Superfusion: Eliminating Intermediate Data Structures via Inductive Synthesis

RUYI JI, Peking University, China
YUWEI ZHAO, Peking University, China
NADIA POLIKARPOVA, University of California, San Diego, USA
YINGFEI XIONG*, Peking University, China
ZHENJIANG HU, Peking University, China

Intermediate data structures are a common cause of inefficiency in functional programming. *Fusion* attempts to eliminate intermediate data structures by combining adjacent data traversals into one; existing fusion techniques, however, are based on predefined rewrite rules and hence are limited in expressiveness.

In this work we explore a different approach to eliminating intermediate data structures, based on inductive program synthesis. We dub this approach *superfusion* (by analogy with *superoptimization*, which uses inductive synthesis for program optimization). Starting from a reference program annotated with data structures to be eliminated, superfusion first generates a *sketch* where program fragments operating on those data structures are replaced with holes; it then fills the holes with constant-time expressions such that the resulting program is equivalent to the reference. The main technical challenge here is scalability because optimized programs are often complex, making the search space intractably large for naive enumeration. To address this challenge, our key insight is to first synthesize a *ghost function* that describes the relationship between the original intermediate data structure and its compressed version; this function, although not used in the final program, serves to decompose the joint sketch filling problem into independent simpler problems for each hole.

We implement superfusion in a tool called SuFu and evaluate it on a dataset of 290 tasks collected from prior work on deductive fusion and program restructuring. The results show that SuFu solves 264 out of 290 tasks, exceeding the capabilities of rewriting-based fusion systems and achieving comparable performance with specialized approaches to program restructuring on their respective domains.

CCS Concepts: • **Software and its engineering** → **Programming by example**; **Automatic programming**; • **Theory of computation** → *Design and analysis of algorithms.*

Additional Key Words and Phrases: Inductive Program Synthesis, Program Optimization, Fusion

ACM Reference Format:
Ruyi Ji, Yuwei Zhao, Nadia Polikarpova, Yingfei Xiong, and Zhenjiang Hu. 2024. Superfusion: Eliminating Intermediate Data Structures via Inductive Synthesis. *Proc. ACM Program. Lang.* 8, PLDI, Article 185 (June 2024), 26 pages. https://doi.org/10.1145/3656415

*Corresponding author

Authors' addresses: Ruyi Ji, Key Laboratory of High Confidence Software Technologies, Ministry of Education; School of Computer Science, Peking University, Beijing, China, jiruyi910387714@pku.edu.cn; Yuwei Zhao, Key Laboratory of High Confidence Software Technologies, Ministry of Education; School of Computer Science, Peking University, Beijing, China, zhaoyuwei@stu.pku.edu.cn; Nadia Polikarpova, University of California, San Diego, USA, npolikarpova@ucsd.edu; Yingfei Xiong, Key Laboratory of High Confidence Software Technologies, Ministry of Education; School of Computer Science, Peking University, Beijing, China, xiongyf@pku.edu.cn; Zhenjiang Hu, Key Laboratory of High Confidence Software Technologies, Ministry of Education; School of Computer Science, Peking University, Beijing, China, huzj@pku.edu.cn.

ACM 2475-1421/2024/6-ART185
https://doi.org/10.1145/3656415

1 INTRODUCTION

Simplicity and efficiency are often at odds in programming. This is especially true in functional languages, where the idiomatic programming style is to compose library functions that operate on lists and other data structures. Programs written in this compositional style, however, are often inefficient because they have to allocate and traverse intermediate data structures.

Consider a function `mts` that returns the maximum tail sum of a (possibly negative) integer list, for example, `mts` $[1, -2, 3, -1, 2] = 4$, the sum of the last three elements $[3, -1, 2]$. This function can be implemented by composing list functions, like so:

```
mts xs = maximum (map sum (tails xs))
```

where `tails` returns a (nested) list of all tails of the input, `map` applies the function `sum` to each tail and obtains the list of all tail sums, and `maximum` returns the maximum among these sums.

This program is short and idiomatic. All four list functions it uses are commonly available in standard libraries, such as `Data.List` in Haskell. However, this program is also inefficient due to the large intermediate data structure constructed by `tails`, the list of all tails: the size of this data structure is quadratic in the size of the input list, causing `mts` to take quadratic time.

The inefficiency of compositional programs can often be addressed by eliminating intermediate data structures, replacing them with *scalar attributes* that are sufficient to compute the final result. For example, in the `mts` program, we can replace the list of all tails with a pair of attributes: (1) the maximum tail sum, and (2) the sum of the whole list. The intuition is that the `mts` of a non-empty list is either the `mts` of its tail, or the sum of the tail plus the head element. Using this observation, we can rewrite `mts` into an efficient program `mts'`, shown on the right, which only takes linear time. This program, however, is harder to write and understand than the original one. Hence we would like to write programs in the style of `mts`, and then transform them into efficient programs like `mts'` automatically.

```
mts' xs = (tails' xs).1
where
tails' Nil = (0, 0)
tails' Cons(h, t) =
  let (tmts, tsum) = tails' t in
  (max tmts (tsum + h), tsum + h)
```

Deductive Fusion. Automatic elimination of intermediate data structures is a well-studied problem in functional programming, also known as *fusion* or *deforestation* [Chin 1992; Coutts et al. 2007; Fokkinga 1992; Gill et al. 1993; Hamilton 2001; Hu et al. 1996; Meijer et al. 1991; Takano and Meijer 1995; Wadler 1988]. Existing approaches to fusion are *deductive*, *i.e.* based on a predefined set of rewrite rules that transform the reference program into an optimized one. Deductive fusion is fast and its results are correct by construction, but its main downside is limited expressiveness: it only applies to programs that match the rewrite rules. The state-of-the-art deductive approach [Hinze et al. 2010] can only handle around 50% of our benchmark suite, which we collected from the literature, and to our knowledge, no existing automatic fusion system can handle the `mts` example.

Superfusion. When faced with a similar expressiveness limitation of traditional compiler optimizations, Massalin [Massalin 1987] proposed *superoptimization*, an approach that abandoned deductive rewrite rules in favor of inductive synthesis, *i.e.* constructing an optimized program from scratch, by searching the space of all programs, until one is found that matches the input-output behavior of the reference implementation. In this paper, we take a similar approach to eliminating intermediate data structures, which we refer to accordingly, as *superfusion*.

The input to superfusion is a reference program annotated with data structures to be eliminated; *e.g.* the nested list returned by `tails` in the `mts` example. The first step is to turn the reference program into a *sketch* [Solar-Lezama et al. 2006], where any program fragment that consumes or produces the undesirable intermediate data structure is replaced with a *hole*. The second step is to

solve the sketch, filling the holes with new expressions that operate only on scalar attributes, while ensuring that the input-output behavior of the whole program is unchanged.[1]

Challenge 1: Scalability. The main technical challenge of superfusion is the scale of the search space. The target program of superfusion is often large because optimized programs are typically much more complex than idiomatic programs. Moreover, even when the solution to each single hole has a manageable size, there is still an issue that all holes have to be solved jointly, since our specification—equivalence to the reference program—is *global*. This scalability challenge makes a direct application of existing inductive synthesizers infeasible.

To address this challenge, our **first insight** is to synthesize a "ghost" *compression function*, denoted as *?compress*, which maps the intermediate data structure to be eliminated to its scalar attributes required in the optimized program. For example, the compression function for `mts` takes a list of tails and returns the maximum tail sum and the sum of the first tail (*i.e.* the full list):

```
?compress ts = (maximum (map sum ts), sum (head ts))
```

This is a "ghost" function, since it does not appear in the final solution `mts'`; its sole purpose is to decompose the global specification into *local* input-output specifications for each sketch hole, which can then be solved independently using existing *programming-by-example* (PBE) solvers [Alur et al. 2017b; Ji et al. 2021]. For example, with the definition of *?compress* above, it is straightforward to get input-output examples for the base case of `tails'` (the optimized version of `tails`): since `tails` [] = [[]] and *?compress* [[]] = $(0, 0)$, so `tails'` [] should return $(0, 0)$.

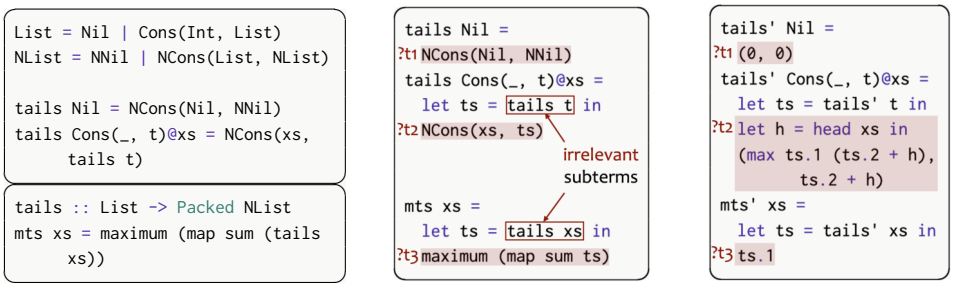
Challenge 2: Synthesizing Compression Functions. At this point, the reader might be wondering why synthesizing *?compress* is any easier than synthesizing the optimized program directly. After all, the only specification we have for *?compress* is that all sketch holes can be filled correctly while only operating on the scalar attributes. This appears to be a chicken-and-egg problem: we need the definition of *?compress* to efficiently fill the holes, but to decide if we got the right *?compress*, we need to know whether the holes can be filled! Our **second insight** is that, as long as the program space of sketch holes is rich enough, this apparent circular dependency can be broken using second-order *quantifier elimination*, resulting in an independent synthesis task for *?compress*. In comparison, this task is simpler than the original superfusion task because *?compress* does not need to be efficient, and hence is smaller (and easier to synthesize) than the sketch holes.

Evaluation. We implement superfusion in a tool called SuFu and evaluate it on a suite of 290 benchmarks collected from prior work. Our first source of benchmarks are fusion tasks from the deductive fusion literature [Bird 1989; Bird and de Moor 1997; Gill et al. 1993; Hu et al. 1997; Wadler 1988]. For our second source of benchmarks, we turns to prior work on *program restructuring* [Acar et al. 2005; Farzan et al. 2022; Farzan and Nicolet 2017, 2021a,b; Ji et al. 2024b; Morita et al. 2007; Pu et al. 2011], where the problem is to transform a reference program into a specific target form, such as "divide-and-conquer" or "single pass". We show that for several specific target forms studied in the literature, program restructuring can be reduced to fusion.

Our evaluation results show that SuFu can solve 264 out of 290 problems, at least half of which are beyond the reach of deductive fusion techniques. Moreover, while being general, SuFu is also competitive with two specialized synthesizers for program restructuring [Farzan et al. 2022; Ji et al. 2024b] on their respective domains, in terms of the number of solved problems (although it is somewhat slower in terms of synthesis times).

Contributions. To sum up, this paper makes the following main contributions.

[1]Like many inductive synthesizers [Solar-Lezama et al. 2006; Torlak and Bodík 2013] our technique only performs bounded verification, *i.e.* it only checks that the two programs are equivalent on a finite set of inputs; an external unbounded verifier can be integrated into superfusion if one is available.



(a) The input to SuFu. (b) Sketch generation. (c) Sketch solving.

Fig. 1. The workflow of SuFu on the `mts` example. (a) The input to SuFu: a reference implementation of `mts` with the output type of `tails` annotated for elimination. (b) Sketch generation: subterms highlighted in red produce or consume `NList` and hence are replaced with holes. (c) Sketch solving: the optimized program synthesized by SuFu where terms filled into sketch holes are highlighted.

- *A sketch generation method* (Sec. 3) that takes a reference program as the input and infers minimal sketch holes for eliminating intermediate data structures, guided by the type.
- *A sketch solving method* (Sec. 4) that decomposes the global sketch problem into local tasks for each sketch hole by synthesizing a ghost function and performing quantifier elimination.
- *An extensive evaluation* of SuFu (Sec. 7) on 290 benchmarks collected from the literature, which demonstrates the effectiveness of SuFu in eliminating intermediate data structures.

2 OVERVIEW

This section gives an overview of our tool SuFu, using the `mts` example from the introduction; the workflow of SuFu on this example is shown in Fig. 1. Recall that the idiomatic implementation of `mts` is inefficient because of the intermediate data structure produced by `tails`. To get a more efficient program, the user annotates the output type of `tails` with `Packed` to specify that it should be eliminated (Fig. 1a). SuFu then replaces the annotated data structure with scalar attributes and rewrites all related program fragments, generating a more efficient implementation (Fig. 1c).

2.1 Superfusion as Program Sketching

The first step in SuFu's workflow is to turn the annotated reference program into a *sketch* [Solar-Lezama et al. 2006], as shown in Fig. 1b. To this end, SuFu uses a type-directed approach, which we detail in Sec. 3, to identify the subterms that produce or consume data structures annotated with `Packed`—in our case, the `NList` generated by `tails`. There are three such terms in `mts`, labeled $?t_1$, $?t_2$, and $?t_3$ in Fig. 1b: $?t_1$ and $?t_2$ produce an `NList`, while $?t_2$ and $?t_3$ consume an `NList`. Each of these three terms is replaced with a *sketch hole*, which the synthesizer will need to fill with a new term, only operating on scalar attributes of the original `NList`.

To make the synthesizer's job easier, SuFu attempts to reuse as much of the original program as possible, moving subterms that do not directly operate on `Packed` data structures out of the holes, into `let`-bindings. For example, the two invocations of `tails` in Fig. 1b are moved out, because they do not contain any `NList`-specific operations.

Once the sketch has been generated, SuFu's task is to solve it, *i.e.* to fill the holes so that the resulting program is both *correct* and *efficient*:

- *Correctness* requires that the final program has the same input-output behavior as the reference. SuFu is based on the CEGIS framework [Solar-Lezama et al. 2006] and assumes

Table 1. Local examples collected by executing `mts` [2]. From left to right: hole id and the variables it has in scope; the original term for this hole; IO behavior of the original term (with intermediate data structures marked in blue); symbolic local examples obtained by "compressing" intermediate data structures with an unknown program *?compress*; concrete local examples obtained using the *?compress* in Eq. 1.

Hole	Original Term	Original IO Behavior	Local Examples	
			Symbolic Form	Concrete From
$?t_1$ ()	`NCons(Nil, NNil)`	$?t_1$ () = [[]]	$?t_1$ () = *?compress* [[]]	$?t_1$ () = (0, 0)
$?t_2$ (*xs*, *ts*)	`NCons(xs, ts)`	$?t_2$ ([2],[[]]) =[[2], []]	$?t_2$ ([2],*?compress* [[]]) =*?compress* [[2], []]	$?t_2$ ([2], (0, 0)) = (2, 2)
$?t_3$ *ts*	`maximum (map sum ts)`	$?t_3$ [[2], []] = 2	$?t_3$ (*?compress* [[2], []]) = 2	$?t_3$ (2, 2) = 2

the existence of an external verifier capable of generating counter-examples for incorrect programs; hence the synthesizer only needs to ensure correctness on a finite set of inputs.

- *Efficiency.* Fusion is only helpful if the resulting program is more efficient than the reference, but without any restrictions on the program space for the holes, this is not guaranteed: for example, the solution for $?t_3$ could *ignore* the new optimized `tails` and simply recreate the original implementation from scratch. To prevent this, we restrict the program space of sketch holes to include only recursion-free programs that run in $O(1)$ time. With this restriction, we can prove an efficiency guarantee on the resulting program (Thm. 4.11).

2.2 Sketch Solving with Compression Functions

Superfusion sketches are challenging to solve because optimized programs are often much more complex than idiomatic programs, so the expressions that need to be synthesized for each hole are relatively large. In our dataset, the average size of these expressions is 45.5 AST nodes, with a maximum of 559. This scale exceeds the capabilities of general-purpose sketch solvers [Solar-Lezama et al. 2006; Torlak and Bodík 2013], especially given that a superfusion sketch typically contains multiple holes, which must jointly satisfy the global IO specification. In our experiments, a state-of-the-art sketch solver [Lu and Bodík 2023] can only solve ~30% of tasks in our dataset.

To overcome this challenge, SuFu *decomposes* the global IO examples for the whole sketch into *local* IO examples for each hole. With these local examples, SuFu uses off-the-shelf PBE solvers for recursion-free programs [Alur et al. 2017b; Ji et al. 2021] to solve each hole independently.

To generate the local examples, we leverage the reference program. Specifically, we start by observing the IO behavior of the original terms that were replaced by holes. For example, Tab. 1 illustrates the behavior of the three terms in the original `mts` program, corresponding to sketch holes $?t_1$, $?t_2$, and $?t_3$, when executed on the input list [2]. Of course, these IO behaviors cannot directly be used as the specification for the holes because they involve intermediate data structures to be eliminated—the nested lists, shown in the table in blue.

Our first **key insight** is to bridge the gap between the behavior of the original and the target programs by introducing an unknown *compression function*, *?compress*, which maps the undesired intermediate data structure to scalar attributes. Using this function, we can express local examples for each hole in a *symbolic* form, by simply compressing the intermediate data structure in each original behavior, as shown in the fourth column of Tab. 1.

Assume for a second we had an oracle for *?compress*, which knew to compress the list of tails intro two scalar attributes, the maximum tail sum and the sum of list elements:

$$?compress\ ts \coloneqq \Big(maximum\ (map\ sum\ ts), sum\ (head\ ts)\Big) \tag{1}$$

This definition can now be substituted into the symbolic local examples to obtain *concrete* local examples for each sketch hole, as shown in the last column of Tab. 1. With enough local examples (collected by executing `mts` on enough inputs), an off-the-shelf PBE solver can efficiently synthesize the correct solution to each hole, shown in Fig. 1c.

The remaining challenge is to synthesize a suitable compression function, *i.e.* to guess which scalar attributes are sufficient to implement the sketch holes in $O(1)$ time; the rest of this section is devoted to this task.

2.3 Synthesizing the Compression Function

The only specification we have for *?compress* are symbolic local examples, such as those in Tab. 1. The issue with this specification, of course, is that it also refers to the unknown sketch holes, $?t_1$, $?t_2$, and $?t_3$. The naive way to approach this problem is to synthesize *?compress* and all sketch holes simultaneously, but that defeats the purpose of introducing *?compress* in the first place.

Our second **key insight** is that domain-specific properties of superfusion enable an efficient synthesis algorithm that combines *enumeration* and *(quantifier) elimination*. Specifically, we observe that the unknown programs (*?compress* and sketch holes) can be further decomposed into components that can be classified into two categories:

(1) components with a small implementation, which can be efficiently enumerated;
(2) components that can be quantified over as uninterpreted functions and eliminated.

This observation leads to the following synthesis algorithm:

- The top-level algorithm *iteratively refines* *?compress*, adding scalar attributes as required by local examples for a single hole.
- In each iteration, SuFu uses a second-order quantifier elimination method inspired by *Ackermann's reduction* [Lewis 1978] to obtain a simpler specification involving only the new scalar attributes plus at most a small component of a sketch hole, allowing efficient enumeration.

Next, we discuss the iterative refinement and the quantifier elimination method in more detail.

2.3.1 *Iterative Refinement of the Compression Function.*

Consider again the compression function in Eq. 1, which computes two scalar attributes from a list of tails. Instead of having to guess both attributes at once (by enumerating a tuple of terms), SuFu iteratively refines the definition of this function, starting from an empty tuple and adding an attribute whenever the specification for one of the sketch holes is found unrealizable.

Let us walk through this iterative refinement for `mts`; in the following, we assume that SuFu has extracted examples from multiple executions of `mts`, including `mts` $[2]$ (in Tab. 1) and `mts` $[2, -1]$.

Iteration 1. We start with the empty compression function, *i.e.* *?compress ts* $\coloneqq ()$. Substituting this definition into the local examples, we find that the constraints for hole $?t_3$ are *unrealizable*:

$$\exists ?t_3 \in \mathcal{L}^{\text{int}}_{O(1)}. \quad ?t_3\ () = 2\ \wedge\ ?t_3\ () = 1\ \wedge\ \ldots$$

Here $\mathcal{L}^{\text{int}}_{O(1)}$ is the space of all constant-time programs with integer output, and the two conjuncts originate from the two executions of `mts` mentioned above. Clearly such a $?t_3$ does not exist because it produces different outputs for the same input. Hence we need to add a new scalar attribute to *?compress* that provides enough information to differentiate between the two executions.

More formally, we need to find a function $?compress_1$ that satisfies the following specification:

$$\exists ?t_3 \in \mathcal{L}^{\text{int}}_{O(1)}. \quad ?t_3\ (?compress_1\ [[2], []]) = 2\ \wedge\ ?t_3\ (?compress_1\ [[2, -1], [-1], []]) = 1\ \wedge\ \ldots \quad (2)$$

Our quantifier elimination procedure, detailed below, can eliminate $?t_3$ from this specification, and, assuming enough local examples are available, will discover the correct definition for $?compress_1$:

$$?compress_1\ ts \coloneqq maximum\ (map\ sum\ ts)$$

Iteration 2. Since a new attribute has been added, we need to check whether it made any other sketch holes unrealizable. Indeed, by substituting our new definition $?compress \coloneqq ?compress_1$ into the local examples, we get the following for hole $?t_2$:

$$\exists ?t_2 \in \mathcal{L}^{\text{int}}_{O(1)}.\quad ?t_2\ ([2], 0) = 2\ \wedge\ ?t_2\ ([2,-1], 0) = 1\ \wedge\ \ldots$$

This specification is also unrealizable. Intuitively, this is because the `mts` of the tail (i.e., the second input of $?t_2$, which is 0 in both cases) clearly does not contain enough information to compute the `mts` of the whole list, which is 2 in the first case and 1 in the second case; and although $?t_2$ also has access to the input list *xs* through its first input, it cannot compute `mts` from *xs* in $O(1)$ time since *xs* can be arbitrarily long.

To fix this unrealizable hole, we need to add another attribute, $?compress_2$, to the compression function, satisfying the specification:

$$\begin{aligned} \exists ?t_2 \in \mathcal{L}^{\text{int}}_{O(1)}.\quad & ?t_2\ \big([2],\ \ (0, ?compress_2\ [[]])\ \big) = 2\ \wedge \\ & ?t_2\ \big([2,-1],\ (0, ?compress_2\ [[-1],[]])\ \big) = 1\ \wedge \ldots \end{aligned} \tag{3}$$

Once again, given enough examples, our quantifier elimination will discover that the additional attribute required here is the sum of the list elements:

$$?compress_2\ ts \coloneqq sum\ (head\ ts)$$

Iteration 3. Once we substitute $?compress\ ts \coloneqq (?compress_1\ ts, ?compress_2\ ts)$ into local examples, we find that all sketch holes are realizable, which concludes the synthesis of *?compress*.

2.3.2 Quantifier Elimination. We conclude this section by explaining how SuFu synthesizes each scalar attribute of *?compress* from the specifications Eq. 2 and Eq. 3.

Iteration 1. Let us consider the specification for $?compress_1$ in Eq. 2. This specification involves two unknown programs, $?compress_1$ and $?t_3$, and our goal is to avoid synthesizing both of them simultaneously. To this end, we observe that:

(1) to synthesize $?compress_1$, we do not need to know the implementation of $?t_3$, only that such a function *exists*.
(2) a function exists *if and only if* it maps every input to a unique output.

These observations suggest a simple second-order quantifier elimination procedure[2], which transforms Eq. 2 into the following equivalent form that involves only $?compress_1$:

$$\forall (ts, out), (ts', out') \in \mathbb{E}_3.\quad (?compress_1\ ts = ?compress_1\ ts') \rightarrow (out = out') \tag{4}$$

where $\mathbb{E}_3$ is the set of local examples collected for $?t_3$ from the executions of `mts` (such as the last row, the Example column of Tab. 1).

We can now enumerate candidate $?compress_1$ from smaller to larger, checking them against the specification Eq. 4. SuFu can find the desired program $?compress_1\ ts \coloneqq maximum\ (map\ sum\ ts)$ from 21 executions of `mts`. Tab. 2 illustrates three smaller but incorrect candidates for $?compress_1$, together with a pair of executions sufficient to reject them.

A careful reader might object that Eq. 4 is only equivalent to Eq. 2 when the space of $?t_3$ can implement all possible functions, whereas in our case $?t_3$ is restricted to the space of $O(1)$-time

[2]In fact, this is a special case of *Ackermann's reduction* [Lewis 1978], used to eliminate uninterpreted functions from a formula by replacing their invocations with fresh variables and adding constraints that equal inputs produce equal outputs.

Table 2. Three invalid candidates for $?compress_1$ and examples sufficient to reject them based on Eq. 4.

Candidate Program	Execution	Symbolic Local Example for $?t_3$	Conflict
length of the list of tails	`mts [2]`	$?t_3\ (?compress\ [[2],[]]) = 2$	$?t_3\ 2 = 2$
$?compress_1\ ts \coloneqq length\ ts$	`mts [1]`	$?t_3\ (?compress\ [[1],[]]) = 1$	$?t_3\ 2 = 1$
sum of elements	`mts [2]`	$?t_3\ (?compress\ [[2],[]]) = 2$	$?t_3\ 2 = 2$
$?compress_1\ ts \coloneqq sum\ (head\ ts)$	`mts [−1, 3]`	$?t_3\ (?compress\ [[-1,3],[3],[]]) = 3$	$?t_3\ 2 = 3$
maximum of elements	`mts [2]`	$?t_3\ (?compress\ [[2],[]]) = 2$	$?t_3\ 2 = 2$
$?compress_1\ ts \coloneqq maximum\ (head\ ts)$	`mts [2, −1]`	$?t_3\ (?compress\ [[2,-1],[-1],[]]) = 1$	$?t_3\ 2 = 1$

programs. In this case, however, it is reasonable to assume that the $O(1)$-time requirement imposes no extra restrictions on the space of $?t_3$, because both its inputs and outputs are scalar values, and most scalar calculations can be done in $O(1)$ time.

Iteration 2. The specification Eq. 3 for $?compress_2$ is a little more involved. The main difference is that we cannot eliminate $?t_2$ using the same reasoning as above, because $?t_2$ no longer operates only on scalar values: it takes the list xs as its first input, and many list-processing functions do not have any $O(1)$-time implementation.

Fortunately, this issue can be addressed by further decomposing $?t_2$. Specifically, since $?t_2$ is an $O(1)$-time program, it can only access a constant number of scalar values in the input. Therefore, without loss of generality, we can assume that $?t_2$ has the form:

$$?t_2\ (xs, ts) \coloneqq ?comb\ (?extract\ (xs, ts))$$

where $?extract$ extracts a tuple of scalar values from the input variables and $?comb$ combines them into the final result. With this decomposition in place, we can now use our previous technique to *eliminate* $?comb$ (which operates on scalars), and then *enumerate* $?extract$ jointly with $?compress_2$ (note that $?extract$ is small because it does not perform any actual computation), resulting in:

$$?extract\ (xs, ts) \coloneqq (ts.1, ts.2, head\ xs) \qquad ?compress_2\ ts \coloneqq sum\ (head\ ts)$$

3 SKETCH GENERATION

Given a reference program whose intermediate data structures have been annotated with `Packed`, SuFu's first step is to translate this program into a *sketch* by replacing some of its subterms with *holes*. Intuitively, sketch holes must satisfy the following requirements:

(1) **Correctness**: the holes must cover all subterms that directly produce or consume intermediate data structures because these data structures must be replaced with scalar attributes.
(2) **Feasibility**: the output of each hole must be scalar; otherwise, this hole's solution will need to reconstruct a non-scalar data structure from scalar attributes, which is generally impossible.
(3) **Minimality**: the subterms covered by holes should be as small as possible because the larger they are, the more SuFu needs to synthesize, and thus the harder the synthesis task will be.

To formalize these requirements, we design an *intermediate language*, dubbed λ_{sk}, which makes the scope of the holes and all uses of intermediate data structures explicit. Importantly, any well-typed λ_{sk} program corresponds to a sketch that is both correct and feasible. Sketch generation is then reduced to the problem of elaborating the reference program into a well-typed intermediate representation, while minimizing the scope of the holes.

3.1 Intermediate Language

The syntax of λ_{sk} is shown in Fig. 2. The language is a simply typed lambda calculus with inductive data types, which we augment with the following *annotations*:

$$
\begin{array}{lcl}
t \in \text{Term} & ::= & \text{constants} \mid x \mid \texttt{app}(t_1, t_2) \mid \lambda(x, T, t) \mid \texttt{fix}(t) \mid \texttt{let}(x, t_1, t_2) \mid \texttt{if}(t_1, t_2, t_3) \mid C(t) \\
& \mid & \texttt{tuple}(\overline{t_i}) \mid \texttt{proj}(t, i) \mid \texttt{match}(t, \overline{C_i(x_i).t_i}) \mid \texttt{rewrite}(t) \mid \texttt{label}(t) \mid \texttt{unlabel}(t) \\
T \in \text{Type} & ::= & B \mid T \to T \mid T_1 \times \cdots \times T_n \\
B \in \text{BaseType} & ::= & \texttt{Unit} \mid \texttt{Int} \mid \texttt{Bool} \mid B_1 \times \cdots \times B_n \mid \texttt{ind}(\overline{C_i}) \mid \texttt{Packed}(B) \\
S \in \text{ScalarType} & ::= & \texttt{Unit} \mid \texttt{Int} \mid \texttt{Bool} \mid S_1 \times \cdots \times S_n \mid \texttt{Packed}(B)
\end{array}
$$

Fig. 2. SuFu's intermediate language λ_{sk} presented via *abstract binding trees* [Harper 2016]. We use $\texttt{ind}(\cdot)$ to denote inductive data types, and C to denote data constructors. Annotations (highlighted in red) are not part of the surface language. SuFu automatically introduces term-level annotations (`rewrite`, `label`, and `unlabel`) given the type-level `Packed` annotations provided by the user.

$\boxed{\Gamma \vdash_s t : T \text{ for scope } s \in \{\text{in}, \text{out}\}}$

(T-Rewrite)
$$\frac{\Gamma \vdash_{\text{in}} t : S \qquad S \in \text{ScalarType}}{\Gamma \vdash_s \texttt{rewrite}(t) : S}$$

(T-Label)
$$\frac{\Gamma \vdash_{\text{in}} t : B \qquad B \in \text{BaseType}}{\Gamma \vdash_{\text{in}} \texttt{label}(t) : \texttt{Packed}(B)}$$

(T-Unlabel)
$$\frac{\Gamma \vdash_{\text{in}} t : \texttt{Packed}(T)}{\Gamma \vdash_{\text{in}} \texttt{unlabel}(t) : T}$$

Fig. 3. Typing rules for annotations in λ_{sk}. The scope s represents whether the current term is inside a `rewrite`.

- At the type level, we introduce `Packed` annotations, which are treated as a type constructor: `Packed` T is a new type, different from T. Besides, λ_{sk} treats `Packed` T as a scalar type because, intuitively, users annotate data structures as `Packed` to turn them into scalars.
- At the term level, we introduce three annotations—`label`, `unlabel`, and `rewrite`—which are added automatically during elaboration. `label` and `unlabel` are the constructor and destructor for `Packed` values; for example, `label` $[1, 2]$ is of type `Packed List` and `unlabel` (`label` $[1, 2]$) = $[1, 2]$. These constructs make the production and consumption of intermediate data structures explicit. Finally, the `rewrite` annotation marks the scope of a hole; `rewrite` t has the same type and behavior as t, but is treated as a hole to be replaced with a constant-time term.

The typing rules of λ_{sk} are standard apart from those for annotations, shown in Fig. 3. Note that the typing judgment $\Gamma \vdash_s t : T$ is parameterized by a *scope* $s \in \{\text{in}, \text{out}\}$, which keeps track of whether the current term t is inside a `rewrite`. The rules T-Label and T-Unlabel only allow producing and consuming intermediate data structures inside a `rewrite`, thereby enforcing the *correctness* requirement above. The rule T-Rewrite restricts the argument to `rewrite` to have a scalar type, thereby enforcing the *feasibility* requirement.

Example 3.1. Consider again the `mts` program from Fig. 1a. Here the user has annotated the output of `tails` with `Packed`, thereby making the program *ill-typed* in λ_{sk}: specifically, the body of `tails` is ill-typed because it produces an `NList` where a `Packed NList` is expected, and conversely, the body of `mts` is ill-typed because `tails xs` returns a `Packed NList` where an `NList` is expected by `map sum`. The task of the elaboration process is to insert `label`, `unlabel`, and `rewrite` annotations to make the program well-typed in λ_{sk}. Consider four possible elaborations of the body of `mts`:

```
maximum (map sum (unlabel (tails xs)))                              (A)
maximum (rewrite (map sum (unlabel (tails xs))))                    (B)
rewrite (maximum (map sum (unlabel (tails xs))))                    (C)
let ts = tails xs in rewrite (maximum (map sum (unlabel ts)))       (D)
```

Term A inserts an `unlabel` to provide the correct argument type to `map sum`; it is still ill-typed, however, because `unlabel` is not inside a `rewrite`. Intuitively, this term violates the *correctness* requirement: the invocation of `tails` produces an intermediate data structure, but is not covered

```
tails :: List -> Packed NList
tails Nil =
  let ts = NCons(Nil, NNil) in
  rewrite (label ts)
tails Cons(_, t)@xs =
  let ts = tails t in
  rewrite (label NCons(xs, unlabel ts))
mts_label xs =
  let ts = tails xs in
  rewrite (maximum (map sum (unlabel ts)))
```

Fig. 4. The sketch for `mts` from Fig. 1a.

Algorithm 1: Sketch solving.

Input: A sketch p and inputs I.
Output: A sketch solution $(\overline{T_i}, \overline{t_h})$.

$\mathbb{E}_{sym} \leftarrow \bigcup_{in \in I}$ `CollectExamples`(p, in);
compress $\leftarrow$ `CompressSynthesis`$(\mathbb{E}_{sym})$;
$\mathbb{E}_{io} \leftarrow$ `Subst`$(\mathbb{E}_{sym}, compress)$
foreach *sketch hole* h **do**
  $t_i \leftarrow$ `PBESolver`$(\mathcal{L}^{\text{scalar}}_{O(1)}, \mathbb{E}_{io}[h])$;
end
return ([output type of *compress*], $\overline{t_i}$);

by a hole. Term B does cover the `unlabel` with a `rewrite`, but it is nevertheless ill-typed because the argument type of its `rewrite` is a `List` instead of a scalar type. Intuitively, this term violates the *feasibility* requirement: if we tried to synthesize a solution for this hole, we would need to reconstruct the full list of tail sums from only scalar attributes of the input list, which is impossible. Finally, terms C and D are both well-typed elaborations of `mts`; the difference is that term D moves the invocation of `tails` into a `let` binding and out of the `rewrite`, thereby reducing the scope of the hole and allowing the synthesizer to reuse the recursion structure of the original program.

3.2 Generating Minimal Annotations

With the intermediate language in place, we can now formalize the task of sketch generation:

Definition 3.2 (Sketch Generation). Given a reference program p in λ_{sk} (potentially ill-typed due to `Packed` annotations), find a sketch, *i.e.* a program p' in λ_{sk}, such that: (1) p' can be obtained from p by adding `label`, `unlabel`, and `rewrite` annotations, or extracting sub-terms into `let`-bindings; (2) p' is well-typed; (3) the total size of arguments to `rewrite` in p' is minimized.

Note that the IO behavior of the sketch is always the same as that of the reference program, because all the transformations performed during sketch generation are semantics-preserving.

SuFu reduces the sketch generation problem to a MaxSAT instance and solves it using an off-the-shelf constraint solver (*Z3* [de Moura and Bjørner 2008] in our implementation). The encoding is straightforward: SuFu encodes the search space (condition 1 above) as a symbolic program, where the choice to insert an annotation or `let`-bindings for each subterm is controlled by a Boolean variable; it encodes the typing (condition 2) as a hard constraint by symbolically executing the type checker on the symbolic program; finally, it encodes the minimality requirement (condition 3) as a soft constraint by symbolically executing the objective function.

Example 3.3. Among the four different elaborations of `mts` in Example 3.1, all of them satisfy condition (1) from definition Def. 3.2, but only terms C and D are well-typed and only term D is also minimal. The resulting sketch—*i.e.* the well-typed and minimal elaboration—for the full `mts` example from Fig. 1a is shown in Fig. 4.

4 SKETCH SOLVING

In this section, we introduce the sketch solving approach in SuFu. We start with the problem definition (Sec. 4.1) and a high-level overview of the approach (Sec. 4.2), and then delve into the details in the following subsections.

4.1 Synthesis Problem

A *solution* for a sketch in λ_{sk} consists of (1) a list of *scalar surface-level types* to replace `Packed` T types, and (2) a list of *constant-time surface-level terms* to replace holes, *i.e.* `rewrite` t terms. The restriction to surface level simply means that the solution cannot use any annotations, and the restriction to scalar types and constant-time terms ensures that the solution is efficient.

Definition 4.1 (Sketch Solution). A *solution* for a sketch p in λ_{sk} is a pair $(\overline{T_i}, \overline{t_h})$ of a list of types $\overline{T_i}$ and a list of terms $\overline{t_h}$ satisfying the following conditions.

- $\overline{T_i}$ contains only scalar types in the surface language.
- $\overline{t_h}$ contains only constant-time terms in the surface language, or formally, terms that can be evaluated within a fixed number of steps under any well-typed context.
- lengths of $\overline{T_i}$ and $\overline{t_h}$ are respectively equal to the number of `Packed` and `rewrite` annotations.

Given a solution $(\overline{T_i}, \overline{t_h})$, we obtain a *synthesized program* from the sketch p by replacing each `Packed` T with the corresponding type in $\overline{T_i}$ and each `rewrite` t with the corresponding term in $\overline{t_h}$.

Example 4.2. For the sketch `mts_label` in Fig. 4, the desired solution is to take the singleton list [`Int` × `Int`] as $\overline{T_i}$ and take the three highlighted terms in Fig. 1c as $\overline{t_h}$.

Our synthesis problem then is to find a sketch solution such that the synthesized program has the same IO behavior as the sketch (and hence, as the reference program it was generated from).

Definition 4.3 (Sketching Problem). Given a sketch p in language λ_{sk} and a finite set I of inputs, the sketching problem is to find a solution of p such that the synthesized program has the same output as p on every input in I.

For simplicity, we consider only a finite set of inputs in this definition. In our implementation, we incorporate the CEGIS framework [Solar-Lezama 2009] to reduce the general case of an infinite input space to a finite set of representative inputs.

In the remainder of this section, we focus our discussion on a special case where `Packed` is used only once (and hence, $|\overline{T_i}| = 1$). Our approach extends straightforwardly to the general case with multiple `Packed` annotations by synthesizing a separate compression function for each intermediate data structure. Details on this extension can be found in the appendix [Ji et al. 2024a].

4.2 Top-Level Algorithm

Algorithm 1 is SuFu's top-level sketch solving algorithm. Given a sketch p and a set of inputs I, SuFu evaluates the sketch on each input and collects *symbolic local examples* $\mathbb{E}_{sym}$ for the holes (Line 1). From the symbolic examples it synthesizes a compression function *compress*, which maps intermediate data structures to their scalar attributes (Line 2). It then substitutes *compress* into the symbolic examples to obtain *concrete local examples* $\mathbb{E}_{io}$ (Line 3). Finally, SuFu uses a PBE solver to synthesize a solution for each sketch h from its concrete examples, $\mathbb{E}_{io}[h]$ (Line 4).

The core of this algorithm is functions `CollectExamples` and `CompressSynthesis`. We shall introduce these two functions in order in the next two subsections (Sec. 4.3 and Sec. 4.4).

4.3 Example Collection

We use *big-step environment semantics* [Dikotter 1990] of λ_{sk} to formalize the collection of local examples. The evaluation in this semantics follows strictly along the syntax, hence local examples can be directly constructed from the results of evaluating `rewrite` terms.

Example 4.4. The following shows a part of the derivation for evaluating `mts_label`$[2, -1]$. The judgment has the form of $E \vdash t \Downarrow v$, where E is an environment assigning values to free variables, t is the term to be evaluated, and v is the evaluation result.

$$\dfrac{\dfrac{\ldots}{(\texttt{xs} \mapsto [2,-1]) \vdash \texttt{tails xs} \Downarrow \texttt{label}\ [[2,-1],[-1],[]]} \quad \dfrac{\ldots}{(\texttt{xs} \mapsto [2,-1], \texttt{ts} \mapsto \texttt{label}\ [[2,-1],[-1],[]]) \vdash \texttt{rewrite (maximum (map sum (unlabel ts)))} \Downarrow 1}}{(\texttt{xs} \mapsto [2,-1]) \vdash \texttt{let ts = tails xs in rewrite (maximum (map sum (unlabel ts)))} \Downarrow 1}$$

The right branch here evaluates the third `rewrite` term in `mts_label`, corresponding to hole $?t_3$. This term evaluates to 1 under the environment $(\texttt{xs} \mapsto [2,-1], \texttt{ts} \mapsto \texttt{label}\ [[2,-1],[-1],[]])$. As you can see, the values collected from a sketch evaluation can contain the `label` constructor (in this case, in the input, but generally also in the output); these `label` v values correspond to intermediate data structures that must be "compressed" into scalar attributes. Hence, we replace `label` with the unknown function *?compress* to obtain a *symbolic local example* for a hole; for instance:

$$\langle in, out \rangle = \left\langle \left(\texttt{xs} \mapsto [2,-1], \texttt{ts} \mapsto \textit{?compress}\ [[2,-1],[-1],[]]\right), 1 \right\rangle$$

is a symbolic local example for $?t_3$ that corresponds to the sketch evaluation above and is added to $\mathbb{E}_{sym}[?t_3]$ by CollectExamples.

To convert symbolic examples to concrete examples, Algorithm 1 uses $\texttt{Subst}(\mathbb{E}_{sym}, compress)$, which substitutes the concrete compression function *compress* for the variable *?compress*, and then β-reduces the resulting term. For example, invoking `Subst` on the symbolic example above with $compress = \lambda x.\texttt{sum (head x)}$ yields the concrete example $\langle(\texttt{xs} \mapsto [2,-1], \texttt{ts} \mapsto 1), 1\rangle$.

4.4 Synthesizing the Compression Function

Given the symbolic local examples $\mathbb{E}_{sym}$, our task is to pick an implementation for *?compress* such that each sketch hole is realizable in constant time, or more formally:

$$\bigwedge_{h=1}^{n} \rho(h) \quad \text{where} \quad \rho(h) = \exists ?t_h \in \mathcal{L}^{\text{scalar}}_{O(1)}, \forall \langle in, out \rangle \in \mathbb{E}_{sym}[h], ?t_h\ in = out \tag{5}$$

Here n is the number of sketch holes and $\mathcal{L}^{\text{scalar}}_{O(1)}$ is the space of constant-time scalar-typed terms in the surface language. We will refer to the constraint $\rho(h)$ as the *realizability constraint* for hole h.

Example 4.5. Assume that we have collected a series of local symbolic examples for the hole $?t_3$, including the one from Example 4.4. Then the realizability constraint $\rho(?t_3)$ for this hole is:

$$\exists ?t_3 \in \mathcal{L}^{\text{scalar}}_{O(1)}, \quad ?t_3\ \left(\texttt{xs} \mapsto [2,-1], \texttt{ts} \mapsto \textit{?compress}\ [[2,-1],[-1],[]]\right) = 1 \wedge \ldots \text{(other examples)}$$

If we use a trivial compression function that does not extract any attributes, *i.e.* $?compress \mapsto \lambda x.()$, this constraint will be unrealizable since we cannot compute the `mts` in constant time just from the input list. However, if we add the attribute of the maximum tail sum, *i.e.* $?compress \mapsto \lambda x.(\texttt{maximum (map sum x)})$, this constraint becomes realizable with $?t_3$ simply returning `ts`.

SuFu solves realizability constraints by *iteratively refining* *?compress* with new attributes, as shown in Algorithm 2. This algorithm maintains the current tuple of attributes A. Every attribute $\alpha \in A$ is a term with a free variable x that denotes the intermediate data structure, so that $\lambda x.A$ is a valid compression function; for example, we might have:

$$A = (\texttt{maximum (map sum x)}, \texttt{sum (head x)})$$

to denote that the current compression function computes two attributes from the list of tails x—maximum tail sum and sum of list elements, referred to hereafter as *mts* and *sum*, respectively.

The tuple A starts out empty and is iteratively extended with new attributes found by `Refine`. In each iteration, SuFu first instantiates the symbolic examples $\mathbb{E}_{sym}$ with the current compression function $\lambda x.A$ to obtain the concrete local examples $\mathbb{E}$ (Line 3). The function `Refine` (Lines 8-12)

Algorithm 2: Function `CompressSynthesis`.

Input: Symbolic local examples $\mathbb{E}_{sym}$.
Output: A compression function.

```
1  A ← ();
2  while true do
3  |   𝔼 ← Subst(𝔼_sym, λx.A);
4  |   A* ← Refine(𝔼);
5  |   if A* is empty then return λx.A;
6  |   A ← A ∪ A*;
7  end
8  Function Refine(𝔼):
9  |   foreach sketch hole h do
10 |   |   A_h ← SolveSingleHole(𝔼[h]);
11 |   end
12 |   return ∪_h A_h;
```

Table 3. Workflow of Algorithm 2 on `mts`. Row *sym* shows two symbolic local examples collected from for holes $?t_2$ and $?t_3$ (with intermediate data structures in blue). The remaining three rows show concrete versions of those examples in the first three iterations, with the intermediate data structures replaced by the scalar attributes added so far. Grayed out examples (and attributes in the output) are ignored because they are already known to be satisfied by the current compression function.

Iter.	Example for $?t_2$		Example for $?t_3$	
	In	Out	In	Out
sym	$\texttt{xs} \mapsto [2,-1], \texttt{ts} \mapsto [[-1],[]]$	$[[2,-1],[-1],[]]$	$\texttt{xs} \mapsto [2,-1], \texttt{ts} \mapsto [[2,-1],[-1][]]$	1
1	$\texttt{xs} \mapsto [2,-1], \texttt{ts} \mapsto ()$	$()$	$\texttt{xs} \mapsto [2,-1], \texttt{ts} \mapsto ()$	1
2	$\texttt{xs} \mapsto [2,-1], \texttt{ts} \mapsto (0)$	(1)	$\texttt{xs} \mapsto [2,-1], \texttt{ts} \mapsto (0)$	1
3	$\texttt{xs} \mapsto [2,-1], \texttt{ts} \mapsto (0,-1)$	$(1,1)$	$\texttt{xs} \mapsto [2,-1], \texttt{ts} \mapsto (0,-1)$	1

then iterates through each hole h, synthesizing additional attributes A_h required to make this hole realizable under the current concrete examples $\mathbb{E}[h]$; the heavy lifting here is done by the function `SolveSingleHole`, which will be introduced in Sec. 4.5. In Line 12, `Refine` returns the union of all new attributes modulo observational equivalence (two attributes are observationally equivalent if their value is the same for all intermediate data structures in $\mathbb{E}_{sym}$).

If `Refine` did not synthesize any new attributes, then all the holes are already realizable with the current compression function, and the algorithm terminates returning $\lambda x.A$ (Line 5). Otherwise, SuFu adds the new attributes to A and continues.[3]

Example 4.6. Let us walk through Algorithm 2 for the `mts` task. Here we assume that enough executions of `mts` are available to avoid spurious solutions and focus on two local examples for holes $?t_2$ and $?t_3$ in Tab. 3; both of these examples are collected from the execution `mts_label` [2, -1]. The row *sym* shows the symbolic local examples (with intermediate data structures in blue), while the remaining rows show the concrete version in each iteration of Algorithm 2 (Lines 2-7), where intermediate data structures are replaced according to the current compression function.

Iteration 1. We start with $A = ()$; the corresponding concrete local examples are shown in Row 1 of Tab. 3. At this point, hole $?t_3$, whose output is a *surface-language scalar*, is unrealizable for the reasons explained in Example 4.5. On the other hand, hole $?t_2$, whose output is an *intermediate data structure*, is trivially realizable, since it simply returns unit; hence, `Refine` can safely ignore $?t_2$ in this iteration (denoted by its examples being grayed out in the table). Based on the examples for

[3]Readers familiar with Constrained Horn Clauses (CHC) might recognize the similarity between the iterative refinement of the compression function and fixpoint CHC solvers [Bjørner et al. 2013; Rondon et al. 2008], which initialize unknown predicates with a trivial solution and then iteratively weaken (or strengthen) them to resolve invalid clauses.

$?t_3$, `Refine` will discover that the attribute *mts* is necessary to make this hole realizable, resulting in $A = (mts)$ after the first iteration.

Iteration 2. Since the first iteration synthesized a new attribute *mts*, we need to call `Refine` again with the new concrete examples (shown in Row 2 of Tab. 3), to check whether the new attribute caused any holes to become unrealizable. This is indeed the case in for hole $?t_2$: this hole now must compute the *mts* of the full list from only the *mts* of the tail (and the whole input list), which is impossible in constant time. Note that the hole $?t_3$ is ignored in this iteration because its output did not change, so we already know it is realizable. In this round `Refine` will discover a new attribute *sum*—the sum of list elements—for hole $?t_2$, thus updating tuple A to $A = (mts, sum)$.

Iteration 3. Since a new attribute *sum* was added, SuFu invokes `Refine` once again on the new concrete examples (Row 3 of Tab. 3). This invocation can still ignore the hole $?t_3$, and additionally can ignore the old attribute *mts* in the output of $?t_2$, because previous iterations already ensure that it can be calculated in constant time. Hence, `Refine` focuses on the *new output* of $?t_2$, checking if it is realizable; since this is the case, no new attributes are added, and the algorithm terminates.

4.5 Synthesizing Attributes for a Single Sketch Hole

Algorithm 2 invokes the function `SolveSingleHole` to discover missing attributes for a hole h based on the concrete examples $\mathbb{E}[h]$ for that hole. In other words, this function needs to synthesize an additional compression function $?compress'$ such that the examples $\mathbb{E}[h]$ would become realizable after extending their inputs with the new attributes specified by $?compress'$:

$$\exists ?t_h \in \mathcal{L}^{\text{scalar}}_{O(1)}, \forall \langle in, out\rangle \in \mathbb{E}[h], ?t_h \left(in, \overline{?compress'\ in_i^{orig}}\right) = out \tag{6}$$

Here we write in_i^{orig} denotes the ith intermediate data structure in the original input, which has been compressed into scalar attributes in the current input *in*.

Example 4.7. When `SolveSingleHole` is invoked on $?t_2$ in the second iteration of Algorithm 2, the specification of $?compress'$ is as follows (with the example in Row 2 of Tab. 3 shown explicitly):

$$\exists ?t_2 \in \mathcal{L}^{\text{scalar}}_{O(1)}, ?t_2 \left((\texttt{xs} \mapsto [2,-1], \texttt{ts} \mapsto 0), ?compress'\ [[2,-1],[-1],[]]\right) = (1) \wedge \ldots \text{(other examples)}$$

It is challenging to solve the specification Eq. 6 because it involves the sketch hole $?t_h$, which is too complex to be synthesized together with $?compress'$. To overcome this challenge, we transform this specification in two steps to eliminate (most of) the sketch hole.

First, since $?t_h$ is limited to constant-time, it can only access a constant number of scalar values in the input. Hence, it can be decomposed as $?t_h\ in := ?comb\ (?extract\ in)$, where $?extract$ extracts a tuple of scalar values and $?comb$ accomplishes the calculation using the extracted values. With this decomposition, Eq. 6 can be rewritten as follows:

$$\exists ?comb, ?extract \in \mathcal{L}^{\text{scalar}}_{O(1)}, \forall \langle in, out\rangle \in \mathbb{E}, ?comb \left(?extract \left(in, \overline{?compress'\ in_i^{orig}}\right)\right) = out$$

Second, since most common scalar calculations can be accomplished efficiently, we assume that our program space $\mathcal{L}^{\text{scalar}}_{O(1)}$ is expressive enough to implement all possible scalar functions.

Assumption 4.8. *For any function f whose input and output are both scalar values, there always exists a constant-time term with the same semantics, i.e., $\exists t \in \mathcal{L}^{scalar}_{O(1)}, \forall in, f\ in = t\ in$.*

We then use a quantifier elimination procedure to remove $?comb$ from the specification. Specifically, by Asm. 4.8, we can regard $?comb$ as an uninterpreted function and thus apply Ackermann's reduction [Lewis 1978] to $?comb$. This reduction eliminates an uninterpreted function using its

congruence property (i.e., identical inputs imply identical outputs), and here, the reduction result is an equivalent specification without involving *?comb*, as shown below.

$$\exists ?extract \in \mathcal{L}^{\text{scalar}}_{O(1)}, \forall \langle in_1, out_1\rangle, \langle in_2, out_2\rangle \in \mathbb{E}, \\ ?extract\left(in_1, \overline{?compress'\ in^{orig}_{1,i}}\right) = ?extract\left(in_2, \overline{?compress'\ in^{orig}_{2,i}}\right) \rightarrow out_1 = out_2 \tag{7}$$

SuFu treats Eq. 7 as a joint specification for *?compress′* and *?extract* and synthesizes them simultaneously by enumeration. Given a program space for *?compress′*, it enumerates all pairs of candidate programs for *?compress′* and *?extract* in the increasing order of the total size, until a pair satisfying Eq. 7 is found. This enumeration method, although straightforward, is effective in practice because in most cases, the target *?compress′* and *?extract* are both small: *?compress′* can be constructed compactly from library functions since its efficiency is not important, while *?extract* needs only to access scalar values in the input as opposed to performing complex scalar calculations.

4.6 Properties

Soundness. SuFu ensures that the synthesized program is observationally equivalent to the sketch, and hence to the reference program.

Theorem 4.9 (Soundness). *For any sketching problem with sketch p and input set I, the synthesized program has the same output as p on set I, if the underlying PBE solver for sketch holes is sound.*

Proof. Due to the space limit, we move the proofs to the appendix [Ji et al. 2024a]. □

Completeness. The completeness of SuFu depends on the program space of compression functions. Since local examples are collected from the evaluation of `rewrite` terms, there is always enough information in the inputs to calculate the scalar outputs and any required scalar attributes. Consequently, when the program space of *?compress* is expressive enough, SuFu can always find an appropriate *?compress*, and with it, a solution to the sketching problem.

Theorem 4.10 (Completeness). *For any sketching problem, SuFu can find a solution if Asm. 4.8 holds, the underlying PBE solver for sketch holes is complete, and the program space of ?compress can implement any function mapping from intermediate data structures to scalar values.*

Efficiency of synthesized programs. SuFu replaces zero or more subterms in the reference program with constant-time terms. Hence, SuFu ensures that its synthesized program cannot have a higher asymptotic complexity than the reference.

Theorem 4.11 (Efficiency). *Let $cost(p, in)$ be the size of the derivation tree of evaluating program p on input in. For any sketching problem, the following formula is always satisfied by the reference program p and the program p' synthesized by SuFu.*

$$\exists c > 0, \forall in, cost(p', in) \leq c \cdot cost(p, in)$$

In practice, the synthesized program is usually strictly more efficient than the reference because the `rewrite` terms typically involve non-constant-time operators on intermediate data structures.

5 APPLICATIONS TO PROGRAM RESTRUCTURING

In this section, we consider several lines of prior work on synthesizing efficient recursive programs, which we collectively refer to as *program restructuring*. Program restructuring aims to rewrite a reference program to follow a given efficient template, such as divide-and-conquer [Farzan and Nicolet 2021b; Ji et al. 2024b; Morita et al. 2007] or single-pass recursion [Farzan et al. 2022; Pu et al. 2011]. We show how this task can be reduced to a superfusion problem and solved by SuFu.

```
List = Elt(Int) | Cons(Int, List)
dac_mts Elt(e) = (max e 0, 0)
dac_mts Cons(_, _)@xs =
  let (ls, rs) = split xs in
  let (lmts, lsum) = dac_mts ls in
  let (rmts, rsum) = dac_mts rs in
  (max rmts (lmts + rsum), lsum + rsum)
```

(a) A D&C program for mts.

```
dac_id :: List -> Packed List
dac_id Elt(_)@xs = xs
dac_id Cons(_, _)@xs =
  let (ls, rs) = split xs in
  concat (dac_id ls) (dac_id rs)

dac_mts xs = mts (dac_id xs)
```

(b) The input to SuFu.

Fig. 5. D&C program restructuring as superfusion. The input to SuFu includes the template dac_id, which is the same for all D&C restructuring tasks, and a reference program dac_mts that composes the template with the original reference program mts. For simplicity, we consider only non-empty lists.

Let us illustrate this reduction taking divide-and-conquer (D&C) as an example. For programs over lists, D&C suggests dividing the input list into two halves, recursively computing the result for each half, and then combining the two results. Fig. 5a shows a D&C program for our running example, mts. To combine the mts of the two halves, this program introduces the sum of list elements as an auxiliary output. The intuition here is that the mts of the whole list is either the mts of the right half or the sum of elements in the right half plus the mts of the left half. As we can see, D&C restructuring is a challenging task: not only do we need to determine how to combine the recursive results, but often we also need to discover auxiliary outputs.

To reduce this problem to superfusion, we introduce a *template program* dac_id, shown in Fig. 5b. A careful reader will see that dac_id is simply the identity function with the recursive structure of a D&C program: splitting the input list into two halves, only to put them back together again.

Since the template program is the identity, we can compose it with our original reference program mts from Fig. 1a to obtain a new program dac_mts, shown in Fig. 5b, which behaves equivalently to mts. In this new program, the template dac_id produces an intermediate data structure, which we can annotate with Packed, and apply SuFu to eliminate it. This will have the effect of forcing SuFu to use the recursive structure of dac_id—that is, divide-and-conquer—to compute the result of mts. In fact, the result of applying SuFu to dac_mts is precisely the D&C program in Fig. 5a.

This approach can be generalized to other templates, beyond D&C. To this end, the first step is to implement a template function that returns the input unchanged, but follows the desired recursive structure; the second step is to compose the template function with the reference program, and finally, we apply SuFu to eliminate the intermediate data structure produced by the template function, thus getting a synthesized program in the target form. Another example of applying this approach can be found in the appendix [Ji et al. 2024a], which is for the template of single-pass recursion. Please note that the template and the Packed annotations need only be written once for each target form, since they are independent of the reference program.

6 IMPLEMENTATION

Our implementation of SuFu is in C++ and is available online [Ji et al. 2024a].

Program spaces. SuFu is parameterized by two program spaces: one for sketch holes and one for compression functions. We construct these two spaces as follows.

- The program space of sketch holes (i.e., $\mathcal{L}^{\text{scalar}}_{O(1)}$) is based on the SyGuS theory of conditional linear integer arithmetic [Alur et al. 2017a], which includes the *if-then-else* operator, arithmetic operators such as +, relational operators such as ≤, and Boolean operators such as *and*.

Besides, to operate on tuples and inductive data types, we augment this program space with tuple constructors and projections, and the pattern-match operator on inductive data types (*e.g.* `match ? with Nil -> ? | Cons(h,t) -> ?` for lists).
- The program space of compression functions includes all functions defined in the reference program[4], the *DeepCoder*'s library [Balog et al. 2017] of list functions, and the fold operators for all involved inductive data types. These fold operators enable SuFu to synthesize recursive compression functions not present in the reference program.

Verification. SuFu is based on the CEGIS framework and requires an external verifier to generate counterexamples. Following previous studies on synthesizing recursive programs [Miltner et al. 2022; Solar-Lezama et al. 2006; Torlak and Bodík 2014], we use bounded verification in our implementation (details in the appendix [Ji et al. 2024a]). To reduce the overhead, SuFu initializes the example set of CEGIS with 10^3 random examples. These examples can exclude the majority of the incorrect results and thus greatly reduce the number of invocations of the bounded verifier.

Solvers. We use Z3 [de Moura and Bjørner 2008] as the MaxSAT solver for sketch generation (Sec. 3.2) and PolyGen [Ji et al. 2021] as the PBE solver for sketch holes (Sec. 4.2).

7 EVALUATION

We design our evaluation to answer the following research questions.

- **RQ1**: How effective is SuFu in eliminating intermediate data structures?
- **RQ2**: How does SuFu compare to specialized program restructuring tools?
- **RQ3**: How does SuFu's synthesizer compare to general-purpose sketch solvers?

7.1 Experimental Setup

Baseline solvers. We compare SuFu with three inductive synthesizers. The first two are specialized solvers for program restructuring tasks that can be reduced to fusion:

- Synduce [Farzan et al. 2022] restructures recursive data-structure traversals according to a user-provided sketch (with the goal of making the traversals more efficient); Synduce uses a whitebox technique based on program unfolding, and requires both the reference and the target programs to traverse the data structure at most once.
- AutoLifter [Ji et al. 2024b] restructures programs into the divide-and-conquer (D&C) paradigm; it leverages domain-specific properties of D&C to decompose the synthesis problem.

SuFu is strictly more general than these two baselines and can be applied to all tasks in their domains. We have explained the reduction from D&C restructuring to fusion in Sec. 5; the reduction from the domain of Synduce to fusion can be found in the appendix [Ji et al. 2024a].

Our third baseline is Grisette [Lu and Bodík 2023], the most recent sketch solver at the time of writing. For this comparison, we first generate a sketch using the generation approach of SuFu (Sec. 3) and then solve this sketch using either SuFu (Algorithm 1) or Grisette.

Note that we do not perform an empirical comparison with traditional deductive fusion systems: to our knowledge, no automated fusion tool (including the most recent work [Hinze et al. 2010]) can synthesize auxiliary attributes—such as the sum of list elements in our `mts` example—which are essential for many tasks in our dataset.

Dataset. We collect a dataset of 290 tasks from three different sources, each task specified by a reference program with some data types annotated as `Packed`.

[4]In our evaluation, we only use functions defined in the original benchmark source and do not introduce new functions.

Table 4. Profile of our dataset. **#No-Aux** reports the number/percent of tasks that *do not* require inventing auxiliary attributes (which is an upper bound on the performance of deductive fusion). **Program Size** reports the number of AST nodes in the reference program per task. **Packed annotations** reports the number of Packed annotations per task, while **Unique** reports the total number of unique data types to be eliminated.

Source	#Tasks	#No-Aux		Program Size		Packed annotations		
				Mean	Max	Mean	Max	Unique
Fusion	16	11	69%	126.5	305	1.3	2	7
Recursion	178	129	73%	157.5	484	1.1	3	35
D&C	96	10	10%	251.2	843	1.0	1	1
Total	290	150	52%	186.8	843	1.1	3	37

Table 5. Performance of SuFu on the full dataset. **#Solved** reports the number/percent of tasks solved. **Time Cost** reports the average time costs (in seconds) of sketch generation (**Gen**) and sketch solving (**Syn**). **Result Size** reports the average number of AST nodes in the programs synthesized by SuFu, **Compress** for *?compress*, **Extract** for *?extract*, and **Holes** for the sketch holes.

Source	#Solved		Time Cost		Result Size		
			Gen	Syn	Compress	Extract	Holes
Fusion	14/16	88%	0.012	6.9	9.1	4.6	20.4
Recursion	170/178	96%	0.015	14.3	7.5	6.8	28.6
D&C	80/96	83%	0.022	49.0	11.6	6.5	85.9
Total	264/290	91%	0.017	24.4	8.8	6.6	45.5

- *Fusion.* We collect 16 tasks from fusion literature [Bird 1989; Bird and de Moor 1997; Gill et al. 1993; Hu et al. 1997; Wadler 1988]; 8 of them come from work on **manual** optimization [Bird 1989; Bird and de Moor 1997], which cannot be straightforwardly automated.
- *Recursion.* We include all 178 tasks from the original dataset of Synduce. In the process, we discovered that 60 of these tasks include manually provided auxiliary attributes, which presumably were added due to the limitations of Synduce. For example, in the `mts` task, the reference program returns not only the maximum tail sum but also the sum of list elements. We remove these auxiliary attributes when constructing our dataset.
- *Divide and Conquer (D&C).* We include all 96 tasks from the original dataset of AutoLifter.

We list some statistics of our dataset in Tab. 4. Note that tasks from different sources present different challenges. For example, *Fusion* tasks often require eliminating multiple data structures at once, while *Recursion* tasks require eliminating 35 different data types, including lists, trees, zippers, natural numbers, and expression ASTs. *D&C* tasks are the most challenging, both in terms of requiring auxiliary attributes and the program size. Overall, about half of the tasks in our dataset require auxiliary attributes, and hence are out of scope for deductive fusion systems.

Our experiments are conducted on Intel Core i7-8700 3.2GHz 6-Core Processor, with a timeout of 10 minutes per task. Our full dataset and results are available in the supplementary material.

7.2 RQ1: Overall Effectiveness

We summarize the performance of SuFu in Tab. 5. Overall, SuFu successfully solves 264 out of 290 tasks in the dataset (91%), taking around 24 seconds on average, and many of them (115 out of 264) require auxiliary attributes. Note that although SuFu uses MaxSAT for sketch generation, in practice this step takes only fractions of a second, and the run time is dominated by sketch solving. In terms of the size of the synthesized expressions, Tab. 5 confirms our key hypothesis that the

Table 6. Sample synthesis results. Terms filled into sketch holes are highlighted in red.

Task Description	Program Fragment

Synthesize a D&C program that calculates the maximum tail product of a (possibly negative) integer list.	`( if r.1 == r.3 then l.1 * r.1`
	`  else if l.2 * r.3 > r.1 then l.2 * r.3 else r.1,`
	`if r.3 < 0 then l.1 * r.2`
	`  else if l.2 * r.3 < r.2 then l.2 * r.3 else r.2,`
	`l.3 * r.3 )`

Given two languages $\mathcal{L}_1$ and $\mathcal{L}_2$ of Boolean expressions, synthesize the interpreter of $\mathcal{L}_1$ from the interpreter of $\mathcal{L}_2$.	`| Lit b -> b`
	`| Neglit b -> not b`
	`| And (e1, e2) -> (eval e1) && (eval e2)`
	`| Or (e1, e2) -> (eval e1) || (eval e2)`

Table 7. Comparison between SuFu and program restructuring tools, where **Time** is the mean time (in seconds) on tasks solved by both tools.

Source	Tool	#Solved	Time	Source	Tool	#Solved	Time
Recursion	SuFu	**170**	11.4	D&C	SuFu	80	46.8
	Synduce	125	**1.7**		AutoLifter	**82**	**15.6**

ghost function *?compress* is much smaller than the solutions to sketch holes, and hence the extra work of synthesizing *?compress* (as well as *?extract*) pays off to decompose the synthesis problem.

Quality of synthesized programs. We manually examined all synthesized programs and confirmed that they are *functionally equivalent* to the reference implementations on the entire input domain (though SuFu only uses bounded verification). In terms of *efficiency*, for *Recursion* and *D&C* tasks, we have a theoretical guarantee that SuFu's results have the same asymptotic complexity as those synthesized by Synduce and AutoLifter respectively; this guarantee is a direct consequence of our restriction on the program space, that is, that SuFu always fills sketch holes with $O(1)$-time expressions. Finally, for the *Fusion* tasks, we manually inspected the 14 programs produced by SuFu and compared them to the fused implementations from the original papers; we confirmed that (1) 13/14 programs are the same as the original, and (2) the remaining program is strictly more efficient, as SuFu fuses a sum over the list $[l, \ldots, r]$ into a constant-time expression $(l + r)(r - l + 1)/2$.

Sample programs. Tab. 6 shows fragments from two sample programs synthesized by SuFu. The first one illustrates the scalability of our tool: it presents a term synthesized for a single hole in a *D&C* task (the size is fairly typical for the tasks in this domain). The second one demonstrates the applicability of SuFu beyond lists: in particular, this task requires eliminating an AST.

7.3 RQ2: Comparison with Program Restructuring Tools

In this experiment, we run Synduce and AutoLifter on their original datasets, and compare their performance with SuFu. The results are shown in Tab. 7 and Fig. 6.

Compared with Synduce, SuFu takes more time on simple tasks but can eventually solve more tasks. We find that SuFu has a clear advantage on tasks that require inventing auxiliary attributes. Although Synduce can do this in principle, its whitebox approach is too restrictive for some of the tasks; in comparison, SuFu uses a blackbox approach, which receives less guidance from the reference program, but is more flexible in terms of supported auxiliary attributes.

Compared with AutoLifter, SuFu is slower but eventually (after about five minutes) solves a similar number of tasks. It is not surprizing that AutoLifter has an advantage on this domain:

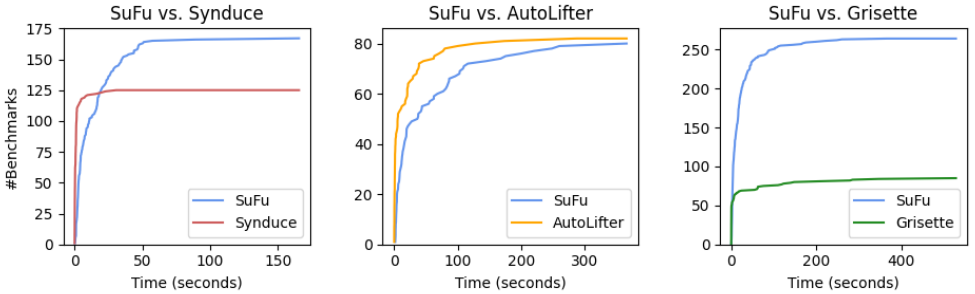


Fig. 6. Number of tasks solved by SuFu and each baseline solver over time. Comparisons with Synduce and AutoLifter are on their respective datasets, while the comparison with Grisette is on our full dataset.

Table 8. Comparison between SuFu and Grisette; **Time** is the mean time on tasks solved by both tools.

Source	Approach	#Solved	Time	Source	Approach	#Solved	Time
Fusion	SuFu	**14**	1.1	Recursion	SuFu	**170**	**11.4**
	Grisette	4	**0.7**		Grisette	60	20.0
D&C	SuFu	**80**	**11.5**	Total	SuFu	**264**	**11.1**
	Grisette	21	58.1		Grisette	85	28.9

like SuFu, it is a blackbox inductive synthesizer, but it implements domain-specific optimizations for divide-and-conquer algorithms, making its synthesis task simpler.

In summary, we observe that although SuFu is slower than specialized tools, it can solve a similar or higher number of tasks given enough time, while being strictly more general.

7.4 RQ3: Comparison with the Sketch Solver

In this experiment, we ablate SuFu's sketch solver, replacing it with an off-the-shelf sketch solver Grisette. The results are shown in Tab. 8 and Fig. 6. Overall, our synthesis algorithm significantly outperforms Grisette on both the number of solved tasks and the time cost. Predictably, SuFu's advantage is most pronounced on tasks that require synthesizing large expressions: for example, on *D&C*, where the average size of a sketch solution is a whopping 85.9 AST nodes, SuFu solves almost *four times more tasks* that Grisette and is more than *four times faster* on jointly solved tasks. This is because Grisette searches for all sketch holes simultaneously, while SuFu can effectively decompose the synthesis problem with the help of a (much smaller!) compression function.

7.5 Discussion

Failure analysis. SuFu fails to solve 26 out of 290 tasks in our dataset (Tab. 5). We identify three reasons for these failures. First, on 17 tasks, SuFu times out finding a valid compression function *?compress*. Second, on 4 tasks, SuFu succeeds in finding the expected *?compress*, but the underlying PBE solver times out synthesizing sketch holes, despite having local IO examples available.

Finally, on the last 5 tasks, SuFu uses an *unexpected* *?compress* whose sketch holes are more complex than necessary, making the PBE solver times out. Note that although our specification for *?compress* (Eq. 5) ensures the *existence* of valid sketch holes, their *size* may vary significantly regarding the choice of *?compress*. For example, one reference program in our dataset is `sqrsum (upto n)`, where `upto` constructs a `Packed` list of numbers from 1 to n, and `sqrsum` sums their squares. On this task, SuFu synthesizes an empty *?compress* because the integer n is enough to determine the output as $n(n+1)(2n+1)/6$, but this expression is too complex for the PBE solver to synthesize.

Verification. Perhaps the biggest limitation of our tool is that it performs only bounded verification. Although this is common in program synthesis with loops and recursion [Miltner et al. 2022; Solar-Lezama et al. 2006; Torlak and Bodík 2013], this makes SuFu unsuitable for applications where correctness is critical, such as compiler optimization. Fortunately, since SuFu can be combined with any verifier, it can automatically benefit from advances in verification technology. Effective verifiers already exist for some of the domains targeted by SuFu, such as structural recursion [K. et al. 2022] and D&C algorithms with a single-pass reference [Farzan and Nicolet 2017].

Moreover, we believe that the compression functions produced by SuFu as a by-product of synthesis can serve as useful lemmas that aid unbounded verification. For example, in the `mts` task, the compression function connects the second output of `tail'` to the sum of the list, *i.e.* `∀xs, sum xs = (tails' xs).2`. This is a necessary lemma for proving the correctness of the synthesis result `mts'`, and it is challenging for existing verifiers to conjecture this lemma out of thin air.

8 RELATED WORK

Fusion. Many deductive fusion systems have been designed to eliminate intermediate data structures. They iteratively apply pre-defined rules to rewrite the reference program toward the direction with fewer intermediate data structures. Representatives of such systems include the fold/unfold framework for handling generic recursion [Chin 1992; Gill et al. 1993; Hamilton 2001; Wadler 1988] and the program calculation framework dealing with specific forms of recursive functions [Bird 1989; Bird and de Moor 1997; Hinze et al. 2010; Meijer et al. 1991; Takano and Meijer 1995].

Compared with these deductive systems, SuFu has both advantages and limitations:

- On the one hand, deductive fusion is correct by construction, generally faster, and does not require users to annotate the intermediate data structures to be eliminated.
- On the other hand, SuFu achieves significantly better expressiveness via inductive synthesis.

Besides deductive transformation, there is another line of work that achieves fusion by permuting the instructions in the reference program [Sakka et al. 2017; Sundararajah and Kulkarni 2019; Wang et al. 2021]. Although these approaches work well in certain domains, they can hardly be applied to our dataset because many of our tasks require generating entirely new expressions that cannot be obtained by shuffling around the sub-terms in the reference program.

Synthesizing efficient programs. There are several lines of previous research on synthesizing efficient programs. First, *superoptimization* [Bornholt et al. 2016; Phothilimthana et al. 2014, 2016; Schkufza et al. 2013; Sharma et al. 2015] uses inductive synthesis to generate the most efficient implementation for the reference program. However, existing superoptimization approaches consider only low-level, loop-free programs, and thus cannot be applied to our problem.

Second, *program restructuring* tools rewrite a reference program into a given target form that is known to be efficient [Acar et al. 2005; Farzan et al. 2022; Farzan and Nicolet 2017, 2021a; Fedyukovich et al. 2017; Ji et al. 2024b; Morita et al. 2007; Pu et al. 2011]. Each one of this tools is designed to handle a specific target form, and thus cannot be applied to the general fusion problem. On the other hand, as discussed in Sec. 5, program restructuring for some target forms [Farzan et al. 2022; Ji et al. 2024b] can be reduced to fusion and solved by SuFu.

Finally, *resource-aware synthesis* [Hu et al. 2021; Knoth et al. 2019] aims to find a program satisfying an efficiency requirement specified in a type system. Compared with SuFu, these approaches can deal with more refined efficiency requirements via complex type systems but do not scale to synthesizing large programs because they synthesize the whole program from scratch.

Other related program synthesis approaches. First, since the core synthesis problem of SuFu is a sketch problem, SuFu is related to previous sketch solvers [Jeon et al. 2015; Lu and Bodík 2023; Lubin et al. 2020; Porncharoenwase et al. 2022; Solar-Lezama et al. 2006; Torlak and Bodík 2014].

General-purposed sketch solvers, however, can hardly scale to superfusion tasks, where the target programs of sketch holes are typically large.

Second, SuFu addresses the scalability challenge by decomposing the sketch problem into subproblems. Hence, SuFu is related to previous approaches for decomposing synthesis tasks.

- *Angelic synthesis* (or *uninterpreted functions*) [Ji et al. 2024b; Kuncak and Blanc 2013; Singh et al. 2014] uses the congruence property of functions (i.e., identical inputs imply identical outputs) to eliminate unknown programs from a complex specification. It is also the key idea of quantifier elimination in SuFu. However, these previous techniques cannot be directly applied to eliminate sketch holes in our tasks because the program space of sketch holes is limited to $O(1)$-time programs for the sake of efficiency. We cannot regard such a sketch hole as a whole as an uninterpreted function, because most functions operating on data structures cannot be implemented in $O(1)$-time. To address this issue, SuFu decomposes the sketch hole into *?extract* and *?comb* and applies the congruence property only to *?comb*.
- *Model learning* [Huang and Qiu 2022] synthesizes models (e.g., pre- and post-conditions) to replace concrete invocations of library functions and thus avoid analyzing complex library functions during synthesis. This process requires input-output oracles for library functions. However, in our case, such oracles are not available, neither for the compression function nor for the sketch holes, because we do not even know the type of the compressed data structures (*i.e.* the scalar attributes) these programs operate on, let alone the concrete values.

Finally, Revamp [Pailoor et al. 2024] solves a related synthesis problem, dubbed *code refactoring for abstract data types (ADTs)*. Given an original implementation of an ADT and a relational specification relating the original data representation to a new one, Revamp synthesizes a new ADT implementation using the new representation. SuFu is similar to Revamp in that both are concerned with replacing data structures in a program, and our compression function is similar to the relational specification in Revamp. Despite these similarities, we believe Revamp and SuFu are complementary. Specifically, Revamp supports replacing a data structure with another complex data structure but requires the user to manually specify the relation connecting the two. In contrast, SuFu automatically synthesizes the relation (i.e., the compression function) but currently only supports compressing data structures into scalar values.

9 CONCLUSION

In this paper, we present *superfusion*, a novel approach to eliminating intermediate data structures in functional programs using inductive program synthesis. Given a reference program annotated with data structures to be eliminated, superfusion first generates a sketch by transforming the reference program into an intermediate language and then fills the sketch holes with $O(1)$-time expressions. To make the synthesis scale, our approach synthesizes a ghost function *?compress* and uses it to decompose the sketch problem into independent synthesis problems for each hole. We implement superfusion in a tool called SuFu and evaluate it on a dataset of 290 tasks collected from previous studies. The results demonstrate SuFu's superior expressiveness compared to existing tools for fusion and program restructuring.

The two most exciting directions for future work are (1) unbounded verification and (2) support for non-scalar compression functions. Unbounded verification will enable using SuFu in settings like compiler optimizations, where correctness is critical and the user cannot be expected to inspect the synthesized program. Supporting arbitrary complex data structures as outputs of the compression function will enable SuFu to be used in a broader range of applications, promoting it from a fusion tool to a general-purpose program optimization tool.

ACKNOWLEDGEMENTS

We thank all anonymous reviewers of this paper for their valuable suggestions. This work is sponsored by the National Key Research and Development Program of China under Grant No. 2022YFB4501902, the National Natural Science Foundation of China under Grant Nos. 62161146003, and the National Science Foundation (USA) under Grants No. 1943623 and 1911149.

DATA AVAILABILITY

The software artifact for this paper is available online [Ji et al. 2024a] and will be maintained on GitHub[5]. It contains the source code of SuFu with a build script, our dataset and all experimental results, and a README file that provides step-by-step instructions for running SuFu and reproducing the experiments. This artifact also contains the appendix of this paper that supplies two more examples of SuFu, proofs of theorems in Sec. 4.6, and some implementation details.

REFERENCES

Umut A Acar et al. 2005. *Self-adjusting computation.* Ph. D. Dissertation. Carnegie Mellon University.

Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. 2017a. SyGuS-Comp 2017: Results and Analysis. In *Proceedings Sixth Workshop on Synthesis, SYNT@CAV 2017, Heidelberg, Germany, 22nd July 2017.* 97–115. https://doi.org/10.4204/EPTCS.260.9

Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. 2017b. Scaling Enumerative Program Synthesis via Divide and Conquer. In *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I.* 319–336. https://doi.org/10.1007/978-3-662-54577-5_18

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

Richard S. Bird. 1989. Algebraic Identities for Program Calculation. *Comput. J.* 32, 2 (1989), 122–126. https://doi.org/10.1093/comjnl/32.2.122

Richard S. Bird and Oege de Moor. 1997. *Algebra of programming.* Prentice Hall.

Nikolaj Bjørner, Ken McMillan, and Andrey Rybalchenko. 2013. On Solving Universally Quantified Horn Clauses. In *Static Analysis*, Francesco Logozzo and Manuel Fähndrich (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 105–125.

James Bornholt, Emina Torlak, Dan Grossman, and Luis Ceze. 2016. Optimizing synthesis with metasketches. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016.* 775–788. https://doi.org/10.1145/2837614.2837666

Wei-Ngan Chin. 1992. Safe Fusion of Functional Expressions. In *Proceedings of the Conference on Lisp and Functional Programming, LFP 1992, San Francisco, California, USA, 22-24 June 1992*, Jon L. White (Ed.). ACM, 11–20. https://doi.org/10.1145/141471.141494

Duncan Coutts, Roman Leshchinskiy, and Don Stewart. 2007. Stream Fusion: From Lists to Streams to Nothing at All. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming* (Freiburg, Germany) *(ICFP '07)*. Association for Computing Machinery, New York, NY, USA, 315âĂŞ326. https://doi.org/10.1145/1291151.1291199

Leonardo Mendonça de Moura and Nikolaj S. Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 4963)*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

Marc Dikotter. 1990. Book review: The Definition of Standard ML by R. Milner, M. Torte, R. Harper. *SIGARCH Comput. Archit. News* 18, 4 (1990), 91. https://doi.org/10.1145/121973.773545

Azadeh Farzan, Danya Lette, and Victor Nicolet. 2022. Recursion synthesis with unrealizability witnesses. In *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, Ranjit Jhala and Isil Dillig (Eds.). ACM, 244–259. https://doi.org/10.1145/3519939.3523726

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 2017, Barcelona, Spain, June 18-23, 2017.* 540–555. https://doi.org/10.1145/3062341.3062355

[5]https://github.com/jiry17/SuFu

Azadeh Farzan and Victor Nicolet. 2021a. Counterexample-Guided Partial Bounding for Recursive Function Synthesis. In *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 12759)*, Alexandra Silva and K. Rustan M. Leino (Eds.). Springer, 832–855. https://doi.org/10.1007/978-3-030-81685-8_39

Azadeh Farzan and Victor Nicolet. 2021b. Phased synthesis of divide and conquer programs. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 974–986. https://doi.org/10.1145/3453483.3454089

Grigory Fedyukovich, Maaz Bin Safeer Ahmad, and Rastislav Bodík. 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 2017, Barcelona, Spain, June 18-23, 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 572–585. https://doi.org/10.1145/3062341.3062382

Maarten M. Fokkinga. 1992. *Law and order in algorithmics.* Univ. Twente.

Andrew John Gill, John Launchbury, and Simon L. Peyton Jones. 1993. A Short Cut to Deforestation. In *Proceedings of the conference on Functional programming languages and computer architecture, FPCA 1993, Copenhagen, Denmark, June 9-11, 1993*, John Williams (Ed.). ACM, 223–232. https://doi.org/10.1145/165180.165214

Geoff W. Hamilton. 2001. Extending Higher-Order Deforestation: Transforming Programs to Eliminate Even More Trees. In *Selected papers from the 3rd Scottish Functional Programming Workshop (SFP01), University of Stirling, Bridge of Allan, Scotland, August 22nd to 24th, 2001 (Trends in Functional Programming, Vol. 3)*, Kevin Hammond and Sharon Curtis (Eds.). Intellect, 25–36.

Robert Harper. 2016. *Practical Foundations for Programming Languages (2nd. Ed.).* Cambridge University Press. https://www.cs.cmu.edu/%7Erwh/pfpl/index.html

Ralf Hinze, Thomas Harper, and Daniel W. H. James. 2010. Theory and Practice of Fusion. In *Implementation and Application of Functional Languages - 22nd International Symposium, IFL 2010, Alphen aan den Rijn, The Netherlands, September 1-3, 2010, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 6647)*, Jurriaan Hage and Marco T. Morazán (Eds.). Springer, 19–37. https://doi.org/10.1007/978-3-642-24276-2_2

Qinheping Hu, John Cyphert, Loris D'Antoni, and Thomas W. Reps. 2021. Synthesis with Asymptotic Resource Bounds. In *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 12759)*, Alexandra Silva and K. Rustan M. Leino (Eds.). Springer, 783–807. https://doi.org/10.1007/978-3-030-81685-8_37

Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi. 1996. Deriving Structural Hylomorphisms From Recursive Definitions. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming, ICFP 1996, Philadelphia, Pennsylvania, USA, May 24-26, 1996*, Robert Harper and Richard L. Wexelblat (Eds.). ACM, 73–82. https://doi.org/10.1145/232627.232637

Zhenjiang Hu, Hideya Iwasaki, Masato Takeichi, and Akihiko Takano. 1997. Tupling Calculation Eliminates Multiple Data Traversals. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP '97), Amsterdam, The Netherlands, June 9-11, 1997*, Simon L. Peyton Jones, Mads Tofte, and A. Michael Berman (Eds.). ACM, 164–175. https://doi.org/10.1145/258948.258964

Kangjing Huang and Xiaokang Qiu. 2022. Bootstrapping Library-Based Synthesis. In *Static Analysis - 29th International Symposium, SAS 2022, Auckland, New Zealand, December 5-7, 2022, Proceedings (Lecture Notes in Computer Science, Vol. 13790)*, Gagandeep Singh and Caterina Urban (Eds.). Springer, 272–298. https://doi.org/10.1007/978-3-031-22308-2_13

Jinseong Jeon, Xiaokang Qiu, Jeffrey S. Foster, and Armando Solar-Lezama. 2015. JSketch: sketching for Java. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, Elisabetta Di Nitto, Mark Harman, and Patrick Heymans (Eds.). ACM, 934–937. https://doi.org/10.1145/2786805.2803189

Ruyi Ji, Jingtao Xia, Yingfei Xiong, and Zhenjiang Hu. 2021. Generalizable synthesis through unification. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–28. https://doi.org/10.1145/3485544

Ruyi Ji, Yuwei Zhao, Nadia Polikarpova, Yingfei Xiong, and Zhenjiang Hu. 2024a. *Artifact for PLDI'24: Superfusion: Eliminating Intermediate Data Structures via Inductive Synthesis.* https://doi.org/10.5281/zenodo.10951760

Ruyi Ji, Yuwei Zhao, Yingfei Xiong, Di Wang, Lu Zhang, and Zhenjiang Hu. 2024b. Decomposition-Based Synthesis for Applying Divide-and-Conquer-Like Algorithmic Paradigms. *ACM Transactions on Programming Languages and Systems* (2024).

Hari Govind V. K., Sharon Shoham, and Arie Gurfinkel. 2022. Solving constrained Horn clauses modulo algebraic data types and recursive functions. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–29. https://doi.org/10.1145/3498722

Tristan Knoth, Di Wang, Nadia Polikarpova, and Jan Hoffmann. 2019. Resource-guided program synthesis. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*. 253–268. https://doi.org/10.1145/3314221.3314602

Viktor Kuncak and Régis Blanc. 2013. Interpolation for synthesis on unbounded domains. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*. IEEE, 93–96. https://ieeexplore.ieee.org/document/6679396/

Harry R. Lewis. 1978. Complexity of Solvable Cases of the Decision Problem for the Predicate Calculus. In *19th Annual Symposium on Foundations of Computer Science, Ann Arbor, Michigan, USA, 16-18 October 1978*. IEEE Computer Society, 35–47. https://doi.org/10.1109/SFCS.1978.9

Sirui Lu and Rastislav Bodík. 2023. Grisette: Symbolic Compilation as a Functional Programming Library. *Proc. ACM Program. Lang.* 7, POPL (2023), 455–487. https://doi.org/10.1145/3571209

Justin Lubin, Nick Collins, Cyrus Omar, and Ravi Chugh. 2020. Program sketching with live bidirectional evaluation. *Proc. ACM Program. Lang.* 4, ICFP (2020), 109:1–109:29. https://doi.org/10.1145/3408991

Henry Massalin. 1987. Superoptimizer: A Look at the Smallest Program. *SIGARCH Comput. Archit. News* 15, 5 (oct 1987), 122âĂŞ126. https://doi.org/10.1145/36177.36194

Erik Meijer, Maarten M. Fokkinga, and Ross Paterson. 1991. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In *Functional Programming Languages and Computer Architecture, 5th ACM Conference, Cambridge, MA, USA, August 26-30, 1991, Proceedings (Lecture Notes in Computer Science, Vol. 523)*, John Hughes (Ed.). Springer, 124–144. https://doi.org/10.1007/3540543961_7

Anders Miltner, Adrian Trejo Nuñez, Ana Brendel, Swarat Chaudhuri, and Isil Dillig. 2022. Bottom-up synthesis of recursive functional programs using angelic execution. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–29. https://doi.org/10.1145/3498682

Kazutaka Morita, Akimasa Morihata, Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi. 2007. Automatic inversion generates divide-and-conquer parallel programs. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, Jeanne Ferrante and Kathryn S. McKinley (Eds.). ACM, 146–155. https://doi.org/10.1145/1250734.1250752

Shankara Pailoor, Yuepeng Wang, and Isil Dillig. 2024. Semantic Code Refactoring for Abstract Data Types. *Proc. ACM Program. Lang.* 8, POPL (2024), 816–847. https://doi.org/10.1145/3632870

Phitchaya Mangpo Phothilimthana, Tikhon Jelvis, Rohin Shah, Nishant Totla, Sarah E. Chasins, and Rastislav Bodík. 2014. Chlorophyll: synthesis-aided compiler for low-power spatial architectures. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 396–407. https://doi.org/10.1145/2594291.2594339

Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodík, and Dinakar Dhurjati. 2016. Scaling up Superoptimization. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '16, Atlanta, GA, USA, April 2-6, 2016*. 297–310. https://doi.org/10.1145/2872362.2872387

Sorawee Porncharoenwase, Luke Nelson, Xi Wang, and Emina Torlak. 2022. A formal foundation for symbolic evaluation with merging. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–28. https://doi.org/10.1145/3498709

Yewen Pu, Rastislav Bodík, and Saurabh Srivastava. 2011. Synthesis of first-order dynamic programming algorithms. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*, Cristina Videira Lopes and Kathleen Fisher (Eds.). ACM, 83–98. https://doi.org/10.1145/2048066.2048076

Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. 2008. Liquid Types. *SIGPLAN Not.* 43, 6 (jun 2008), 159âĂŞ169. https://doi.org/10.1145/1379022.1375602

Laith Sakka, Kirshanthan Sundararajah, and Milind Kulkarni. 2017. TreeFuser: a framework for analyzing and fusing general recursive tree traversals. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 76:1–76:30. https://doi.org/10.1145/3133900

Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic superoptimization. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS 2013, Houston, TX, USA, March 16-20, 2013*, Vivek Sarkar and Rastislav Bodík (Eds.). ACM, 305–316. https://doi.org/10.1145/2451116.2451150

Rahul Sharma, Eric Schkufza, Berkeley R. Churchill, and Alex Aiken. 2015. Conditionally correct superoptimization. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, Jonathan Aldrich and Patrick Eugster (Eds.). ACM, 147–162. https://doi.org/10.1145/2814270.2814278

Rohit Singh, Rishabh Singh, Zhilei Xu, Rebecca Krosnick, and Armando Solar-Lezama. 2014. Modular Synthesis of Sketches Using Models. In *Verification, Model Checking, and Abstract Interpretation - 15th International Conference, VMCAI 2014, San Diego, CA, USA, January 19-21, 2014, Proceedings (Lecture Notes in Computer Science, Vol. 8318)*, Kenneth L. McMillan and Xavier Rival (Eds.). Springer, 395–414. https://doi.org/10.1007/978-3-642-54013-4_22

Armando Solar-Lezama. 2009. The Sketching Approach to Program Synthesis. In *Programming Languages and Systems, 7th Asian Symposium, APLAS 2009, Seoul, Korea, December 14-16, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5904)*, Zhenjiang Hu (Ed.). Springer, 4–13. https://doi.org/10.1007/978-3-642-10672-9_3

Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. 2006. Combinatorial sketching for finite programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006*. 404–415. https://doi.org/10.1145/1168857.1168907

Kirshanthan Sundararajah and Milind Kulkarni. 2019. Composable, sound transformations of nested recursion and loops. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 902–917. https://doi.org/10.1145/3314221.3314592

Akihiko Takano and Erik Meijer. 1995. Shortcut Deforestation in Calculational Form. In *Proceedings of the seventh international conference on Functional programming languages and computer architecture, FPCA 1995, La Jolla, California, USA, June 25-28, 1995*, John Williams (Ed.). ACM, 306–313. https://doi.org/10.1145/224164.224221

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*, Antony L. Hosking, Patrick Th. Eugster, and Robert Hirschfeld (Eds.). ACM, 135–152. https://doi.org/10.1145/2509578.2509586

Emina Torlak and Rastislav Bodík. 2014. A lightweight symbolic virtual machine for solver-aided host languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 530–541. https://doi.org/10.1145/2594291.2594340

Philip Wadler. 1988. Deforestation: Transforming Programs to Eliminate Trees. In *ESOP '88, 2nd European Symposium on Programming, Nancy, France, March 21-24, 1988, Proceedings (Lecture Notes in Computer Science, Vol. 300)*, Harald Ganzinger (Ed.). Springer, 344–358. https://doi.org/10.1007/3-540-19027-9_23

Yanjun Wang, Jinwei Liu, Dalin Zhang, and Xiaokang Qiu. 2021. Reasoning about recursive tree traversals. In *PPoPP '21: 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Virtual Event, Republic of Korea, February 27- March 3, 2021*, Jaejin Lee and Erez Petrank (Eds.). ACM, 47–61. https://doi.org/10.1145/3437801.3441617

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