}'(x) = (\text{LIS}(x), g_1(x), head(x))$ since *head* is the trivial result of recursion discovery from the unfoldings of g_2 in Example 6.5. ⌟

7 Synthesizing Divide and Join

Once the divide predicate p is successfully synthesized, the two remaining tasks are the synthesis of the join function and the synthesis the divide function using p as its specification. These are simple tractable synthesis problems for search-based synthesis tools, which is precisely the reason why we decomposed the problem in this particular manner. We briefly explain each task (at the high level) for the sake of the completeness of the paper.

7.1 Divide Function Synthesis

We use syntax-guided synthesis [3] to synthesize a divide function according to specification $\forall z \in \mathcal{S}, p(\curlyvee(z)) \wedge \chi(\curlyvee)$ (from Section 4). For a SyGuS solution, the search space for synthesis has to be defined.

If $p(x, y) \equiv \text{true}$, then the inverse of concatenation is a valid solution. Incidentally, it is the only valid constant-time divide function. If $p(x, y) \not\equiv \text{true}$, we assume that $\curlyvee$ has at least linear time complexity. By analyzing the predicate p, we can distinguish whether only a splitting divide (Definition 4.4) is required, or a partition divide needs to be synthesized. If the predicate $p(x, y)$ is a condition on a prefix of its second argument y, then a splitting divide is synthesized. The divide is constructed as a function that scans the input sequence from a random location, until the condition on the prefix starting from the location is met, at which point the sequence is split into the current prefix and the suffix.

Otherwise, the divide is *sketched* [3] as a partition function that operates in two phases, first by selecting one or more *pivots*, and then partitioning the elements of the inputs list according to their relation to these pivots. For a given sketch, a number q of pivots is fixed. For a budget (m, k, c), the unknowns are the q pivot selection functions and $c - 1$ two-way partition functions using the pivots. If no solution for a given q is found, q can be increased. 2 pivots seemed to be sufficient to cover all our benchmarks. The time complexity is at least linear, but can be higher if the selection of pivot requires super-linear time. While none of our benchmarks required a super-linear pivot selection function, we successfully experimented with synthesizing one with our tool to test its robustness. The example is a pivot constrained to be the median of a list. Detailed descriptions of these sketches are given in Appendix C.4.

7.2 Join Operator Synthesis

With $\curlyvee$ known, the specification is simplified to $\forall z \in \mathcal{S}, f(z) = f(\curlyvee(z).1) \odot f(\curlyvee(z).2)$. In general case, this synthesis problem is *identical* to a similar problem that was solved in [9], with good theoretical guarantees. We do not repeat that contribution here. Whenever the procedure described in Section 6 succeeds in producing the join, the synthesis step in (III) is effectively reduced to a bounded verification of the already discovered $\odot$, effectively checking that the divide-and-conquer algorithm with the divide synthesized in step (II) and the join operator inferred at step (I) is functionally equivalent to the original function. For example, concatenation, as the join for the two-way divide solution of POP, is inferred at the divide predicate synthesis step.

8 Experimental Results

Our approach is implemented as an extension of the tool Parsynt [20], which accepts as input C-like iterative programs with loops or functional programs written in Scheme. It is implemented in OCaml [16] and uses Z3 [6] as SMT solver and Rosette [25] as syntax-guided synthesis solver. All experiments were run on a desktop with two 8-core Intel Xeon E5-2620 and 32GB of RAM running Ubuntu 18.04.

To the best of our knowledge Parsynt is the only fully automatic tool that can synthesize divide-and-conquer programs of the class described in this paper from a reference implementation. A number of tools, including BIGλ [23], and Casper [1], synthesize various types of MapReduce [7] programs. The MapReduce model is too restrictive for splitting or partitioning divides, and all the tools mentioned fail to synthesize a solution for POP example from Section 2 or LIS example from Section 6. GraSSP [11] goes slightly beyond MapReduce and parallelizes single pass array computations, but the most expressive class they target is subsumed by

our solutions with splitting divides. An earlier version of Parsynt [9] targets nested loops and performs lifting, but the divide operations are limited to the inverse of concatenation. We use the most difficult benchmarks from GraSSP as some of our simplest benchmarks.

Benchmarks. The theoretical results of Section 4 suggest a classification of the algorithms targeted in this paper into those with *splitting divides* (e.g. LIS) and those with *partitioning divides* (e.g. POP). We evaluate the efficacy and efficiency of Parsynt in synthesizing divide-and-conquer algorithms for two sets of benchmarks, one for each category. We collected our benchmarks from algorithm textbooks and related work on divide-and-conquer programming. These are non-trivial iterative algorithms for which equivalent divide-and-conquer algorithms according to Equation 1 exist. Those that have a solution with a partitioning divide are listed in Table 1 and those with a splitting divide in Table 7(a). The first set includes sophisticated algorithms, where some have several distinct divide-and-conquer implementations, synthesized using different budgets. The second set is composed of single-pass algorithms computing counts or maximal lengths of subsequences that have a given property.

Performance of Parsynt. Table 1 and Figure 7(a) report the synthesis times for each phase of the synthesis procedure. For each synthesis task in Table 1, we report the synthesis times separately for each budget that led to a synthesized solution. When the predicate synthesized is trivially true, there is no need to synthesize a divide; these cases are denoted by †. The synthesis times range from a few seconds to up to 26 min. The solutions with three-way divides ($c = 3$) require significantly more time to be synthesized. This is due to the fact that the size of the bounded model required by the synthesis needs to be increased to take in account the increase in dimension. All input implementations have $O(n^2)$ time complexity[3], and therefore the synthesized solutions with $O(n \log n)$ complexity are highly non-trivial, on par with the three solutions of POP discussed in Section 2.

	$\mathcal{B}_\odot$	p	$\vee$	$\odot$	$O(?)$
Sorting	(0,2,2)	4.5	1.1	5.1	$n \log n$
k-largest	(0,2,2)	3.4	1.2	120	$n \log n$
Closest pair	(0,2,2)	5.6	1.2	10	$n \log n$
Intersecting intervals	(0,2,2)	12	54	30	$n \log n$
	(0,3,3)	31	421	1.5	$n \log n$
Histogram	(0,2,2)	4.1	1.1	25.3	$n \log n$
	(2,2,2)	3.0	†	9.4	n^2
POP	(0,2,2)	5.3	69	8.7	$n \log n$
	(1,2,2)	6.2	20	240	$n \log n$
	(2,2,2)	3.1	†	12	n^2
	(0,2,3)	35	1560	91	$n \log n$
Minimal points	(0,2,2)	5.0	64	10.5	$n \log n$
	(1,2,2)	6.4	21.5	206	$n \log n$
	(2,2,2)	3.0	†	11.5	n^2
	(0,2,3)	35	1430	87.0	$n \log n$
Quadrant orthogonal convex hull	(0,2,2)	5.2	67	13	$n \log n$
	(1,2,2)	6.7	24.5	201	$n \log n$
	(2,2,2)	3.0	†	12	n^2
	(0,2,3)	35	1540	88	$n \log n$
Orthogonal convex hull	(1,2,2)	×	×	×	$n \log n$
	(2,2,2)	6.1	†	24	n^2
Encircling set	(0,2,2)	×	×	×	$n \log n$

Table 1. Partitioning Divides: Columns p, $\vee$, $\odot$ present the synthesis time (in seconds) for the respective functions, and column $O(?)$ lists the time complexity of the synthesized code. † indicates that no divide needs to be synthesized, a greyed cell signals that lifting was required, and × means that Parsynt fails.

In Table 7(a), we report the times spent in the predicate and the join synthesis steps. Note that the predicate synthesis times listed are the combined times for both lifting and the predicate synthesis step (which take place in one step).

The benchmarks for which GraSSP [11] can also synthesize a solution are highlighted in blue. Note that Parsynt synthesizes two solutions for each benchmark against one for the other tool. The synthesis times are overall small for these benchmarks, but Parsynt still illustrates a time advantage (see Appendix D.1). The rest of our benchmarks are rejected by other tools (including GraSSP), because they require *lifting* in the form of addition of one or more conditional accumulators with a non-trivial accumulation operation.

Quality of the synthesized code. The synthesized implementations for the benchmarks in Table 1 belong to one of the two categories: (1) the synthesized divide-and-conquer algorithm has a strictly *lower asymptotic complexity* than the input sequential code (any row with $O(n \log n)$ complexity) or (2) its asymptotic complexity is the same (about 22% of the cases). In Section 2, we discussed how different input distributions may result in a preference for one solution over another, for the latter case.

The solutions with a splitting divide lead to scalable parallel implementations, as showcased in Figures 7(b) and 7(c). In Figure 7(b) we compare the speedup of the different parallel implementations of the benchmarks of Table 7(a) with varying number of threads, for an input of 10^{10} integers with indivisible blocks of 100 elements in average. For each benchmark, we have two solutions: one with a splitting divide (plotted with a continuous line) and one with a lifting (dashed line). Both implementations scale in parallel with comparable performance gains. The relative speedups for these can also depend on the input data composition. Figure 7(c) compares the relative speedups of the two implementations of LIS, for varying sizes of increasing sequences in the input. When increasing sequences are long, the splitting divide implementation performs significantly worse than the one with lifting. This observation generalizes across all benchmarks that have splitting divide and lifting solutions, and makes a case for why synthesizing two solutions is useful.

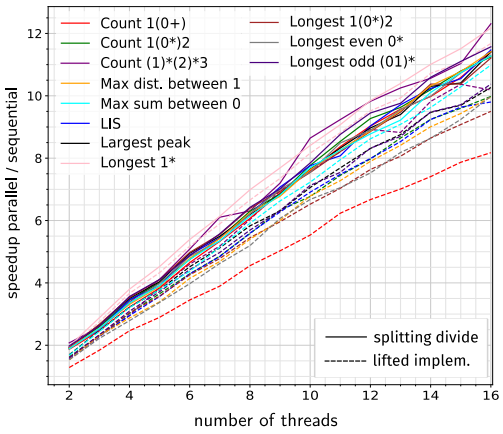
Limitations. Each table also lists the benchmarks that underline the limitations of the various steps of our synthesis process. We indicate by × the synthesis steps for which Parsynt fails. For example, in Table 1 the solution for $\mathcal{B}_\odot = (1, 2, 2)$ of the *orthogonal convex hull* benchmark, which requires a complex lifting, could not be automatically synthesized. The tool cannot synthesize the divide for the *encircling set* benchmark because it involves the synthesis of a function with non-linear arithmetic operations. The benchmark

[3]For the k-largest benchmark we consider the case where k is large.

(a)

	$p + \curlyvee$	$\odot$
Count 1(0+)	0.1	1.2
Count 1(0*)2	0.1	1.4
Count 1*2*3	0.2	3.1
Max dist. between 1s	0.2	2.6
Max sum between 0s	0.2	1.8
LIS	0.1	1.3
Largest peak	0.1	1.2
Longest 1*	0.1	1.2
Longest 1(0*)2	0.2	3.1
Longest even 0*	0.2	19.1
Longest odd (01)*	0.2	25.0
Longest 1(0*)2(0*)3	×	×

(b)



(c)

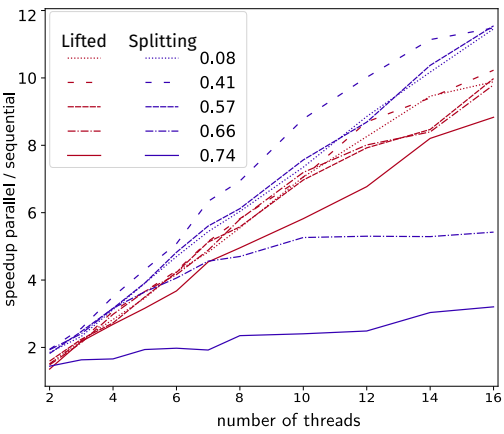


Figure 7. Splitting Divides: Table (a) lists the synthesis times (in seconds) of the benchmarks with splitting divides. × indicates that the tool failed to find a solution. Figure (b) illustrates the speedups of the parallel implementations. Figure (c) compares the speedups of two parallel implementations of the LIS benchmark, for different ratios of size of increasing sequences to total size of input.

Longest 1(0)2(0*)3* in the last row of Table 7(a) (where the code computes the length of the longest substring matching the regular expression $1(0^*)2(0^*)3$) admits a splitting divide, but the deductive synthesis procedure cannot infer a divide predicate due to the large number of new variables required to compute the predicate. The reader can refer to Appendix E.1 where we outline how difficult it is to derive these divide-and-conquer algorithms, even manually.

9 Related Work

There is a vast body of work on program synthesis. Here we only survey the work related to divide-and-conquer synthesis. Map-reduce is one of the most popular subclasses of divide-and-conquer, which formally relies on the computation being a list homomorphism, the precise class of functions that can be written as a composition of a *map* and a *reduction* (cf. first homomorphism theorem). The literature on divide-and-conquer synthesis can be divided into two categories based on the class of input computations targeted: (1) those with list homomorphisms as input, with the aim of synthesizing efficient map-reduce [7] programs [1, 14, 21, 23], (2) those that go beyond list homomorphisms [8, 9, 11, 15, 19, 22], and target code with more dependencies. In category (2), the techniques in [8, 9, 11] *synthesize list homomorphisms* through some variation of *lifting*, the approach in [22] uses symbolic execution at runtime and to identify and defer dependencies, and Bellmania [15] targets input programs in the style of dynamic programming and orchestrates an efficient execution schedule to accommodate the dependencies. A direct comparison with work in [1, 9, 11, 23] with respect to the class of input programs appears in Section 8.

Derivation of list homomorphisms includes approaches based on the third homomorphism theorem [13, 14, 19], function composition [12], and quantifier elimination [18], as well as those based on recurrence equations [4]. These techniques are either not fully automatic, or rely on additional guidance from the programmer beyond the input sequential code. In contrast, the techniques in [8, 9, 11, 22] derive list homomorphisms automatically through *lifting*. The lifting performed in [9] is strictly the most general one and subsumes the rest.

The class of divide-and-conquer algorithms targeted in this paper is strictly more general than list homomorphisms, and therefore more general than both categories (1) and (2) of work mentioned earlier. To the best of our knowledge, no prior work targets a class as general as this automatically. In [24], *manual synthesis* of general classes is discussed.

10 Conclusion

We solve a program synthesis problem with three unknown components, related through a single specification, by decomposing it into tractable subtasks. The key takeaways are: (1) our deductive synthesis technique based on *induction*, *rewriting*, and *recursion discovery* is a powerful method for the synthesis of recursive code where another recursive code is available as the functional specification, and (2) an imperfect deductive synthesis algorithm can be utilized as an oracle producing powerful hints, which can be used to decompose a monolithic synthesis problem with multiple unknowns into a sequence of more tractable synthesis problems over subsets of these unknowns.

Our deductive synthesis module differs from the classic one in that instead of operating on the source code, it manipulates the results of its symbolic evaluation. Small variations in code may result in the same symbolically evaluated term,

and therefore, the technique is more robust with respect to syntactic variations in the input implementations.

Our approach in decomposing the monolithic divide-and-conquer specification is in the spirit of multi-abduction [2]: a specification with multiple unknowns is decomposed into specifications for each unknown. The problem is that in this domain, like many others, individual maximal specifications for each component do not exist; a stronger specification on divide would imply less work to be done at join time. We exploit the structure of the problem to effectively enumerate all admissible pairs of specifications, by relying on the complexity of the join function to guide this enumeration.

References

[1] Maaz Bin Safeer Ahmad and Alvin Cheung. 2018. Automatically Leveraging MapReduce Frameworks for Data-Intensive Applications. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 1205–1220.

[2] Aws Albarghouthi, Isil Dillig, and Arie Gurfinkel. 2016. Maximal Specification Synthesis. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. Association for Computing Machinery, New York, NY, USA, 789–801.

[3] 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*. 1–25.

[4] Yosi Ben-Asher and Gadi Haber. 2001. Parallel Solutions of Simple Indexed Recurrence Equations. *IEEE Trans. Parallel Distrib. Syst.* 12, 1 (Jan. 2001), 22–37.

[5] Jon Louis Bentley, Dorothea Haken, and James B Saxe. 1978. *A General Method for Solving Divide-and-Conquer Recurrences*. Technical Report.

[6] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '08)*. Springer, 337–340.

[7] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (Jan. 2008), 107–113.

[8] Azadeh Farzan and Victor Nicolet. 2017. Synthesis of Divide and Conquer Parallelism for Loops. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '17)*. ACM, 540–555.

[9] Azadeh Farzan and Victor Nicolet. 2019. Modular Divide-and-conquer Parallelization of Nested Loops. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19)*. ACM, 610–624.

[10] Azadeh Farzan and Victor Nicolet. 2021. From Iterative Implementations to Single-pass Functions. (2021). http://www.cs.toronto.edu/~azadeh/resources/papers/functional_translation.pdf (manuscript - not peer reviewed).

[11] Grigory Fedyukovich, Maaz Bin Safeer Ahmad, and Rastislav Bodik. 2017. Gradual Synthesis for Static Parallelization of Single-Pass Array-Processing Programs. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '17)*. Association for Computing Machinery, New York, NY, USA, 572–585.

[12] Allan L. Fisher and Anwar M. Ghuloum. 1994. Parallelizing Complex Scans and Reductions. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation (PLDI '94)*. 135–146.

[13] Alfons Geser and Sergei Gorlatch. 1997. Parallelizing Functional Programs by Generalization. In *Proceedings of the 6th International Joint Conference on Algebraic and Logic Programming (ALP '97-HOA '97)*. 46–60.

[14] Sergei Gorlatch. 1996. Systematic Extraction and Implementation of Divide-and-Conquer Parallelism. In *Proceedings of the 8th International Symposium on Programming Languages: Implementations, Logics, and Programs (PLILP '96)*. 274–288.

[15] Shachar Itzhaky, Rohit Singh, Armando Solar-Lezama, Kuat Yessenov, Yongquan Lu, Charles Leiserson, and Rezaul Chowdhury. 2016. Deriving divide-and-conquer dynamic programming algorithms using solver-aided transformations. In *ACM SIGPLAN Notices*, Vol. 51. ACM, 145–164.

[16] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. 2018. The OCaml system release 4.07: Documentation and user's manual. (2018).

[17] Zohar Manna and Richard Waldinger. 1979. Synthesis: dreams → programs. *IEEE Transactions on Software Engineering* 4 (1979), 294–328.

[18] Akimasa Morihata and Kiminori Matsuzaki. 2010. Automatic Parallelization of Recursive Functions Using Quantifier Elimination. In *Functional and Logic Programming, 10th International Symposium, FLOPS 2010, Sendai, Japan, April 19-21, 2010. Proceedings*. 321–336.

[19] Kazutaka Morita, Akimasa Morihata, Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi. 2007. Automatic Inversion Generates Divide-and-conquer Parallel Programs. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. 146–155.

[20] Victor Nicolet. 2017. Parsynt. https://github.com/victornicolet/parsynt

[21] Cosmin Radoi, Stephen J. Fink, Rodric Rabbah, and Manu Sridharan. 2014. Translating Imperative Code to MapReduce. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14)*. 909–927.

[22] Veselin Raychev, Madanlal Musuvathi, and Todd Mytkowicz. 2015. Parallelizing User-defined Aggregations Using Symbolic Execution. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15)*. 153–167.

[23] Calvin Smith and Aws Albarghouthi. 2016. MapReduce Program Synthesis. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*, Vol. 51. 326–340.

[24] Douglas R Smith. 1985. Top-down synthesis of divide-and-conquer algorithms. *Artificial Intelligence* 27, 1 (1985), 43–96.

[25] Emina Torlak and Rastislav Bodík. 2013. Growing solver-aided languages with rosette. In *ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2013, part of SPLASH '13, Indianapolis, IN, USA, October 26-31, 2013*. 135–152.

A Additional Background

A.1 Model of an Input Imperative Program

This section introduces the syntax of the sequential imperative programs. Figure 8 presents the syntax of the input sequential programs. We assume an imperative language with basic constructs for branching and looping. Scalar variables or of `int` or `bool` type, and we can build sequences from these types. We can also have sequence variables, to which elements can be appended either via append or by concatenation of a singleton list.

$$
\begin{array}{lr}
v \in \mathsf{LhVar} ::= v'[e] & v' \in \mathsf{LhVar}, e \in \mathsf{Exp} \\
\quad | \; x & x \in \mathsf{Var} \\
\\
e \in \mathsf{Exp} ::= e \circ e' & e, e' \in \mathsf{Exp} \\
\quad | \; e \circledcirc e' & e, e' \in \mathsf{Exp} \\
\quad | \; be \wedge be' | \neg be & be, be' \in \mathsf{Exp} \\
\quad | \; le \bullet le' & le, le' \in \mathsf{Exp} \\
\quad | \; v & v \in \mathsf{LhVar} \\
\quad | \; \mathsf{if}\ be\ \mathsf{then}\ e\ \mathsf{else}\ e' & \\
\quad | \; k & k \in \mathbb{Z}, \mathbb{Q}, \mathbb{R} \\
\quad | \; \mathsf{true} \mid \mathsf{false} & \\
\mathsf{Program} ::= c; c' & c, c' \in \mathsf{Program} \\
\quad | \; v := e & v \in \mathsf{LhVar}, e \in \mathsf{Exp} \\
\quad | \; \mathtt{if}\ (e)\ \{c\}\ \mathtt{else}\ \{c'\} & be \in \mathsf{Exp}, c, c' \in \\
\mathsf{Program} & \\
\quad | \; \mathsf{for}\ (i \in \mathcal{I})\ \{c\} & i \in \mathsf{Iterator}
\end{array}
$$

Figure 8. Program Syntax The binary $\circ$ operator represents any arithmetic operation $(+, -, *, /)$, $\circledcirc$ operator represents any comparator $(<, \leq, >, \geq, =, \neq)$. $\bullet$ is the list concatenation operator. $\mathcal{I}$ is an iteration domain, and $\wedge$ operator represents any boolean operation $(\wedge, \vee)$.

A.2 Rewrite Rules for $\mathcal{B}_\odot$-Normalization

Figure 9 represents the rewrite rules using in the normalization process. Similarly to [8, 9] the rules are applied when they reduce the cost of an expression. In the context of $\mathcal{B}_\odot$-normalization, the cost is minimal when the expression is in $\mathcal{B}_\odot$-normal form. A difference from the previous work [8, 9] using a similar approach is that there are cases where induction on the leftmost input of the join is required. In previous work, the cost function was defined as to measure the depth and number of occurrences of symbols corresponding to the initial state of the unfoldings. In this work, we add an additional step to the cost inference, which consists in reducing the value of the cost when subexpressions can be summarized as the initial state (as opposed to single symbols being the initial state).

A.3 Multi-way Conditional Expressions

Any expression from the language can be translated to a multi-way conditional form in two steps. Figure 10 presents the rewrite rules that allow to rewrite an expression of the language with conditionals to an expression where all the

$a \odot b$	$\rightarrow$	$b \odot a$	
$(a \odot b) \odot c$	$\rightarrow$	$a \odot (b \odot c)$	
$(a \odot b) \otimes c$	$\rightarrow$	$(a \otimes c) \odot (b \otimes c)$	
$(a \otimes c) \odot (b \otimes c)$	$\rightarrow$	$(a \odot b) \otimes c$	*factor-right*
$(c\ ?\ x : y) \odot z$	$\rightarrow$	$c\ ?\ (x \odot z) : (y \odot z)$	
$z \odot (c\ ?\ x : y)$	$\rightarrow$	$c\ ?\ (z \odot x) : (z \odot y)$	
$c\ ?\ (x \odot z) : (y \odot z)$	$\rightarrow$	$(c\ ?\ x : y) \odot z$	
$c\ ?\ (z \odot x) : (z \odot y)$	$\rightarrow$	$z \odot (c\ ?\ x : y)$	
$c1\ ?\ (c2\ ?\ x : y) : z$	$\rightarrow$	$c1 \wedge c2\ ?\ x : (\neg c2\ ?\ y : z)$	
$max(a,\ b) > c$	$\rightarrow$	$a > c \vee b > c$	
$c > max(a,\ b)$	$\rightarrow$	$c > a \wedge c > b$	
$max(a,\ b) < c$	$\rightarrow$	$a < c \wedge b < c$	
$c < max(a,\ b)$	$\rightarrow$	$c < a \vee c < b$	
$min(a,\ b) > c$	$\rightarrow$	$a > c \wedge b > c$	*min-distr->*
$c > min(a,\ b)$	$\rightarrow$	$c > a \vee c > b$	*min-distr->-left*
$min(a,\ b) < c$	$\rightarrow$	$a < c \vee b < c$	*min-distr-<*
$c < min(a,\ b)$	$\rightarrow$	$c < a \wedge c < b$	*min-distr-<-left*
$\neg(a \wedge b)$	$\rightarrow$	$(\neg a) \vee (\neg b)$	
$\neg(a \vee b)$	$\rightarrow$	$(\neg a) \wedge (\neg b)$	*push-neg-down'*
$-(a + b)$	$\rightarrow$	$(-a) - b$	
$-(a - b)$	$\rightarrow$	$b - a$	*push-min-down'*
$a > c \vee b > c$	$\rightarrow$	$max(a,\ b) > c$	
$c > a \wedge c > b$	$\rightarrow$	$c > max(a,\ b)$	
$c < max(a,\ b)$	$\rightarrow$	$c < a \vee c < b$	
$a < c \wedge b < c$	$\rightarrow$	$max(a,\ b) < c$	

Figure 9. Algebraic rewrite rules. a, b and c are expressions. $\odot$ stands for any associative and commutative operator.

conditionals are at the top of the expression tree, and all the leaves are unconditional expressions. From the expression resulting of applying the rewrite rules, the multi-way conditional can be obtained by searching for the unconditional expressions while memorizing the conditional path in the expression tree. For each of the unconditional expressions, one branch of the multi-way is produced. The second step uses a solver to eliminate all the branches that are unfeasible. By construction, the disjunction of all the branch conditions is valid since all the conditionals in the function are balanced.

B Proofs

B.1 Proof of Lemma B.1

Lemma B.1. $\Psi^\bullet(\curlyvee, \odot) \implies \Phi(\pi, p, \odot)$ *if for all sequences* z *we have* $p(\curlyvee(z).1, \curlyvee(z).2)$ *and* $z = \pi(\curlyvee(z).1 \bullet \curlyvee(z).2)$.

The proof is trivial, the lemma simply makes the relation between the divide function and the permutation and predicate function explicit.

Assume we have a valid solution $(\curlyvee, \odot)$ for $\Psi^\bullet$, a predicate p and a permutation function π such that $\forall z \in \mathcal{S} : p(\curlyvee(z)) \wedge z = \pi(\curlyvee(z).1 \bullet \curlyvee(z).2)$. Additionally, we know that since $(\curlyvee, \odot)$ satisfy $\Psi^\bullet$ then $\curlyvee$ satisfies χ. So for any $m, k \in \mathbb{N}$,

$$
\begin{array}{ccc}
c\ ?\ e_1 : e_2 \oplus c'\ ?\ e_1' : e_2' & \rightarrow & c\ ?\ (c'\ ?\ e_1 \oplus e_1' : e_1 \oplus e_2') : (c'\ ?\ e_2 \oplus e_1' : e_2 \oplus e_2') \\
e \oplus (c'\ ?\ e_1 : e_2) & \rightarrow & c'\ ?\ e \oplus e_1 : e \oplus e_2 \\
(c'\ ?\ e_1 : e_2) \oplus e & \rightarrow & c'\ ?\ e_1 \oplus e : e_2 \oplus e \\
\ominus c'\ ?\ e_1 : e_2 & \rightarrow & c'\ ?\ \ominus e_1 : \ominus e_2 \\
(c\ ?\ c_1 : c_2)\ ?\ e_1 : e_2 & \rightarrow & (c \wedge c_1) \vee (\neg c \wedge c_2)\ ?\ e_1 : e_2
\end{array}
$$

Figure 10. Rewrite rules for translation to multi-way conditional form.

there exists $z \in \mathcal{S}$ such that $\curlyvee(z).1 = m \wedge \curlyvee(z).2 = k$ and since $p(\curlyvee(z).1 \bullet \curlyvee(z).2)$ we have $\chi^\bullet(p)$.

B.2 Proof of Theorem 4.3 and B.2

The proof relies on the following lemma, which defines the relation between divide predicates and divide functions:

Lemma B.2. $\Phi^\bullet(p, \odot) \implies \Psi^\bullet(\curlyvee, \odot)$ *under the assumption that* $\forall z \in \mathcal{S} : p(\curlyvee(z).1, \curlyvee(z).2)$.

Let us first prove this lemma. Suppose we have $(p, \odot)$ a valid solution to specification $\Phi^\bullet$, and a $\curlyvee$ function that satisfies:

$$\forall z \in \mathcal{S} : p(\curlyvee(z).1, \curlyvee(z).2)$$

Suppose $z \in \mathcal{S}$. We have $p(\curlyvee(z).1, \curlyvee(z).2) \implies f(\curlyvee(z).1 \bullet \curlyvee(z).2) = f(\curlyvee(z).1) \odot f(\curlyvee(z).2)$ since $(p, \odot)$ is a solution of $\Phi^\bullet$. Since f is permutation invariant, we can rewrite $p(\curlyvee(z).1, \curlyvee(z).2) \implies f(z) = f(\curlyvee(z).1) \odot f(\curlyvee(z).2)$. By construction of $\curlyvee$, $p(\curlyvee(z).1, \curlyvee(z).2)$ holds, therefore $f(z) = f(\curlyvee(z).1) \odot f(\curlyvee(z).2)$.

Since $p, \odot$ is a solution of $\Phi^\bullet$, $\chi^\bullet(p)$ holds. That is, $\forall m, k \in \mathbb{N}, \exists x, y \in \mathcal{S}^2 : |x| = m \wedge |y| = k \wedge p(x, y)$. Therefore, for any $k, m \in \mathbb{N}$ there are two sequences in the image of $\curlyvee$ (two sequences x, y such that $p(x, y)$ holds) of lengths k and m. If the sequences are in the image of $\curlyvee$, then there exists a sequence in the domain of divide, therefore $\chi(\curlyvee)$.

To prove Theorem 4.3, we prove the other realizability implication.

((3) $\implies$ (5)) Suppose we have a join function $\odot$ and a divide $\curlyvee$ such that $\Psi^\bullet(\curlyvee, \odot)$. Since the domain of $\curlyvee$ is countable we can define the predicate p as follows, for any sequences x and y: $p(x, y) = \exists \pi : (x, y) = \curlyvee(\pi(x \bullet y))$. The existence of a permutation π can be determined by enumerating all the permutations of $x \bullet y$ until dividing the permutation returns (x, y). If no such permutation is found, then there is no solution. We have then $x, y \in \mathcal{S}^2$ such that $p(x, y) \implies \exists z \in \mathcal{S}, \curlyvee(z) = (x, y)$, and so there is a permutation π such that $p(x, y) \implies f(\pi(z)) = f(x) \odot f(y)$. Since f is permutation invariant, we can rewrite $p(x, y) \implies f(x \bullet y) = f(x) \bullet f(y)$ and therefore $(p, \odot)$ is a solution of $\Phi^\bullet$ (noting the the generality constraints transfer as in the proof of Lemma B.1).

B.3 Proof of Proposition 4.5

(6 $\implies$ 5) Trivial: we do not need the permutation in the proof of 4.3 because of the splitting divide hypothesis.

Counterexample for the converse: Pareto is a counterexample for this particular case.

B.4 Ranking Lifting

Given a sequence x we denote by $\check{x}$ as the sequence of elements of x paired with their rank (position) in x. We denote by $\check{\pi}$ the function that projects a sequence of elements with their rank to the sequence of element in the original order given the rank of their elements. Given a function f, one can define a function $\check{f}$ by $\check{f}(\check{x}) = f(\check{\pi}(\check{x}))$. Then $\check{f}$ is permutation invariant. Remark the the constructed $\check{f}$ would not be an efficient function if there are no simplifications to make.

Suppose there exists a predicate p, a permutation function π and a join $\odot$ such that

$$\forall x, y \in \mathcal{S}, p(x, y) \implies f(\pi(x \bullet y)) = f(x) \odot f(y)$$

Then there is a predicate $\check{p}$ and join $\odot$ such that:

$$\forall x, y \in \mathcal{S}, \check{p}(\check{x}, \check{y}) \implies \check{f}(\check{x} \bullet \check{y}) = \check{f}(\check{x}) \odot \check{f}(\check{y}))$$

$\check{p}$ can be constructed simply by $\check{p}(\check{x}, \check{y}) = p(\check{\pi}(\check{x}), \check{\pi}(\check{y}))$. Since $\check{f}$ is permutation invariant, we have the guarantee that a divide operation that satisfies $\Psi^\bullet(\curlyvee, \odot)$ for $\check{f}$ exists. Remark that such a divide operation will be invertible since it operates on sequences of elements that remember their rank.

C Extras

C.1 Detailed Example: Rewriting POP

This example details the rewriting steps taken in order to discover a divide predicate in the introductory example of Section 6. We assumed that the starting state $\mathsf{POP}(x) = [s_1, s_2]$ is of length two. Let us compute a first unfolding of the function starting from this state. We want to compute the expression of $\mathsf{POP}(x \bullet [a_1])$ for some point a_1. Simply using the function definition we have:

$$
\begin{aligned}
\mathsf{POP}(x \bullet [a_1]) = {} & (s_1 \succcurlyeq a_1\ ?\ [s_1] : []) \\
& \bullet (s_2 \succcurlyeq a_1\ ?\ [s_2] : []) \\
& \quad \bullet (a_1 \succcurlyeq s_1 \wedge a_1 \succcurlyeq s_2\ ?\ [a_1] : [])
\end{aligned}
$$

We can distribute the concatenation operations inside the conditionals using the rewrite rule $(c\ ?\ a : b) \oplus d \rightarrow c\ ?\ a \oplus d : b \oplus d$. Distributing the second line in the first line of the

expression above yields:

$$(s_1 \gg a_1 \; ?[s_1] \bullet (s_2 \gg a_1 \; ? \; [s_2] : []) \\ : [] \bullet (s_2 \gg a_1 \; ? \; [s_2] : [])) \\ \bullet (a_1 \gg s_1 \wedge a_1 \gg s_2 \; ? \; [a_1] : [])$$

Let us write $c_1 = a_1 \gg s_1 \wedge a_1 \gg s_2$. Distributing the third line inside yields:

$$(s_1 \gg a_1 \; ?[s_1] \bullet (s_2 \gg a_1 \; ? \; [s_2] : []) \bullet (c_1 \; ? \; [a_1] : []) \\ : [] \bullet (s_2 \gg a_1 \; ? \; [s_2] : []) \bullet (c_1 \; ? \; [a_1] : []))$$

We used the right-distributivity rule so far. We can also use left-distributivity, rewriting the expression above as follows:

$$(s_1 \gg a_1 \; ? \\ (s_2 \gg a_1 \; ? \\ (c_1 \; ? \; [s_1] \bullet [s_2] \bullet [a_1] : [s_1] \bullet [s_2] \bullet []) \\ : (c_1 \; ? \; [s_1] \bullet [] \bullet [a_1] : [s_1] \bullet [] \bullet [])) \\ : (s_2 \gg a_1 \; ? \\ (c_1 \; ? \; [] \bullet [s_2] \bullet [a_1] : [] \bullet [s_2] \bullet []) \\ : \; (c_1 \; ? \; [] \bullet [] \bullet [a_1] : [] \bullet [] \bullet [])))$$

In the above expression, we have identified the branch expression that corresponds to $\text{POP}(x) \bullet \text{POP}([a_1])$. The other expressions cannot be directly constructed from $\text{POP}(x)$ and $\text{POP}([a_1])$. We can rewrite the complete expression by factoring the conditionals, i.e. $c \; ? \; (d \; ?a : e) \; : \; b \rightarrow c \wedge d \; ?a : (c \; ? \; e \; : b)$. Using this rule twice, we obtain the following expression:

$$(s_1 \gg a_1 \wedge s_2 \gg a_1 \wedge c_1 \;\; ? \;\; [s_1] \bullet [s_2] \bullet [a_1] \;\; : \;\; \ldots)$$

Where the else-branches of the conditional have been omitted. Note that the omitted expression can always be replaced by $\text{POP}(x \bullet [a_1])$, since it is the original expression being rewritten. We can conclude that:

$$\text{POP}(x \bullet [a_1]) = \\ s_1 \gg a_1 \wedge s_2 \gg a_2 \wedge a_1 \gg s_1 \wedge a_1 \gg s_2 ? \\ \text{POP}(x) \bullet \text{POP}([a_1]) : \text{POP}(x \bullet [a_1])$$

The rewriting steps for $\text{POP}(x \bullet [a_1, a_2])$ are similar.

C.2 Inductive Cost Inference of Skeletons

In this section, we describe how the cost of an expression skeleton $\dot{S}^k$ is inferred during the induction process. In the instance where $c = 2$ and the input function f is rightwards single pass, the induction is done on $f(x)$ (when it is not a scalar) and y. In the general case with $\mathcal{B}_\odot = (m_\odot, k, c)$, we have c **induction parameters indexed by** $0 < i_c \leq c$, and at each step one of the induction parameters is expanded. For example, when f is rightwards and $c = 2$, then parameter $i_c = 1$ is $f(x)$ and parameter $i_c = 2$ is y. The inputs of the join are directly related to the induction parameters, e.g if $k = 2$ then the inputs are $f(x)$ and $f(y)$.

At each induction step, the **cost vector** $\vec{m}$ **is of size** c. Intuitively, it represents the conjectures that the computation time of the join varies as a polynomial of degree $\vec{m}[i_c]$ with respect to the input of the join matching the induction parameter i_c, and the join computation is of asymptotic complexity $O(n^{max(\vec{m})})$.

Algorithm 1 presents how the cost of a skeleton $\dot{S}^k$ is inferred, given the skeleton $\dot{S}^k_{prev}$ at the previous step, and the cost $\vec{m}$ inferred at the previous step. It is a generalization of the pseudo-algorithm presented in Section 6.1, where the cases were limited to identifying whether a join is linear or constant time. The main difference here is that the subexpression relation can be more complicated, and more than one hole can appear in a new skeleton.

We assume that the skeletons are **always of degree** k. If the skeleton at the current step is of degree larger than k, then considering its *cost* is not necessary, since it does not even adhere to the required *shape*. If the degree is l less than k, the skeleton can always be considered as of degree k, where some of the holes $??_i$ for $l < i \leq k$ simply do not appear.

We also assume that we have a **mapping** $\mu : [1, k] \rightarrow [1, c]$ from the induction parameter indices to the holes indices. The mapping is simply required because the c outputs of the divide (which correspond to the induction parameters) are fixed for the problem, but the k inputs of the join can be chosen.

Algorithm 1: Computing the cost of an expression skeleton after an induction step.

Input: A context: the previous skeleton $\dot{S}^k_{prev}$ and its cost $\vec{m}$, and the induction parameter index $0 < i_c \leq c$.
Output: The updated $\vec{m}$ of the current skeleton $\dot{S}^k$
if $\dot{S}^k = \dot{S}^k_{prev}$ **then**
  $\vec{m}[i_C] = 0$;
else
  if $\exists ??_i \in \dot{S}^k \setminus \dot{S}^k_{prev}$ **then**
    foreach $??_i \in \dot{S}^k \setminus \dot{S}^k_{prev}$ **do**
      $d = Degree(??_i, \dot{S}^k \setminus \dot{S}^k_{prev})$;
      $\vec{m}[\mu(i)] = max(d, \vec{m}[\mu(i)])$;
  else
    $\vec{m}[i_c] = max(1, \vec{m}[i_c])$
return $\vec{m}$;

The algorithm implements the following intuition. If the new skeleton is the same as the previous skeleton, it means that for an additional step of induction on a given parameter, the time required to compute the join is unchanged, and therefore it must be constant. If the new skeleton is different, there must be a subexpression relation between $\dot{S}^k$ and $\dot{S}^k_{prev}$

(we are considering recursively defined functions). $\dot{S}^k \setminus \dot{S}^k_{prev}$ is the smallest tree constituted of the parts of $\dot{S}^k$ that could not be mapped by a bijection to subexpressions of $\dot{S}^k_{prev}$. To compute $\dot{S}^k \setminus \dot{S}^k_{prev}$, one must find a maximal bijection between subtrees of $\dot{S}^k$ and $\dot{S}^k_{prev}$. In the simplest case, $\dot{S}^k_{prev}$ is a subtree of $\dot{S}^k$.

When no holes appear in $\dot{S}^k \setminus \dot{S}^k_{prev}$, (there are no new holes in $\dot{S}^k$) then there are only constants that appear. Therefore, an induction step on parameter i_c implies an additional constant time step in the computation of $\dot{S}^k$, and the latter is linear on i_c.

If new holes appear, the index of the holes gives an indication that an induction step on parameter i_C requires an additional step of computation involving another parameter (or i_c itself). $Degree(??_i, \dot{S}^k \setminus \dot{S}^k_{prev})$ is an approximation of the computation required. If there is only one occurrence of $??_i$ in $\dot{S}^k \setminus \dot{S}^k_{prev}$ then $Degree(??_i, \dot{S}^k \setminus \dot{S}^k_{prev}) = 1$. If there are n occurrences of $??_i$, where n is the number of times parameter $\mu(i)$ has been unfolded during the induction process, then $Degree(??_i, \dot{S}^k \setminus \dot{S}^k_{prev}) = 2$. An implementation for this procedure is heuristic by nature: it is impossible to determine with certainty at each step the computational cost of the function being discovered. However, the inductive process permits the generalization of good observations and the invalidation of bad ones, which is the source of the robustness of our algorithm.

C.3 Synthesizing Conditional Auxiliaries

In this section, we present an algorithm that discovers a certain type of lifting, *conditional accumulators*. In Example 6.6, we give a high-level intuition of how the general *lifting* algorithm works on the LIS example.

Definition C.1 (Conditional accumulation)**.** A conditional accumulation $f : \mathcal{S} \to \text{Bool} \times D$ is a single-pass function defined by a boolean operation $\oslash : \text{Bool} \times Sc \to \text{Bool}$ and two scalar operations $\oplus^+, \oplus^- : D \times Sc \to D$ with two initial values $c_0 \in \text{Bool}$ and $f_0 \in D$ as follows:

$$f([]) = (c_0, f_0)$$

$$f(x \bullet [a]) = \textbf{let } c, s = f(x) \textbf{ in } (c \oslash a, c \oslash a \ ? \ s \oplus^+ a : s \oplus^- a)$$

We describe in Algorithm 2 an algorithm for the lifting procedure of the general auxiliary and predicate synthesis that synthesizes conditional auxiliaries. The algorithm relies on the `Normalize` procedure that rewrites an expression to $\mathcal{B}_\odot$-normal form, and a `Collect` procedure that collect the expressions of the auxiliaries for which a recursive definition needs to be found. `Collect` simply collects the expression that are not computed by the function under the expression skeleton $\dot{S}$.

Suppose that two successive unfoldings of f have been rewritten into MC-expressions:

$$e^{(u-1)} = \{{c_i}^{(u-1)} : {e_i}^{(u-1)} \mid 1 \le i \le n^{(u-1)}\}$$

Algorithm 2: Conditional auxiliary synthesis.

Input : Two successive unfoldings in mcnf:
$e^{(u-1)} = \{{c_i}^{(u-1)} : {e_i}^{(u-1)} \mid 1 \le i \le n^{(u-1)}\}$
and
$e^{(u)} = \{{c_i}^{(u)} : {e_i}^{(u)} \mid 1 \le i \le n^{(u)}\}$

Output : A conditional accumulation with operations $\oslash, \oplus^+, \oplus^-$

Normalize the branch expressions and collect auxiliaries to be computed.;
Eu =
`Collect`$\{{c_i}^{(u-1)} :$ `Normalize`$({e_i}^{(u-1)}) \mid 1 \le i \le n^{(u-1)}\}$;
Eu' = `Collect`$\{{c_i}^{(u)} :$ `Normalize`$({e_i}^{(u)}) \mid 1 \le i \le n^{(u)}\}$;
for $aux^{(u)} \in$ Eu' **do**
  Annotate the branches with the corresponding branch in the previous unfolding;
  $aux^{(u)} = \{{c_i}^{(u)}[{c_i}^{(u)} \Rightarrow {c_j}^{(u-1)}] : {aux_i}^{(u)} \mid 1 \le i \le n^{(u)}\}$;
  Find the different subtree isomorphisms, store them in the set Op;
  Op = $\emptyset$;
  for $\left({c_i}^{(u)}[{c_i}^{(u)} \Rightarrow {c_j}^{(u-1)}] : {aux_i}^{(u)}\right) \in aux^{(u)}$ **do**
    `Add(Op, Find(`$\oplus : {aux_i}^{(u)} = {aux_j}^{(u-1)} \oplus a[u]$`))`;
  if |Op| > 2 **then return**;
  else $\oplus^+, \oplus^-$ = Op;
  $I^{+(u)} = \left\{1 \le i \le n^{(u)} \mid {c_i}^{(u)}[{c_i}^{(u)} \Rightarrow {c_j}^{(u-1)}] : {aux_j}^{(u-1)} \oplus^+ a[u]\right\}$;
  $I^{-(u)} = \left\{1 \le i \le n^{(u)} \mid {c_i}^{(u)}[{c_i}^{(u)} \Rightarrow {c_j}^{(u-1)}] : {aux_j}^{(u-1)} \oplus^- a[u]\right\}$;
  $\varphi^{(u)} \leftarrow \left(\forall i \in I^{+(u)}, {c_i}^{(u)} \Rightarrow c^{(u)}\right) \wedge \left(\forall i \in I^{-(u)}, {c_i}^{(u)} \Rightarrow \neg c^{(u)}\right) \wedge \left(c^{(u)} = c^{(u-1)} \oslash a[u]\right) \wedge \varphi^{(u-1)}$;
  $\oslash \leftarrow$ `Synthesize(`$\exists \oslash, \varphi$`)`

$$e^{(u)} = \{{c_i}^{(u)} : {e_i}^{(u)} \mid 1 \le i \le n^{(u)}\}$$

The MC-expression of unfolding $u-1$ has $n^{(u-1)}$ branches, and the MC-expression of unfolding u has $n^{(u)}$ branches.

The normalization procedure `Normalize` is applied to the expressions in the branches, and then the sub-expressions that need to be computed are collected in each branch of the MC-expression of the unfolding. Then an MC-expression is built from the result of the collection. When collecting a subexpression, the information about where the subexpression appears is used to identify different types of auxiliaries.

For each type of auxiliary *aux* we have:

$$aux^{(u-1)} = \{{c_i}^{(u-1)} : {aux_i}^{(u-1)} \mid 1 \le i \le n^{(u-1)}\}$$

$$aux^{(u)} = \{{c_i}^{(u)} : {aux_i}^{(u)} \mid 1 \leq i \leq n^{(u)}\}$$

The conditions of the branches of the second unfolding are annotated with the branch they are issued from, that is a branch condition in the previous unfolding. The annotation, given below, is a logical implication that can be checked by a solver:

$$aux^{(u)} = \{{c_i}^{(u)}[{c_i}^{(u)} \Rightarrow {c_j}^{(u-1)}] : {aux_i}^{(u)} \mid 1 \leq i \leq n^{(u)}\}$$

The subtree relations mapping the expressions ${aux_j}^{(u-1)}$ to subexpressions of ${aux_j}^{(u)}$ are then computed, given the annotations computed previously. That is, subexpression mappings are computed between ${aux_i}^{(u-1)}$ and ${aux_j}^{(u)}$ if ${c_j}^{(u)} \implies {c_i}^{(u-1)}]$. If the goal auxiliary is a conditional accumulation, then there will be only two types of subexpression mappings, which correspond to the operators $\oplus^+$ and $\oplus^-$. If we denote the inputs read at unfolding u by $a[u]$, the auxiliary computation has the following form:

$$aux^{(u)} = \bigcup_{i \in I^+(u)} \{{c_i}^{(u)}[{c_i}^{(u)} \Rightarrow {c_j}^{(u-1)}] : {aux_j}^{(u-1)} \oplus^+ a[u]\} \bigcup_{i \in I^-(u)} \{{c_i}^{(u)}[{c_i}^{(u)} \Rightarrow {c_j}^{(u-1)}] : {aux_j}^{(u-1)} \oplus^- a[u]\}$$

Once the two operators $\oplus^+$ and $\oplus^-$ have been found, the conditional accumulator can be synthesized. The specification $\varphi^{(u)}$ gives the constraints that the expression of the conditional accumulator has to satisfy.

The success of the conditional auxiliary synthesis algorithm depends on the `Normalize` procedure. That is, if there is a conditional auxiliary that allows to parallelize the function and we have and ideal `Normalize` procedure, then the algorithm discovers this auxiliary. In practice the `Normalize` procedure is implemented in a similar way as the one described in [8, 9]. That is, a symbolic expression is rewritten by applying a set of rewrite rules that decrease a predetermined cost. This cost is defined to be minimal when a $\mathcal{B}_\odot$-normal form is reached during the rewrite process.

Remark that this algorithm can be extended to recognize more than two accumulation schemes. For n operators, $n-1$ conditionals will need discovering.

Example C.2 (Expressions of the conditional auxiliary for LIS)**.** Recall the LIS function, given in Example 6.3. We remind the MC-expression obtained for the second unfolding of the function:

$$\begin{array}{lll} 1: & ml_0 \uparrow cl_0 + 1 \uparrow cl_0 + 1 + 1 & \textbf{if } (a_2 > a1) \wedge (a_1 > prev_0) \\ 2: & ml_0 \uparrow cl_0 + 1 \uparrow 0 & \textbf{if } (a_2 \leq a1) \wedge (a_1 > prev_0) \\ 3: & ml_0 \uparrow 0 \uparrow 0 + 1 & \textbf{if } (a_2 > a1) \wedge (a_1 \leq prev_0) \\ 4: & ml_0 \uparrow 0 \uparrow 0 & \textbf{if } (a_2 \leq a1) \wedge (a_1 \leq prev_0) \end{array}$$

In Example 6.4 we have deduced the expression of predicate that isolates the branches that are not normalizable with an auxiliary and the branches that require an auxiliary. The auxiliary synthesis focuses on the branches that requires an auxiliary synthesis. By assuming $(a_1 \leq prev_0)$ the MC-expression can be simplified to:

$$\begin{array}{l} 1: ml_0 \uparrow cl_0 + 1 + 1 \textbf{ if } (a_2 > a1) \\ 2: ml_0 \uparrow cl_0 + 1 \uparrow 0 \textbf{ if } (a_2 \leq a1) \end{array}$$

The same process can be repeated for the third unfolding of LIS. The expression of $\mathsf{LIS}(x \bullet [a_1, a_2, a_3])$ is written to an MC-expression, the same predicate expression is deduced to separate the branches that require and unfolding from those who do not. The resulting MC-expression after simplification is:

$$\begin{array}{lll} 1'(1): & ml_0 \uparrow cl_0 + 1 + 1 + 1 & \textbf{if } (a_3 > a_2) \wedge (a_2 > a1) \\ 2'(2): & ml_0 \uparrow cl_0 + 1 \uparrow 1 & \textbf{if } (a_3 > a_2) \wedge (a_2 \leq a1) \\ 3'(1): & ml_0 \uparrow cl_0 + 1 + 1 \uparrow 0 & \textbf{if } (a_3 \leq a_2) \wedge (a_2 > a1) \\ 4'(2): & ml_0 \uparrow cl_0 + 1 \uparrow 0 & \textbf{if } (a_3 \leq a_2) \wedge (a_2 \leq a1) \end{array}$$

A pattern is emerging between the sums of ones. In the previous MC-expression, each branch has been annotated with the predecessor branch number in the previous unfolding. For example, the expression of branch 1' comes from branch 1. Note that a +1 has been added to the expressions that require lifting only in branch 1. In other branches, we have deduced that the auxiliary does not need changing. By extrapolating the sequence $(a_2 > a1), (a_3 > a_2) \wedge (a_2 > a1), \ldots$ one can discover the conditional auxiliary for the LIS example.

***Example* C.3.** In Example 6.6, we determine that a function g_1 such that $g_1([a_1, a_2]) = a_1 < a_2 \ ? \ 1 + 1 : 1$ is required for a join to exist without a predicate (for all the branches in the unfoldings of f to be normalizable). The same normalization process yields the expression of the next unfolding, which can be written as $g_1([a_1, a_2, a_3]) = a_2 < a_3 \wedge a_1 < a_1 \ ? \ g_1([a_1, a_2]) + 1 : g_1([a_1, a_2])$. Some subexpressions have been replaced by the expressions of the previous unfolding to highlight recursion. Then recursion discovery deduces that g_1 can be computed, with two components $(cond, aux)$, by the following recursion relation:

$$g_1(x \bullet [a]) = \textbf{let } c = g_1(x).cond \wedge (f(x).prev < a) \textbf{ in } \textbf{let } b = g_1(x).aux \textbf{ in } (c, c \ ? \ b + 1 : b)$$

The final lifting of LIS is $\mathsf{LIS}'(x) = (\mathsf{LIS}(x), g_1(x), head(x))$ since *head* is the trivial result of recursion discovery from the unfoldings of g_2 in Example 6.5. ⌟

C.4 Divide Function Synthesis

Conditional predicate. If the divide predicate $p(x, y)$ is defined by a constant time computation on a suffix of x and prefix of y, then the splitting divide function is constructed as follows. First, it splits its input z randomly into x, y such that $z = x \bullet y$, and then computes the prefix y_0 of y ($y = y_0 \bullet y_{rest}$) such that $p(x \bullet y_0, y_{rest})$ holds. This computation is linear time in the length of the average prefix length y_0.

General case. The sketches are Racket functions that are solved by the Rosette [25] syntax-guided synthesis solver. A pivot partition sketch with two outputs has the following form:

```
(define (div X)
    (let ([pivot (foldl (lambda (x piv)
        (if ?? x piv)) (car X) X)])
    (partition (lambda (x) ??) X)))
```

where ?? stands for a hole that has to be filled with an expression. A sketch for more than two partitions is constructed by nesting partitions. For example, the following sketch can be completed to implement a function that partitions a list X into three partitions:

```
(define (div X)
  (let ([pivot (foldl (lambda (x piv)
                    (if ?? x piv)) (car X) X)])
    (let*-values
      ([(X1 Y) (partition (lambda (x) ??) X)]
       [(X2 X3) (partition (lambda (x) ??) Y)])
    (values X1 X2 X3))))
```

In syntax-guided synthesis, the user provides the grammar of expressions from which an expression can be drawn to fill a hole. In our case, the grammar depends on the operators used in the predicate that the divide has to match, and the data type of the elements of the list. The holes in the above sketches are all of boolean type, and complete by expressions in the following grammar with start symbol P:

$$\begin{array}{lcl} P & \rightarrow & C \wedge P \mid C \vee P \\ C & \rightarrow & \neg C \mid A\ op_{>}\ A \mid b \\ A & \rightarrow & A\ op_{+}\ A \mid x \mid n \end{array}$$

where $op_{>}$ is a comparison operator that appears in the predicate or reference implementation, op_{+} is an arithmetic operator appearing in the predicate, b is a boolean variable and x an integer variable, and n is an integer constant. If there are no boolean variables or integer variables (this depends on the input data type), then the grammar is restricted accordingly. If no solution is found with the restricted set of operators, the set of operators is extended to include every comparison operator of the language and every arithmetic operator. Practically, the tool tries out both possibilities in parallel.

The expression grammar for the first hole for the pivot selection also contains the constant false `#f`, which allows a solution to skip the pivot selection, and use the first element of the list as pivot.

In our set of benchmarks, no example required a partition that is more than linear time. We test the divide synthesis on a fabricated example, where the following predicate p had to be satisfied:

$$\forall x, y \in \mathcal{S} : p(x, y) = (\forall a \in x, \forall b \in y : a < b) \wedge |x| > |y|$$

A partition that selects its pivot in quadratic time is synthesized by extending the sketch with another fold function for the pivot selection, and a restricted space for hole completion.

	Parsynt	GraSSP
Count 1(0*)2	1.5	2.1
Count 1*2*3	3.3	11.6
Max dist. between 1s	2.8	1.8
Max sum between 0s	2.0	5.0

Table 2. Comparison of Parsynt with GraSSP on the benchmarks that have a solution for all tools.

D Additional Experimental Results

D.1 Additional Experiments for Table 7(a)

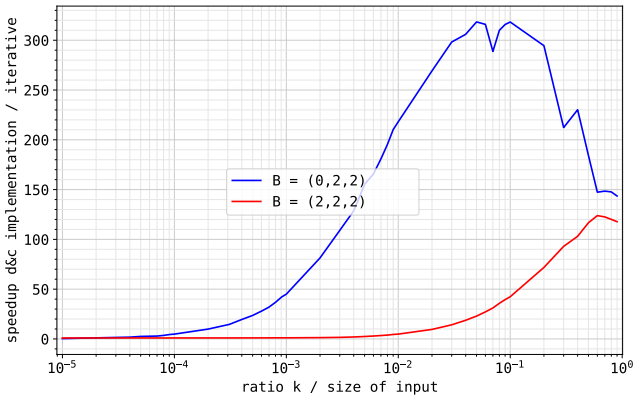
Table 2 compares our tool Parsynt with GraSSP [11] on the benchmarks for which both can synthesize solutions. Remark that our tool is more expressive as illustrated by Table 7(a), and it also synthesizes two solutions for the problem for these benchmarks: a splitting divide solution, and a solution with lifting.

D.2 Quality of Synthesized Solutions from Table 1

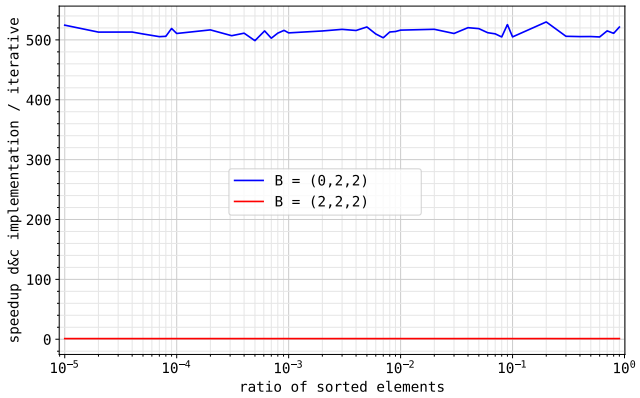
In this section, we give additional insight on the quality of the solutions synthesized for the benchmarks given in Table 1. In some cases the performance depends on the input distribution, which is specific to each benchmark. In the case of the POP example, one possible measure of the distribution is to count how many points are optimal in the input. In Figure 4, the performance of the POP divide-and-conquer implementations is analyzed with varying ratios of optimal points to total number of points in the input. Figure 11 presents similar plots, which are explained in the following paragraph.

Sorting. The first benchmark is the selection sort algorithm, which takes $O(n^2)$ algorithm, and the output, for budget $\mathcal{B}_{\odot} = (0, 2, 2)$ is the quicksort algorithm. In the solution synthesized, the pivot chosen is the first element of the input of the divide. One can improve on this choice by taking a random pivot, but our default solution performs well in average, as expected of a quicksort algorithm.

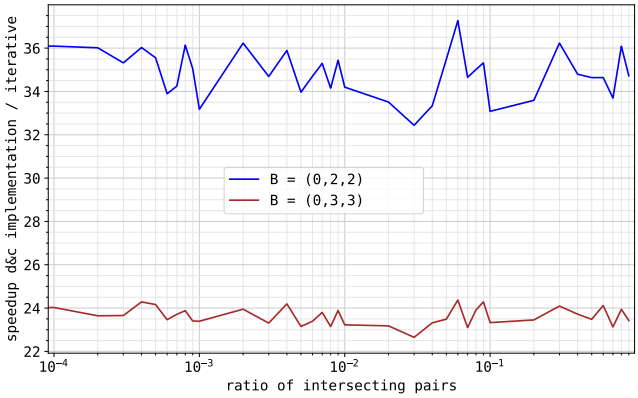
k-largest. This benchmark selects the k-largest elements in a list. The input algorithm is in $O(nk)$, similar to the selection sort above but the size of the output is of size k only. In Figure 11(a), we plot the speedup of two divide-and-conquer implementations compared to the naive sequential input, with varying values of k, represented as the ratio of k to the input size. On the left of the graph, for small values of k, the speedup is small: $O(nk)$ is comparable to n. However, for larger values of k, the implementation synthesized by Parsynt for $\mathcal{B}_{\odot} = (0, 2, 2)$ is significantly faster. The naive solution, for a budget $\mathcal{B}_{\odot} = (2, 2, 2)$ consists in splitting the input at random, computing the k-largest of each subdivision, and in the join computing the k-largest of the concatenation



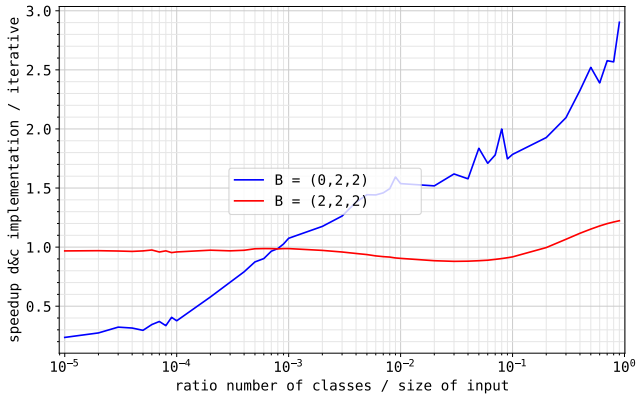
(a) *k-largest*: speedup of two divide-and-conquer implementations with respect to the input single-pass implementation, with increasing ratio of k to total input size. Input size is 10^5. The synthesized solution for $\mathcal{B}_{\odot} = (0, 2, 2)$ is compared to a naive quadratic solution (where the budget would be $\mathcal{B}_{\odot} = (2, 2, 2)$).



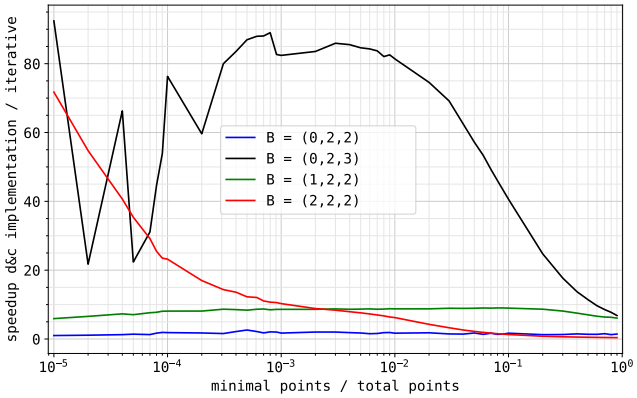
(b) *Closest pair*: speedup of two divide-and-conquer implementations with respect to the input single-pass implementation. ParSynt synthesizes the solution with $B = (0, 2, 2)$. The input is a sorted array of 10^5 elements, which are then swapped, and the y-axis represents the ratio of elements that are still sorted in the input.



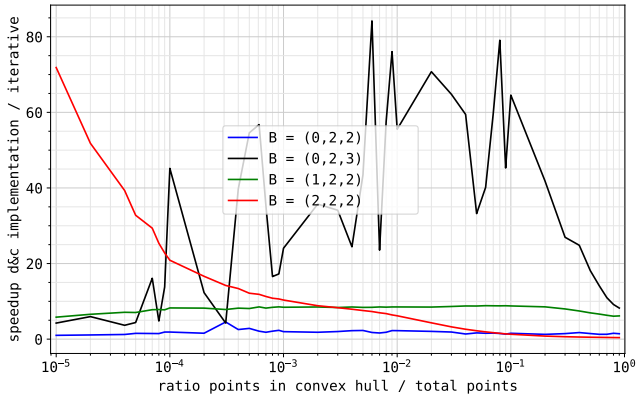
(c) *Intersecting intervals*: speedup of two divide-and-conquer implementations with respect to the input single-pass implementation, with varying ratio of number of pairs of intersecting intervals to input size.



(d) *Histogram*: speedup of two divide-and-conquer implementations with respect to the input single-pass implementation, with varying ratio of number of input classes to total input size.



(e) *Minimal points*: speedup of three divide-and-conquer implementations with respect to the input single-pass implementation, with varying ratio of number of minimal points in input to total input size.



(f) *Quadrant orthogonal convex hull*: speedup of two divide-and-conquer implementations with respect to the input single-pass implementation, with varying ratio of number of points in the convex hull to total input size.

Figure 11. Additional plots evaluating the quality of the code synthesized for the benchmarks in Table 1. Divide-and-conquer implementations of *k-largest* (a), *closest pairs* (b), *intersecting intervals* (c), *histogram* (d), *minimal points* (e) and *quadrant orthogonal convex hull* (f) are evaluated, with the speedup compared to the single-pass implementations with a single thread plotted with varying input distributions in each case.

of the two partial results. This solution performs worse than the one synthesized by Parsynt, except for small values of k, but in that case the divide-and-conquer implementations do not yield a significant performance advantage.

Closest pair. This benchmark is an algorithm that search a pair of integers in a list of integers, such that the distance between the two elements is minimal. The input algorithm is a naive $O(n^2)$ algorithm that explores all the possible pairs. Figure 11(b) shows the performance of the synthesized solution. We also added a *naive* solution, which would be synthesized for $\mathcal{B}_\odot = (2, 2, 2)$, but this solution does not have any performance advantage. The synthesized solution performs well, independently of the distribution of the input data, for the measure chosen. Note that the speedup depends on the input size, and varies as $\frac{n}{\log n}$. In this instance, the theoretical speedup would be 25000×, but we observe 500×.

Intersecting intervals. The input algorithm in this benchmarks checks whether there is a pair of intersecting intervals within a list of intervals. The input algorithm is a naive $O(n^2)$ algorithm that checks every pair. Parsynt synthesized two implementations. The implementation with $\mathcal{B}_\odot = (0, 2, 2)$ requires a simple lifting that consists in remembering the intervals with the smallest lower bound and the largest upper bound. The join is constant time and takes the disjunction of the partial result and whether the intervals in the lifting intersect. The implementation with $\mathcal{B}_\odot = (0, 3, 3)$ splits the space in three subspaces: given a pivot p, all the intervals that intersect with p, all the intervals whose upper bound is smaller than p's lower bound, and all the interval whose lower bound is larger than p's upper bound. Remark that is the first partition has more than two elements, there is (trivially) a pair of intersecting intervals. However the solution synthesized does not benefit from this simple intuitive optimization. The comparison of the two implementations is done in Figure 11(c), for varying ratios of intersecting intervals. Remark that the relative speedup is independent from this measure, but the solution with $\mathcal{B}_\odot = (0, 3, 3)$ performs worse since more work needs to be done in the divide function.

Histogram. The *histogram* algorithm is a well known example. In our paper, we use an implementation using only lists. The input is a list of pairs, where the first element is the class of the element, and the second element is the value of the element. The input algorithm is in the worst case $O(n^2)$, if every element has a different class. Figure 11(d) shows the performance of the synthesized solution, for varying number of classes as ratio to the input size.

Minimal points. In this benchmark, the algorithm computes the set of points that are minimal within a set of points on a plan, where minimal means that there is no other point that has both a lower x and a lower y coordinate. Four solutions are synthesized. The speedups of each implementation with respect to the sequential input implementation is represented in Figure 11(e), represented on the graph for a varying ratio of minimal points to the total input size.

```
Point[] O = [];
int i,j;
for(i = 0; i < n; i++){
  bool b1, b2, b3, b4 = true;
  for(j = 0; j < n; j++){
    b1 = b1 && comp1(S[i], S[j]);
    b2 = b2 && comp2(S[i], S[j]);
    b3 = b3 && comp3(S[i], S[j]);
    b4 = b4 && comp4(S[i], S[j]);
  }
  if(b1 || b2 || b3 || b4)
    O += [S[i]];
}
```

Figure 12. Computing the orthogonal convex hull of S.

Quadrant orthogonal convex hull. This algorithm computes the set of points that form an orthogonal convex hull, for points that are in one quadrant of the 2D space. The four solutions synthesized are compared in Figure 11(f), for a varying ratio of points in the orthogonal convex hull to total size of input. The performance of the solution for budget $\mathcal{B}_\odot = (0, 2, 3)$ is highly variable for a fixed ratio but different generated inputs, since it depends on a good selection of a pivot. This can be mitigated by simple adjustments to the pivot selection. The solution with three partitions is more sensitive to these changes than the other solutions, because the efficiency depends on how much is discarded in one of the partition. In the other solutions, no points are discarded after the divide function. This sensitivity to the pivot selection explains the high variability of the performance of the solution plotted with a black line.

E Benchmarks

This section gives a detailed explanation of the benchmarks used in this paper. Section E.1 explains the benchmarks that show the limitation of our tool Parsynt. Section E.2 gives the input implementations of the benchmarks for which the tool succeeds, that is, at least one divide-and-conquer implementation is synthesized.

E.1 Limitations of Parsynt

In this section, we briefly describe three different benchmarks that showcase the limitations of the synthesis steps in our method. We give the input implementation and outline where the difficulty in the automated synthesis process lies.

E.1.1 Orthogonal Convex Hull

The orthogonal convex hull of a set of points S in the 2D plane is the smallest area orthogonal polygon O such that (i) each point p in S lies inside O, and the intersection of O with

```
bool a, b = false;
int cl = 0;
int ml = 0;
int i;
for(i = 0; i < n; i++) {
  cl = A[i] = 3 && b || a || b ? cl + 1 : 0;
  ml += A[i] == 3 && b ? max(ml, cl) : ml;
  b = A[i] == 2 && a || A[i] == 0 && d;
  a = A[i] == 1 || A[i] == 0 && a;
}
return count;
```

Figure 13. Longest substring of A matching `1(0*)2(0*)3`

any horizontal or vertical line is either empty, or exactly one segment.

The implementation in Figure 12 stems from the observation that a point is in the orthogonal convex hull iff all the other points are all in one of the four quadrants that are defined by splitting the space with a vertical and a horizontal line passing through the point. The four comparison functions are `comp1(p,q)= p.x >= q.x && p.y >= q.y`, `comp2(p,q) = p.x >= q.x && p.y <= q.y`, `comp3(p,q)= p.x <= q.x && p.y >= q.y` and `comp4(p,q)= p.x <= q.x && p.y <= q.y`. After the inner loop, one of the booleans is true iff the point `S[i]` is in the orthogonal convex hull `0`.

This implementation requires a non-trivial lifting to synthesize a single-pass function. Intuitively, a point is optimal for one of the four reasons encoded in each of the different booleans computed in the inner loop. However, once the point has been added to the convex hull, this implementation forgets about why it is optimal. In a single pass function, the points of set are read sequentially. But the list of optimal points `0` does not contain enough information to decide which points must be kept in `0` and which ones must be removed when adding a new point to the set. To solve this problem, one can lift the list `0` to some list `0'` that contains quintuples, where the first element is a point, and the four other elements are the booleans `b1, b2, b3, b4` that were computed when adding the point.

E.1.2 Longest Substring Matching $1(0^*)2(0^*)3$

The program in Figure 13 computes the length of the longest substring matching $1(0*)2(0*)3$ using two boolean variables `a, b` and one counter variable `count`. The splitting divide function for this benchmark ensures that the substrings matching $1(0*)2(0*)3$ cannot be split arbitrary in the middle. While the verification of the solution in a bounded case only takes a few seconds, the complexity lies in inferring all the possible ways the regular expression could be split. Our deductive synthesis procedure is currently unable to do so, but this exercise would take significant effort, even for a good programmer.

```
int i, j;
for(i =0; i < n; i++){
  b = true;
  d = dist(S[i], c);
  for(j = 0; j < n; j++){
     b = b && ! col(S[i], S[j], c)
             || dist(S[j], c) >= d;
  }
  if(b) Y += [S[i]];
 }
```

Figure 14. Computing the encircling set of a point `c`

E.1.3 Encircling Set of a Point

Suppose we have a set X of points in the plane and a point c. We want to compute the set of encircling points Y of c, which is the subset of points of X such that for any of the points a in this subset, there is no point in X between a and c.

A quadratic implementation is presented in Figure 14. The squared distance between two points is `dist(p,p')= sq(p.x - p'.x)+ sq(p.y - p'.y)` and the colinearity function is `col(p_1, p_2, p_3)= p_1.x(p_2.y - p_3.y)+ p_2.x(p_3.y - p_1.y)+ p_3.x( p_1.x - p_2.y)= 0`.

The difficulty in automatically synthesizing a divide-and-conquer algorithm for this example lies in the non-linear operations in the distance and colinearity functions. We could not verify the correctness of the hand written solutions for the functional translation, divide synthesis and join synthesis steps in less than 10 minutes each with Rosette, the syntax guided synthesis tool we use.

E.2 Code of the Benchmarks

Section E.2.1 describes the input algorithm for each of the benchmarks in Table 1. Section E.2.2 describes the benchmarks of Table 7(a). The implementations presented here are written in a simple imperative language with lists, and list concatenation is the operator `+`.

E.2.1 Benchmarks of Table 1

- *Sorting.* The input algorithm for sorting is a selection sort algorithm. The implementation is presented in Figure 15. The implementation presented here is single-pass: each input element is read only once, and the inner loop iterates over the state (the list of sorted elements) to insert each new element. For $\mathcal{B}_{\odot} = (0, 2, 2)$, the solution synthesized is the quicksort algorithm.
- *k-largest.* This benchmark is an $O(nk)$ algorithm that collects the k largest element in an input list. Figure 16 is a single-pass input implementation: each element of the input list is read only once and stored in the list part of the state if it is larger than some element already stored, or if there are not yet k elements stored.

- *Closest pair.* Figure 17 is an implementation of an algorithm that computes the closest pair of elements in an input of integers. The distance between to elements of the list a and b is $|a - b|$. The input implementation is not single pass, and first needs to be translated to a single pass version using the technique described in [10].
- *Intersecting intervals.* This algorithm returns a boolean representing whether there is a pair of (distinct) intersecting intervals in an input list, where an interval $[lo, hi]$ is a pair of an integer lower bound lo and integer upper bound $[hi]$. Figure 18 presents an $O(n^2)$ implementation, that needs to be translated to a single-pass implementation.
- *Histogram.* We model the histogram as a list of pairs of class and count, not as a map. Therefore, updating the count for a given class takes linear time. The input algorithm , presented in Figure 19, is worst case $O(n^2)$, if every cell of the input array has a different class.
- POP. In the main body of the paper, we presented the code in Figure 3. Parsynt accepts a more naive algorithm, and then translates it to the code presented in the paper using the technique described in [10]. Figure 20 presents an input iterative implementation, and remark that this implementation is not single-pass.
- *Minimal points.* The minimal points of a set of points on a plane are a subset of the points such that all the other points have either a greater x- or y-coordinate. Figure 21 presents an implementation that is not single-pass and requires the functional translation step described in [10].
- *Quadrant orthogonal convex hull.* The implementation of Figure 22 collects the points that are in one quadrant of the orthogonal convex hull of a set of points with integer coordinates. In our example, the quadrant is the upper left part of the hull.

E.2.2 Benchmarks of Table 7(a)

All the benchmarks in Table 7(a) are linear time single-pass function over an input list. For each of these benchmarks, there are two solutions: a solution with a splitting divide (Definition 4.4) and a solution with a lifting, in which case the divide is the arbitrary splitting. The algorithms in this category can be seen as an integer.

- *Count* $1(0+)$. The implementation in Figure 23 counts the number of occurrences of substrings matching $1(0^+)$ in the input list in a single pass.
- *Count* $1(0^*)2$. In this benchmark, we have an algorithm that counts the number of substrings that match the regular expression $1(0^*)2$. The code is in Figure 24.
- *Count* $1^*2^*3^*$. The benchmark is similar to the previous ones, but now we count the number of occurrences of $1^*2^*3^*$. The code is in Figure 25.

```
list<int> tmp;
bool added;
int i, j;
list<int> sorted = [];
for(i = 0; i < n; i++) {
  tmp = [];
  added = false;
  for(j = 0; j < sorted.length(); j++) {
    if(!added && A[i] > sorted[j]) {
      tmp += [A[i],sorted[j]]; added = true;
    } else {
      tmp += [sorted[j]];
    }
  }
  if(!added) tmp += [A[i]];
  sorted = tmp;
}
return sorted;
```

Figure 15. *Sorting*: selection sort implementation sorts an input list A of length n.

```
list<int> tmp;
bool added;
int i, j, counter;
list<int> klargest = [];
for(i = 0; i < n; i++) {
  tmp = [];
  added = false;
  count = 0;
  for(j = 0; j < klargest.length(); j++) {
    if(!added && A[i] > klargest[j]) {
      tmp += [A[i]];
      added = true;
      count += 1;
    if(count < k - 1){
      tmp += [klargest[i]];
      count += 1;
    }
  }
  if(!added && count < k - 1) tmp += [A[i]];
  klargest = tmp;
}
return klargest;
```

Figure 16. *k-largest*: the algorithm returns the k largest elements in the input array A of length n. We assume $k < n$ and $k > 2$.

- *Max sum between ones.* The algorithm in Figure 26 counts the maximum sum of elements between two ones, in a list of integers.
- *Max distance between zeroes.* The algorithm in Figure 27 measures the maximum distance between two zeroes in a list of integers.

```
bool added;
int i, j;
int dist, min_dist;
pair p; // a struct {a : int; b : int}
p = {a = min(a[0],a[1]), b=max(a[0], a[1])};
min_dist = abs(p.b-p.a);
for(i = 0; i < n; i++) {
  for(j = 0; j < n; j++){
    dist = abs(A[i] - A[j]);
    if(dist > 0 && dist < min_dist) {
      min_dist = dist;
      p = {a = min(a[i],a[i]),
           b = max(a[j], a[j])};
    }
  }
return p;
```

Figure 17. *Closest pair*: the algorithm returns the pair of (distinct) closest elements in the input array A of length n.

```
// A is a list of
// struct {lo : int; hi : int}
bool intersect = false;
int i,j;
for(i = 0; i < n; i++) {
  for(j = 0; j < i; j++){
    intersect = intersect ||
      (!((x.lo == y.lo) && (x.hi == y.hi))
        && ((x.lo < y.hi) && (x.hi > y.lo)));
  }
}
return intersect;
```

Figure 18. *Intersecting intervals*: the algorithm returns whether there is a pair of intersecting intervals in the input array A of length n.

- *Largest peak.* In Figure 29, we give a single pass algorithm to compute the largest peak in a list of integers. A peak is a sublist of positive elements only, and the largest peak is the sublist, such that the sum of its elements is maximal. In other words, we compute the maximum segment sum of segments of positive elements.
- LIS. The implementation in Figure 28 is the imperative code for Example 6.3.
- *Longest* 1^*. The code in Figure 30 computes the length of the longest block of continuous ones in a list of integers.
- *Longest* $1(0^*)2$. The code in Figure 31 computes the length of the largest substring matching $1(0^*)2$ in a list of integers, in a single pass.

```
// A is a list of
// struct {key : int; count : int} data
list<data> hist = [];
list<data> tmp = [];
int i,j;
bool b;

for(i = 0; i < n; i++) {
  b = false;
  tmp = [];
  for(j = 0; j < hist.length(); j++){
      if(hist[j].key == A[i].key) {
        tmp += [{key = A[i].key;
                 count = hist[j].count +
                         A[i].count}];
        k = true;
      } else {
        tmp += [hist[j]];
      }
  }
  if(!k) hist += {A[i]};
  hist = tmp;
}
return hist;
```

Figure 19. *Histogram*: the algorithm returns the histogram of an array A of length n. The histogram is a list of pairs of key and count.

```
// A is a list of
// struct {x : int; y : int} point
list<point> optimal_points = [];
bool optimal;
for(i = 0; i < n; i++){
  optimal = true;
  for(j = 0; j < n; j++){
    is_opt = is_opt &
      ((A[i].x >= A[j].x) || (A[i].y >= A[j].y));
  }
  if(is_opt) optimal_points += [A[i]];
}
return optimal_points;
```

Figure 20. POP: this algorithm is an alternative to the algorithm presented in Figure 3, and is not single pass.

- *Longest even* 0^*. The code in Figure 32 computes the lengths of the longest block of zeroes of *even* length in a list of integers.
- *Longest odd* $(10)^*$. In Figure 33, we show an algorithm that computes the longest block matching $(01)^*$ of odd length, in a list of integers.

```
// A is a list of
// struct {x : int; y : int} point
list<point> minpoints = [];
bool bx;

for(i = 0; i < n; i++){
  bx = false;
  for(j = 0; j < n; j++){
    bx = bx || (A[j].x < A[i].x
               && A[j].y < A[i].y);
  }
  if(!bx) minpoints += [A[i]];
}
return minpoints;
```

Figure 21. *Minimal points.*

```
// A is a list of
// struct {x : int; y : int} point
bool is_hull;
list<point> hull = [];
point p1, p2;

for(i = 0; i < n; i++){
  is_hull = true;
  p1 = A[i];
  if(p1.x <= 0 && p1.y >= 0){
    for(j = 0; j < n; j++){
      p2 = A[j];
      is_hull &= p1.x <= p2.x || p1.y >= p2.y;
    }
    if(is_hull) hull += [A[i]];
  }
}
return hull;
```

Figure 22. *Quadrant Orthogonal Convex Hull.*

```
bool s0 = false;
bool s1 = false;
int scount = 0;
int i;
for(i = 0; i < n; i++){
  scount += (s1 && (A[i] != 0)) ? 1 : 0;
  s1 = (A[i] == 0) && (s0 || s1);
  s0 = A[i] == 1;
}
return scount;
```

Figure 23. *Count* $1(0+)$.

```
bool s0 = false;
int c = 0;
int i;
for(i = 0; i < n; i++){
  c += s0 && A[i] == 2 ? 1 : 0;
  s0 = A[i] == 1 || (s0 && A[i] == 0);
}
return c;
```

Figure 24. *Count* $1(0^*)2$.

```
bool s1 = false;
bool s2 = false; bool s3 = false;
bool fin = false; int c = 0;

for(i = 0; i < n; i++) {
    c = c + ((x == 3 && (s2 || s1)) ? 1 : 0);
    s2 = (x == 2) && (s1 || s2);
    s1 = x == 1;
}
return c;
```

Figure 25. *Count* $1^*2^*3^*$

```
int ms = 0;
int cs = 0;
for(i = 0; i < n; i++){
    cs = A[i] != 1 ? cs + A[i] : 0;
    ms = max(ms, cs);
}
return ms;
```

Figure 26. *Max sum between ones.*

```
int md = 0;
int cd = 0;
int i;
for(i = 0; i < n; i++){
    cd = A[i] != 0 ? cd + 1 : 0;
    md = max(md, cd);
}
return md;
```

Figure 27. *Max distance between zeroes.*

```
int cl = 0, ml = 0;
int prev = A[0];
int i;
for(i = 1; i < n; i++){
  cl = prev < A[i] ? cl + 1 : 0;
  ml = max(ml, cl);
  prev = A[i];
}
return ml;
```

Figure 28. LIS.

```
int cmo =0, lpeak = 0;
int i;
for(i = 1; i < n; i++){
  cmo = A[i] > 0 ? cl + A[i] : 0;
  lpeak = max(cmo, lpeak);
}
return lpeak;
```

Figure 29. *Largest peak.*

```
int ml = 0, len = 0;
int i;
for(i = 1; i < n; i++){
  len = A[i] == 1 ? len + 1 : 0;
  ml = max(ml, len);
}
return ml;
```

Figure 30. *Longest* 1^*.

```
bool s0 = false, s1 = false;
int ml = 0, len = 0;
int i;
for(i = 1; i < n; i++){
  s1 = s0 && A[i] == 2;
  s0 = A[i] == 1 || (A[i] == 0 && s0);
  len = (s1 || s0) ? len + 1 : 0;
  ml = s1 ? max(ml, len) : ml;
}
return ml;
```

Figure 31. *Longest* $1(0^*)2$.

```
int i;
int cl = 0, ml = 0, ml_tmp = 0;
for(i = 1; i < n; i++){
  cl = A[i] == 0 ? cl + 1 : 0;
  ml_tmp = max(ml_tmp, cl);
  if(ml_tmp % 2 == 0)
    ml = ml_tmp;
}
return ml;
```

Figure 32. *Longest even* 0^*.

```
int i;
bool s1 = false, s2 = false;
int cl = 0, ml = 0;
for(i = 0; i < n; i++){
  s1 = s2 && A[i] == 1;
  cl = s1 ? cl + 1 : (s2 ? cl : 0);
  ml = cl % 2 == 1 ? max(ml, cl) : ml;
  s2 = x == 0;
}
return ml;
```

Figure 33. *Longest odd* $(10)^*$.