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:

‘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.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

© 2024 Copyright held by the owner/author(s).

ACM 2473-1421/2024/6-ART185
https://doi.org/10.1145/3656415

Proc. ACM Program. Lang., Vol. 8, No. PLDI, Article 185. Publication date: June 2024.
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, \( mts \left[ 1, -2, 3, -1, 2 \right] = 4 \), the sum of the last three elements \( [3, -1, 2] \). This function can
be implemented by composing list functions, like so:

\[
mts \; xs = \max \left( \text{map} \; \text{sum} \; (\text{tails} \; xs) \right)
\]

where \( \text{tails} \) returns a (nested) list of all tails of the input, \( \text{map} \) applies the function \( \text{sum} \) to each tail
and obtains the list of all tail sums, and \( \max \) 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 \texttt{Data.List} in Haskell. However, this program is also inefficient due to
the large intermediate data structure constructed by \( \text{tails} \), 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.

\[
mts' \; xs = (\text{tails'} \; xs) \cdot 1
\]

where
\[
tails' \; \text{Nil} = (0, 0)
tails' \; \text{Cons}(h, t) = \begin{cases} 
(\max \; tmts \; (tsum + h), \; \text{tsum} + h) & \text{if } tmts \neq 0 \\
(0, 0) & \text{otherwise}
\end{cases}
\]

The inefficiency of compositional programs can often be ad-
ressed 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.

**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 opti-
mizations, 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 \( \text{tails} \) 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

Proc. ACM Program. Lang., Vol. 8, No. PLDI, Article 185. Publication date: June 2024.
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):

\[
?compress ts = (\text{maximum } (\text{map sum ts}), \text{sum } (\text{head ts}))
\]

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 turn 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.
List = Nil | Cons(Int, List)
NList = NNil | NCons(List, NList)
tails Nil = NCons(Nil, NNil)
tails Cons(_, t)@xs = NCons(xs, tails t)
tails :: List -> Packed NList
mts xs = maximum (map sum (tails xs))

(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 ?t1, ?t2, and ?t3 in Fig. 1b: ?t1 and ?t2 produce an NList, while ?t2 and ?t3 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 \( \texttt{compress} \); concrete local examples obtained using the \( \texttt{compress} \) in Eq. 1.

<table>
<thead>
<tr>
<th>Hole</th>
<th>Original Term</th>
<th>Original IO Behavior</th>
<th>Local Examples Symbolic Form</th>
<th>Concrete From</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t_1() )</td>
<td>NCons(Nil, NNil)</td>
<td>( t_1() = [[]] )</td>
<td>( t_1() = \texttt{compress}([[[]])] )</td>
<td>( t_1() = (0, 0) )</td>
</tr>
<tr>
<td>( t_2(xs, ts) )</td>
<td>NCons(xs, ts)</td>
<td>( t_2([[2], [[]]]) )</td>
<td>( t_2([[2], \texttt{compress}([[]])] )</td>
<td>( t_2([[2], (0, 0)]) )</td>
</tr>
<tr>
<td>( t_3 ) ts</td>
<td>maximum (map sum ts)</td>
<td>( t_3([[2], [[]]] = 2 )</td>
<td>( t_3(\texttt{compress}([[[2], [[]]]) = 2 )</td>
<td>( t_3(2, 2) = 2 )</td>
</tr>
</tbody>
</table>

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 \( \sim 30\% \) 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, \( \texttt{compress} \), 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 \( \texttt{compress} \), which knew to compress the list of tails intro two scalar attributes, the maximum tail sum and the sum of list elements:

\[
\texttt{compress} \ ts := \left( \text{maximum (map sum ts), sum (head ts)} \right)
\]
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 $\texttt{compress}$ 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 $\texttt{compress}$ and all sketch holes simultaneously, but that defeats the purpose of introducing $\texttt{compress}$ 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 ($\texttt{compress}$ 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 $\texttt{compress}$, 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. $\texttt{compress ts := ()}$. Substituting this definition into the local examples, we find that the constraints for hole $?t_3$ are unrealizable:

$$
\exists ?t_3 \in L^{\text{int}}_{O(1)}, \ ?t_3 () = 2 \land \ ?t_3 () = 1 \land \ldots
$$

Here $L^{\text{int}}_{O(1)}$ 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 $\texttt{compress}$ that provides enough information to differentiate between the two executions.

More formally, we need to find a function $\texttt{compress}_1$ that satisfies the following specification:

$$
\exists ?t_3 \in L^{\text{int}}_{O(1)}, \ ?t_3 (\texttt{compress}_1 [[2], []]) = 2 \land \ ?t_3 (\texttt{compress}_1 [[2, -1], [-1], []]) = 1 \land \ldots
$$
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 \( \texttt{compress}_1 \):

\[
\texttt{compress}_1 \ ts := \ \textit{maximum} \ (\textit{map} \ ts)
\]

**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 \( \texttt{compress} := \texttt{compress}_1 \) into the local examples, we get the following for hole \( ?t_2 \):

\[
\exists ?t_2 \in L_{O(1)}^{\text{int}}: \ ?t_2 \ ([2], 0) = 2 \land \ ?t_2 \ ([2, -1], 0) = 1 \land \ldots
\]

This specification is also unrealizable. Intuitively, this is because the \( nts \) 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 \( nts \) 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 \( nts \) from \( xs \) in \( O(1) \) time since \( xs \) can be arbitrarily long.

To fix this unrealizable hole, we need to add another attribute, \( \texttt{compress}_2 \), to the compression function, satisfying the specification:

\[
\exists ?t_2 \in L_{O(1)}^{\text{int}}: \ ?t_2 \ ([2], \ (0, \texttt{compress}_2 \ [[[]]])) = 2 \land \ ?t_2 \ ([2, -1], \ (0, \texttt{compress}_2 \ [[[-1]], [[]]])) = 1 \land \ldots
\]

(3)

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

\[
\texttt{compress}_2 \ ts := \ \textit{sum} \ (\textit{head} \ ts)
\]

**Iteration 3.** Once we substitute \( \texttt{compress} := (\texttt{compress}_1 \ ts, \texttt{compress}_2 \ ts) \) into local examples, we find that all sketch holes are realizable, which concludes the synthesis of \( \texttt{compress} \).

2.3.2 Quantifier Elimination. We conclude this section by explaining how SuFu synthesizes each scalar attribute of \( \texttt{compress} \) from the specifications Eq. 2 and Eq. 3.

**Iteration 1.** Let us consider the specification for \( \texttt{compress}_1 \) in Eq. 2. This specification involves two unknown programs, \( \texttt{compress}_1 \) and \( ?t_3 \), and our goal is to avoid synthesizing both of them simultaneously. To this end, we observe that:

1. to synthesize \( \texttt{compress}_1 \), 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 \( \texttt{compress}_1 \):

\[
\forall (ts, out), (ts', out') \in E_3. \ (\texttt{compress}_1 \ ts = \texttt{compress}_1 \ ts') \rightarrow (out = out')
\]

(4)

where \( E_3 \) is the set of local examples collected for \( ?t_3 \) from the executions of \( nts \) (such as the last row, the Example column of Tab. 1).

We can now enumerate candidate \( \texttt{compress}_1 \) from smaller to larger, checking them against the specification Eq. 4. SuFu can find the desired program \( \texttt{compress}_1 \ ts := \textit{maximum} \ (\textit{map} \ \textit{sum} \ ts) \) from 21 executions of \( nts \). Tab. 2 illustrates three smaller but incorrect candidates for \( \texttt{compress}_1 \), 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 $\comps{1}$ and examples sufficient to reject them based on Eq. 4.

<table>
<thead>
<tr>
<th>Candidate Program</th>
<th>Execution</th>
<th>Symbolic Local Example for $t_3$</th>
<th>Conflict</th>
</tr>
</thead>
<tbody>
<tr>
<td>length of the list of tails</td>
<td>mts [2]</td>
<td>$t_3 (\comps{} [2], []) = 2$</td>
<td>$t_3 2 = 2$</td>
</tr>
<tr>
<td>$\comps{} t_s := \text{length ts}$</td>
<td>mts [1]</td>
<td>$t_3 (\comps{} [1], []) = 1$</td>
<td>$t_3 2 = 1$</td>
</tr>
<tr>
<td>sum of elements</td>
<td>mts [2]</td>
<td>$t_3 (\comps{} [2], []) = 2$</td>
<td>$t_3 2 = 2$</td>
</tr>
<tr>
<td>$\comps{} t_s := \text{sum (head ts)}$</td>
<td>mts [-1, 3]</td>
<td>$t_3 (\comps{} [-1, 3], [3], []) = 3$</td>
<td>$t_3 2 = 3$</td>
</tr>
<tr>
<td>maximum of elements</td>
<td>mts [2]</td>
<td>$t_3 (\comps{} [2], []) = 2$</td>
<td>$t_3 2 = 2$</td>
</tr>
<tr>
<td>$\comps{} t_s := \text{maximum (head ts)}$</td>
<td>mts [2, -1]</td>
<td>$t_3 (\comps{} [2, -1], [-1], []) = 1$</td>
<td>$t_3 2 = 1$</td>
</tr>
</tbody>
</table>

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 $\comps{}$ 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:

$$t_2 (xs, ts) := \comps{2} (\comps{3} (xs, ts))$$

where $\comps{3}$ extracts a tuple of scalar values from the input variables and $\comps{2}$ combines them into the final result. With this decomposition in place, we can now use our previous technique to eliminate $\comps{2}$ (which operates on scalars), and then enumerate $\comps{3}$ jointly with $\comps{}$ (note that $\comps{3}$ is small because it does not perform any actual computation), resulting in:

$$\comps{3} (xs, ts) := (ts.1, ts.2, head xs) \quad \comps{} ts := \text{sum (head ts)}$$

### 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 $\lambda_{sk}$, which makes the scope of the holes and all uses of intermediate data structures explicit. Importantly, any well-typed $\lambda_{sk}$ 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 $\lambda_{sk}$ is shown in Fig. 2. The language is a simply typed lambda calculus with inductive data types, which we augment with the following annotations:
Fig. 2. SuFu’s intermediate language $\lambda_{sk}$ presented via abstract binding trees [Harper 2016]. We use $\text{ind}(\cdot)$ 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 $\text{Packed}$ annotations provided by the user.

$$
\begin{align*}
  t \in \text{Term} & \quad := \quad \text{constants} \mid x \mid \text{app}(t_1, t_2) \mid \lambda(x, T, t) \mid \text{fix}(t) \mid \text{let}(x, t_1, t_2) \mid \text{if}(t_1, t_2, t_3) \mid C(t) \\
  T \in \text{Type} & \quad := \quad B \mid T \rightarrow T \mid T_1 \times \cdots \times T_n
\end{align*}
$$

$$
B \in \text{BaseType} := \text{Unit} \mid \text{Int} \mid \text{Bool} \mid B_1 \times \cdots \times B_n \mid \text{ind}(C) \mid \text{Packed}(B)
$$

$$
S \in \text{ScalarType} := \text{Unit} \mid \text{Int} \mid \text{Bool} \mid S_1 \times \cdots \times S_n \mid \text{Packed}(B).
$$

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

- At the type level, we introduce $\text{Packed}$ annotations, which are treated as a type constructor: $\text{Packed} T$ is a new type, different from $T$. Besides, $\lambda_{sk}$ treats $\text{Packed} T$ as a scalar type because, intuitively, users annotate data structures as $\text{Packed}$ 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 $\text{Packed}$ values; for example, $\text{label}[1, 2]$ is of type $\text{Packed} \text{List}$ and $\text{unlabel}(\text{label}[1, 2]) = [1, 2]$. 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 $\lambda_{sk}$ 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 $\text{mts}$ program from Fig. 1a. Here the user has annotated the output of tails with $\text{Packed}$, thereby making the program ill-typed in $\lambda_{sk}$: specifically, the body of tails is ill-typed because it produces an $\text{NList}$ where a $\text{Packed} \text{NList}$ is expected, and conversely, the body of $\text{mts}$ is ill-typed because tails $xs$ returns a $\text{Packed} \text{NList}$ where an $\text{NList}$ is expected by $\text{map sum}$. The task of the elaboration process is to insert label, unlabel, and rewrite annotations to make the program well-typed in $\lambda_{sk}$. Consider four possible elaborations of the body of $\text{mts}$:

$$
\begin{align*}
  \text{maximum (map sum (unlabel (tails xs)))} & \quad (A) \\
  \text{maximum (rewrite (map sum (unlabel (tails xs))))} & \quad (B) \\
  \text{rewrite (maximum (map sum (unlabel (tails xs)))))} & \quad (C) \\
  \text{let ts = tails xs in rewrite (maximum (map sum (unlabel ts))))} & \quad (D)
\end{align*}
$$

Term $A$ inserts an $\text{unlabel}$ to provide the correct argument type to $\text{map sum}$; it is still ill-typed, however, because $\text{unlabel}$ 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...
Fig. 4. The sketch for mts from Fig. 1a.

Algorithm 1: Sketch solving.

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

1. $E_{sym} \leftarrow \bigcup_{i \in I} \text{CollectExamples}(p, in)$;
2. $compress \leftarrow \text{CompressSynthesis}(E_{sym})$;
3. $E_{io} \leftarrow \text{Subst}(E_{sym}, compress)$
4. foreach sketch hole $h$ do
   5. $t_i \leftarrow \text{PBESolver}(L^{\text{scalar}}_O(1), E_{io}[h])$;
5. end
6. return $(\text{output type of } compress, \overline{t_i})$;

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 $\lambda sk$ (potentially ill-typed due
to Packed annotations), find a sketch, i.e. a program $p'$ in $\lambda sk$, 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 $\lambda_{sk}$ 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 $\lambda_{sk}$ is a pair $(\overline{T}, \overline{t})$ of a list of types $T_i$ and a list of terms $t_h$ satisfying the following conditions.

- $\overline{T}$ contains only scalar types in the surface language.
- $\overline{t}$ 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 $\overline{T}$ and $\overline{t}$ are respectively equal to the number of Packed and rewrite annotations.

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

Example 4.2. For the sketch $mts\_label$ in Fig. 4, the desired solution is to take the singleton list $[\text{Int} \times \text{Int}]$ as $\overline{T}$ and take the three highlighted terms in Fig. 1c as $\overline{t}$.

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 $\lambda_{sk}$ 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, $|\overline{T}| = 1$). 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 $E_{sym}$ 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 $E_{io}$ (Line 3). Finally, SuFu uses a PBE solver to synthesize a solution for each sketch $h$ from its concrete examples, $E_{io}[h]$ (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 $\lambda_{sk}$ 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.
\[(xs \mapsto [2, -1]) \vdash tails\ xs \parallel \mathsf{label} \ [[2, -1], [-1], []] \]

\[(xs \mapsto [2, -1], ts \mapsto \mathsf{label} \ [[2, -1], [-1], []]) \vdash rewrite \ (maximum \ (map \ sum \ (\mathsf{unlabel} \ ts))) \ \parallel \ 1 \]

\[(xs \mapsto [2, -1]) \vdash let \ ts = tails \ xs \ in \ rewrite \ (maximum \ (map \ sum \ (\mathsf{unlabel} \ ts))) \ \parallel \ 1 \]

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

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

is a symbolic local example for \(?ts\) that corresponds to the sketch evaluation above and is added to \(\mathcal{E}_{\text{sym}}[?ts]\) by CollectExamples.

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

### 4.4 Synthesizing the Compression Function

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

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

Here \(n\) is the number of sketch holes and \(L_{O(1)}^{\mathsf{scalar}}\) 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 \(?ts\), including the one from Example 4.4. Then the realizability constraint \(\rho(?ts)\) for this hole is:

\[
\exists t_3 \in L_{O(1)}^{\mathsf{scalar}}, \ ?t_3 \ (xs \mapsto [2, -1], ts \mapsto ?compress \ [[2, -1], [-1], []]) = 1 \land \ldots \text{(other examples)}
\]

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, \(i.e. \?compress \mapsto \lambda x. \mathsf{maximum} \ (\mathsf{map} \ \mathsf{sum} \ x)\), this constraint becomes realizable with \(?ts\) simply returning \(ts\).

\(\text{SuFu}\) 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:

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

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 \text{Refine}. In each iteration, \(\text{SuFu}\) first instantiates the symbolic examples \(\mathcal{E}_{\text{sym}}\) with the current compression function \(\lambda x.A\) to obtain the concrete local examples \(\mathcal{E}\) (Line 3). The function \text{Refine} (Lines 8-12)
Algorithm 2: Function CompressSynthesis.

Input: Symbolic local examples $E_{sym}$.
Output: A compression function.

1. $A \leftarrow ()$;
2. while true do
3. \hspace{1em} $E \leftarrow \text{Subst}(E_{sym}.\lambda x.A)$;
4. \hspace{1em} $A^* \leftarrow \text{Refine}(E)$;
5. \hspace{1em} if $A^*$ is empty then return $\lambda x.A$;
6. \hspace{1em} $A \leftarrow A \cup A^*$;
7. end
8. Function Refine($E$):
9. \hspace{1em} foreach sketch hole $h$ do
10. \hspace{2em} $A_h \leftarrow \text{SolveSingleHole}(E[h])$;
11. \hspace{1em} end
12. return $\cup_h A_h$;

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.

<table>
<thead>
<tr>
<th>Iter.</th>
<th>Example for $?t_2$</th>
<th>Example for $?t_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>In</td>
<td>Out</td>
</tr>
<tr>
<td>sym</td>
<td>$xs \mapsto [2, -1], ts \mapsto [[-1], [1]]$</td>
<td>$xs \mapsto [2, -1], ts \mapsto [[2, -1], [-1], [1]]$</td>
</tr>
<tr>
<td>1</td>
<td>$x_1 \mapsto [2, -1], ts \mapsto ()$</td>
<td>$x_1 \mapsto [2, -1], ts \mapsto ()$</td>
</tr>
<tr>
<td>2</td>
<td>$x_1 \mapsto [2, -1], ts \mapsto (0)$</td>
<td>$x_1 \mapsto [2, -1], ts \mapsto (0)$</td>
</tr>
<tr>
<td>3</td>
<td>$x_1 \mapsto [2, -1], ts \mapsto (0, -1)$</td>
<td>$x_1 \mapsto [2, -1], ts \mapsto (0, -1)$</td>
</tr>
</tbody>
</table>

then iterates through each hole $h$, synthesizing additional attributes $A_h$ required to make this hole realizable under the current concrete examples $E[h]$; 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 $E_{sym}$).

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.
\(?t_3\), \text{\texttt{Refine}}\) will discover that the attribute \(mts\) is necessary to make this hole realizable, resulting in \(A = (mts)\) after the first iteration.

\textbf{Iteration 2.} Since the first iteration synthesized a new attribute \(mts\), we need to call \texttt{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 \(?t_5\) is ignored in this iteration because its output did not change, so we already know it is realizable. In this round \texttt{Refine} will discover a new attribute \(sum\)—the sum of list elements—for hole \(?t_2\), thus updating tuple \(A = (mts, sum)\).

\textbf{Iteration 3.} Since a new attribute \(sum\) was added, \texttt{SuFu} invokes \texttt{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, \texttt{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.

\section{4.5 Synthesizing Attributes for a Single Sketch Hole}

Algorithm 2 invokes the function \texttt{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 \(\texttt{\#compress}\) such that the examples \(\mathbb{E}[h]\) would become realizable after extending their inputs with the new attributes specified by \(\texttt{\#compress}\):

\[
\exists t_h \in L_{\text{O}(1)}^{\text{\#compress}}, \forall (\text{in, out}) \in \mathbb{E}[h], t_h \left(\text{in, \#compress' \ in}_{orig}^i\right) = \text{out}
\]

(6)

Here we write \(\text{in}_{orig}^i\) denotes the \(i\)th intermediate data structure in the original input, which has been compressed into scalar attributes in the current input \(\text{in}\).

\textbf{Example 4.7.} When \texttt{SolveSingleHole} is invoked on \(?t_2\) in the second iteration of Algorithm 2, the specification of \(\texttt{\#compress}\) is as follows (with the example in Row 2 of Tab. 3 shown explicitly):

\[
\exists t_2 \in L_{\text{O}(1)}^{\text{\#compress}}, t_2 \left((x s \mapsto [2, -1], \text{ts} \mapsto 0), \texttt{\#compress'} \ [2, -1], [\text{-1}], []\right) = (1) \land \ldots \text{(other examples)}
\]

It is challenging to solve the specification Eq. 6 because it involves the sketch hole \(?t_2\), which is too complex to be synthesized together with \(\texttt{\#compress}\). 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 \(?t_h \texttt{\ in} := \texttt{comb } (\texttt{\#extract in})\), where \texttt{\#extract} extracts a tuple of scalar values and \texttt{comb} accomplishes the calculation using the extracted values. With this decomposition, Eq. 6 can be rewritten as follows:

\[
\exists \texttt{comb, \#extract} \in L_{\text{O}(1)}^{\text{\#scalar}}, \forall (\text{in, out}) \in \mathbb{E}, \texttt{comb} \left(\texttt{\#extract} \left(\text{in, \#compress' \ in}_{orig}^i\right)\right) = \text{out}
\]

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

\textbf{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., \(\exists t \in L_{\text{O}(1)}^{\text{\#scalar}}, \forall \text{in}, f \texttt{ in} = t \texttt{ in}\).

We then use a quantifier elimination procedure to remove \(\texttt{comb}\) from the specification. Specifically, by Asm. 4.8, we can regard \(\texttt{comb}\) as an uninterpreted function and thus apply Ackermann’s reduction [Lewis 1978] to \(\texttt{comb}\). 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 \( \text{comb} \), as shown below.

\[
\exists \text{extract} \in L^{\text{scalar}} \circ \forall (in_1, out_1), (in_2, out_2) \in E,
\text{extract} \left( in_1, \text{compress}' \left( \text{in}^{\text{orig}}_{1,i} \right) \right) = \text{extract} \left( in_2, \text{compress}' \left( \text{in}^{\text{orig}}_{2,i} \right) \right) \rightarrow out_1 = out_2
\] (7)

SuFu treats Eq. 7 as a joint specification for \( \text{compress}' \) and \( \text{extract} \) and synthesizes them simultaneously by enumeration. Given a program space for \( \text{compress}' \), it enumerates all pairs of candidate programs for \( \text{compress}' \) and \( \text{extract} \) 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 \( \text{compress}' \) and \( \text{extract} \) are both small: \( \text{compress}' \) can be constructed compactly from library functions since its efficiency is not important, while \( \text{extract} \) 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]. \( \square \)

**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 \( \text{compress}' \) is expressive enough, SuFu can always find an appropriate \( \text{compress}' \), 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 \( \text{compress}' \) 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 \( \text{cost}(p, \text{in}) \) be the size of the derivation tree of evaluating program \( p \) on input \( \text{in} \). For any sketching problem, the following formula is always satisfied by the reference program \( p \) and the program \( p' \) synthesized by SuFu.

\[
\exists c > 0, \forall \text{in}, \text{cost}(p', \text{in}) \leq c \cdot \text{cost}(p, \text{in})
\]

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.
Fig. 5. D&C program restructuring as superfusion. The input to SuFu includes the template \( \text{dac\_id} \), which is the same for all D&C restructuring tasks, and a reference program \( \text{dac\_mts} \) that composes the template with the original reference program \( \text{mts} \). 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, \( \text{mts} \). To combine the \( \text{mts} \) of the two halves, this program introduces the sum of list elements as an auxiliary output. The intuition here is that the \( \text{mts} \) of the whole list is either the \( \text{mts} \) of the right half or the sum of elements in the right half plus the \( \text{mts} \) 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 \( \text{dac\_id} \), shown in Fig. 5b. A careful reader will see that \( \text{dac\_id} \) 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 \( \text{mts} \) from Fig. 1a to obtain a new program \( \text{dac\_mts} \), shown in Fig. 5b, which behaves equivalently to \( \text{mts} \). In this new program, the template \( \text{dac\_id} \) produces an intermediate data structure, which we can annotate with \( \text{Packed} \), and apply SuFu to eliminate it. This will have the effect of forcing SuFu to use the recursive structure of \( \text{dac\_id} \)—that is, divide-and-conquer—to compute the result of \( \text{mts} \). In fact, the result of applying SuFu to \( \text{dac\_mts} \) 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 \( \text{Packed} \) 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., \( \mathcal{L}_{\text{scal}}^{O(1)} \)) is based on the SyGuS theory of conditional linear integer arithmetic [Alur et al. 2017a], which includes the \text{if-then-else} operator, arithmetic operators such as +, relational operators such as \( \leq \), and Boolean operators such as \text{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. \( \text{match } \) with \( \text{Nil} \to \text{?} \mid \text{Cons}(\text{h}, \text{t}) \to \text{?} \) 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.
- **AUTO LIFTER** [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.

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

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.

<table>
<thead>
<tr>
<th>Source</th>
<th>#Solved</th>
<th>Time Cost</th>
<th>Result Size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Gen</td>
<td>Syn</td>
</tr>
<tr>
<td>Fusion</td>
<td>14/16</td>
<td>0.012</td>
<td>9.1</td>
</tr>
<tr>
<td>Recursion</td>
<td>170/178</td>
<td>0.015</td>
<td>7.5</td>
</tr>
<tr>
<td>D&amp;C</td>
<td>80/96</td>
<td>0.022</td>
<td>11.6</td>
</tr>
<tr>
<td>Total</td>
<td>264/290</td>
<td>0.017</td>
<td>8.8</td>
</tr>
</tbody>
</table>

- **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.

<table>
<thead>
<tr>
<th>Task Description</th>
<th>Program Fragment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synthesize a D&amp;C program that calculates the maximum tail product of a (possibly negative) integer list.</td>
<td><code>( if r.1 == r.3 then 1.1 * r.1</code>&lt;br&gt;<code>else if 1.2 * r.3 &gt; r.1 then 1.2 * r.3 else r.1,</code>&lt;br&gt;<code>if r.3 &lt; 0 then 1.1 * r.2</code>&lt;br&gt;<code>else if 1.2 * r.3 &lt; r.2 then 1.2 * r.3 else r.2,</code>&lt;br&gt;<code>1.3 * r.3 )</code></td>
</tr>
<tr>
<td>Given two languages $\mathcal{L}_1$ and $\mathcal{L}_2$ of Boolean expressions, synthesize the interpreter of $\mathcal{L}_1$ from the interpreter of $\mathcal{L}_2$.</td>
<td>`</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Source</th>
<th>Tool</th>
<th>#Solved</th>
<th>Time</th>
<th>Source</th>
<th>Tool</th>
<th>#Solved</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recursion</td>
<td>SuFu</td>
<td>170</td>
<td>11.4</td>
<td>D&amp;C</td>
<td>SuFu</td>
<td>80</td>
<td>46.8</td>
</tr>
<tr>
<td></td>
<td>SYNDUCE</td>
<td>125</td>
<td>1.7</td>
<td></td>
<td>AUTOlIFTER</td>
<td>82</td>
<td>15.6</td>
</tr>
</tbody>
</table>

ghost function $\texttt{compress}$ is much smaller than the solutions to sketch holes, and hence the extra work of synthesizing $\texttt{compress}$ (as well as $\texttt{extract}$) 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 surprising 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.

<table>
<thead>
<tr>
<th>Source</th>
<th>Approach</th>
<th>#Solved</th>
<th>Time</th>
<th>Source</th>
<th>Approach</th>
<th>#Solved</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fusion</td>
<td>SuFu</td>
<td>14</td>
<td>1.1</td>
<td>Recursion</td>
<td>SuFu</td>
<td>170</td>
<td>11.4</td>
</tr>
<tr>
<td></td>
<td>Grisette</td>
<td>4</td>
<td>0.7</td>
<td></td>
<td>Grisette</td>
<td>60</td>
<td>20.0</td>
</tr>
<tr>
<td>D&amp;C</td>
<td>SuFu</td>
<td>80</td>
<td>11.5</td>
<td>Total</td>
<td>SuFu</td>
<td>264</td>
<td>11.1</td>
</tr>
<tr>
<td></td>
<td>Grisette</td>
<td>21</td>
<td>58.1</td>
<td></td>
<td>Grisette</td>
<td>85</td>
<td>28.9</td>
</tr>
</tbody>
</table>

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 than 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 \( \texttt{compress} \). Second, on 4 tasks, SuFu succeeds in finding the expected \( \texttt{compress} \), 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 \( \texttt{compress} \) whose sketch holes are more complex than necessary, making the PBE solver times out. Note that although our specification for \( \texttt{compress} \) (Eq. 5) ensures the existence of valid sketch holes, their size may vary significantly regarding the choice of \( \texttt{compress} \). For example, one reference program in our dataset is \( \texttt{sqrsum (upto n)} \), where \( \texttt{upto} \) constructs a Packed list of numbers from 1 to \( n \), and \( \texttt{sqrsum} \) sums their squares. On this task, SuFu synthesizes an empty \( \texttt{compress} \) 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 Bodik 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. \( \forall xs, \sum xs = (\text{tails'} xs).2 \). 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 Bodik 2023; Lubin et al. 2020; Porncharoenwase et al. 2022; Solar-Lezama et al. 2006; Torlak and Bodik 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


---

\(^5\)https://github.com/jiry17/SuFu


tails :: List -> Packed NList
map :: (List -> Int) -> NList -> Packed List
map f Nil = Nil
map f Cons(h, t) = Cons(f h, map f t)
mts xs = maximum (map sum (tails xs))

(a) The reference program.

(b) The generated sketch.

Fig. 7. An example of eliminating multiple data structures at once. (a) A reference program of mts with two annotated data structures. (b) The sketch generated by SuFu, where [i] following each label and Packed shows the kind label of that constructor. Here we omit the details in tails for simplicity.

APPENDIX OVERVIEW

Our appendix is organized as follows. First, Appendix A includes two running examples of SuFu: one for eliminating multiple intermediate data structures at once (Appendix A.1), and the other for applying SuFu to restructure data structure traversals (Appendix A.2). Then, Appendix B supplies details on the bounded verifier in our implementation, and at last, Appendix C proves the properties of SuFu discussed in Sec. 4.6.

A APPENDIX: MORE EXAMPLES

A.1 Eliminating Multiple Intermediate Data Structures

In Sec. 4, we have seen how SuFu handles the special case where Packed is used only once. Now, we demonstrate how SuFu extends to the general case with multiple Packed annotations using a running example, shown in Fig. 7. The reference program (Fig. 7a) here is almost the same as the previous mts example, except that the output of map (i.e., the list of tail sums) is also annotated.

**Sketch generation.** Fig. 7b shows the sketch program generated by SuFu. Besides generating term-level annotations (label, unlabel, and rewrite), SuFu further infers kind labels for each Packed and label, shown as [i] in Fig. 7b. These labels represent the kind of intermediate data structures denoted by each Packed and constructed by each label. In this example, SuFu infers the same kind label for every Packed and label in map because all of them are related to the intermediate list of tail sums, and SuFu infers a different kind label for the Packed in tails because it denotes a different data structure, the intermediate list of all tails.

SuFu infers kind labels from types. Specifically, given a program in λsk, SuFu first attaches a symbolic kind label to each Packed and label, and then runs a round of type-checking with these symbolic labels. In this process, SuFu regards Packed types with different kind labels as different types and takes label[x] as the specific constructor for type Packed[x]. Hence, the type-checking will raise a series of equivalent constraints between kind labels, and SuFu will assign the same value to each equivalent class. For example, suppose the kind label of the Packed in map is x, and the kind label of the label in hole ?t1 is y. Then, the output type of map will be Packed[x] List, and the function body of map f Nil will have a type of Packed[y] List. These two types must be the same by the typing rule of function definitions, thus inducing an equivalent constraint of x = y.

**Example collection.** SuFu still collects local examples from the execution of the sketch program, but it will also record the kind label for each intermediate data structure to track its source. For
Tree = Leaf(Int) | Node(Tree, Tree)

\(\text{rec Leaf}(v) = (\max v \ 0, v)\)
\(\text{rec Node}(l, r) = (\text{rec } l) \ (\text{rec } r)\)

(a) The original restructuring task.

mits t = mts (flatten t)

\(\text{rec } \text{Leaf}(v) = ?f1\ v\)
\(\text{rec } \text{Node}(l, r) = ?f2\ (\text{rec } l) \ (\text{rec } r)\)

(b) The expected result.

res t = mts (rec t)

(c) The input of SuFu.

Fig. 8. An example for restructuring data structure traversals. (a) The input of the restructuring task, where \(\text{mits}\) is the reference program, and \(\text{rec}\) is the user-provided sketch. (b) The expected result of program restructuring, which introduces the sum of leaf values as an auxiliary output. (c) The corresponding input of SuFu for solving this restructuring task.

example, from the execution of \(\text{mts}\[2]\), SuFu can collect the local example below for hole \(?t_2\).

environment: \((h \mapsto [2], ts \mapsto \text{label}[2] [0])\) value: \text{label}[2] [2, 0]

Here both ts and the output value have the kind label [2] because they are constructed by the \text{label}[2] in map. Similarly, SuFu can collect the local example below for \(?t_3\) from the same execution \(\text{mts}\[2]\), where ts has the kind label [1] because it is constructed by some \text{label}[1] in tails.

environment: \((xs \mapsto [2], ts \mapsto \text{label}[1] [[2], [2]])\) value: 2

To obtain symbolic local examples, SuFu introduces two compression functions \(\text{compress}_1\) and \(\text{compress}_2\), each for one kind of intermediate data structures, as they can be replaced with different scalar attributes. Correspondingly, SuFu replaces every \text{label} with the compression function of its kind label, for example, the above two examples will result in the symbolic examples below.

\[\langle \text{in}_1, \text{out}_1 \rangle = \left(\left(h \mapsto [2], ts \mapsto \text{compress}_2 [0]\right), \text{compress}_2 [2, 0]\right)\]
\[\langle \text{in}_2, \text{out}_2 \rangle = \left(\left(xs \mapsto [2], ts \mapsto \text{compress}_1 [[2], [2]]\right), 2\right)\]

**Compression synthesis.** Our synthesis algorithm for a single compression function (Algorithm 2) can be straightforwardly extended to synthesize both \(\text{compress}_1\) and \(\text{compress}_2\), as shown below.

1. Algorithm 2 can be extended to iterate with two sets of scalar attributes, respectively corresponding to the two compression functions.
2. The specification solved by \text{Refine} (Eq. 6) can be extended as follows to synthesize new attributes for both \(\text{compress}_1\) and \(\text{compress}_2\).

\[\exists t_h \in \mathcal{L}_{\text{L scalar}}^{O(1)}, \forall (in, out) \in \mathbb{E}[h], \exists t_h (in, \text{compress}_1^{\text{orig}}, \text{compress}_2^{\text{orig}}) = out\]

Here, unknown functions \(\text{compress}_1^{\text{orig}}\) and \(\text{compress}_2^{\text{orig}}\) are introduced to respectively specify new attributes for the two compression functions, and the intermediate data structures in the original input are divided into two sets, \(in_{1, i}^{\text{orig}}\) and \(in_{2, i}^{\text{orig}}\), according to their kind labels.

3. The transformation for eliminating \(t_h\) still works, and the final enumeration approach can be extended to this case by enumerating \(\text{extract}, \text{compress}_1^{\text{orig}}\), and \(\text{compress}_2^{\text{orig}}\) at the same time.

### A.2 Program Restructuring for Data Structure Traversals

One baseline solver in our evaluation is SYNDUCE [Farzan et al. 2022], which restructures recursive data-structure traversals according to a user-provided sketch that traverses the input data structure.
at most once. This kind of program restructuring tasks can be reduced to fusion and thus can be solved by SuFu. Fig. 8 shows an example of this reduction.

- In the original task (Fig. 8a), the reference program \( m \) is a composition of two traversals: \( \text{flatten} \) traverses the input tree and collects all leaf values into a list in order, and \( n \) then returns the maximum tail sum of the value list. The goal of this task is to restructure \( m \) using the given template \( \text{rec} \), which traverses the input tree only once and directly calculates the result during the traversal, via unknown functions \( ?f_1 \) and \( ?f_2 \).
- Fig. 8b shows the expected result of this restructuring task, which calculates the sum of leaf values as an auxiliary output for calculating \( m \) in a single traversal.

As mentioned in Sec. 5, SuFu can be applied to a program restructuring task by (1) implementing an identity function in the target form as the template function, (2) composing the template function with the reference program to be restructured, and (3) invoking SuFu to eliminate the intermediate data structure generated by the template function. Fig. 8c shows the input of SuFu for this task: \( \text{rec} \) is the template function that implements the identity function as a traversal, and \( \text{res} \) composes this template with the reference program \( m \). By invoking SuFu to eliminate the intermediate tree produced by \( \text{rec} \), we can get the same program as the expected result in Fig. 8b.

### B APPENDIX: VERIFICATION

SuFu follows the CEGIS framework and thus requires a verifier to generate counter-examples for incorrect results. We use bounded verification in our implementation. Specifically, we limit the size of inductive data structures to no larger than 10 and limit the range of integers to \([-3, 3]\). Given a candidate program, our verifier will evaluate this program and the reference program on every input within these limits and check whether their outputs are the same. If not, the corresponding input will be returned as a counter-example.

Besides, to reduce the time cost of verification, SuFu initializes the input set of CEGIS with \(10^3\) random inputs (sampled from the same range as verification). These random inputs can exclude most incorrect results and thus can significantly reduce the number of necessary CEGIS iterations.

### C APPENDIX: PROOFS

**Theorem C.1 (Thm. 4.9).** 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.** A sound PBE solver ensures that the solution to every hole satisfies the concrete examples corresponding to the synthesized \( \text{compress} \). Hence, to prove the soundness of SuFu, we only need to prove that any \( \text{compress} \) and sketch solution that satisfy all local examples always induce a correct synthesized program. We achieve this by proving the claim below.

- For any sketch solution \( (\overline{T}_i, \overline{th}) \) and compression function \( \text{compress} \), an evaluation judgment \( E \vdash t \Downarrow v \) still holds if all intermediate data structures in environment \( E \) and value \( v \) are replaced with \( \text{compress} \), term \( t \) is rewritten by the sketch solution, and SuFu and the sketch solution satisfy all local examples collected from \( E \vdash t \Downarrow v \). Or more formally:

\[
E \vdash t \Downarrow v \implies (\text{sub} \text{\ compress} E) \vdash (\text{sub} (\overline{T}_i, \overline{th}) t) \Downarrow (\text{sub} \text{\ compress} v)
\]

where \( \text{compress} \) and \( (\overline{T}_i, \overline{th}) \) are a compression function and a sketch solution satisfying all local examples collected from \( E \vdash t \Downarrow v \), and function \text{sub} replaces \text{label} with \( \text{compress} \), Packed types with \( \overline{T}_i \), and \text{rewrite} terms with \( \overline{th} \).

If this claim is correct, we can obtain the correctness of the synthesized program by taking \( E \) as empty, taking \( t \) as \( p \) in for every \( in \in I \), and taking \( v \) as the corresponding evaluation result.
These judgments form the premise of applying rule E-Let. In this case, the target claim becomes as follows.
\[
\left( E \vdash \text{let}(x, t, t') \Downarrow v \right) \land \left( E \vdash t \Downarrow v_x \right) \land \left( E, x \mapsto v_x \vdash t' \Downarrow v \right) \implies \\
\left( \text{sub} \ x \mapsto \text{compress} \ E \right) \vdash \text{let}(x, \{\text{compress \ } t\}, \{\text{compress \ } t'\}) \Downarrow \left( \text{sub} \ \text{compress} \ v \right)
\]

Using the latter two premises, the induction hypothesis, and the definition of sub on environments, we obtain the following two evaluation judgments.
\[
\left( \text{sub} \ \text{compress} \ E \right) \vdash \left( \text{sub} \ \{\text{compress} \ h\} \ t \right) \Downarrow \left( \text{sub} \ \text{compress} \ v_x \right)
\]
\[
\left( \text{sub} \ \text{compress} \ E, x \mapsto \text{sub} \ \text{compress} \ v_x \right) \vdash \left( \text{sub} \ \{\text{compress} \ h\} \ t' \right) \Downarrow \left( \text{sub} \ \text{compress} \ v \right)
\]

These judgments form the premise of applying rule E-Let, and by applying this rule, we obtain the consequence of the target claim in this case, as shown below.
\[
\left( \text{sub} \ \text{compress} \ E \right) \vdash \text{let}(x, \{\text{compress} \ h\} \ t, \{\text{compress} \ h\} \ t') \Downarrow \left( \text{sub} \ \text{compress} \ v \right)
\]

Case 3: the last evaluation rule is E-Rewrite. In this case, the target claim becomes as follows, where h denotes the index of the current rewrite term.
\[
E \vdash \text{rewrite}(t) \Downarrow v \implies \left( \text{sub} \ \text{compress} \ E \right) \vdash t_h \Downarrow \left( \text{sub} \ \text{compress} \ v \right)
\]

Recall that SuFu collects local examples from the evaluation judgments of rewrite terms. Here, SuFu will collect environment E and value v as a local example, whose symbolic form is as follows.
\[
\text{?} t_h \left( \text{sub} \ \text{compress} \ E \right) = \left( \text{sub} \ \text{compress} \ v \right)
\]

Hence, the target claim in this case is implied by the soundness of the PBE solver.

\[\square\]

**Theorem C.2 (Thm. 4.10).** 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 \text{compress} can implement any function mapping from intermediate data structures to scalar values.

**Proof.** A complete PBE solver can always find sketch holes from concrete examples (if realizable), hence SuFu can always find a sketch solution if it can find a valid compression function \text{compress}. Moreover, since the synthesis approach of \text{compress} (Algorithm 2) returns only when the result is known to be valid, we only need to prove the terminality of this algorithm.

We achieve this by proving the following claim.

Proc. ACM Program. Lang., Vol. 8, No. PLDI, Article 185. Publication date: June 2024.
• For any final set of symbolic local examples \( \mathbb{E}_{sym} \), there always exists a function \( \text{compress}^* \) such that for any candidate compression function \( \text{compress} \), \( \text{compress}^* \) is always a valid result for \( \text{Refine}(\text{Subst}(\mathbb{E}_{sym}, \text{compress})) \) (Lines 3-4 in Algorithm 2).

Let us first show how this claim implies the terminality of Algorithm 2. Suppose such a \( \text{compress}^* \) exists but Algorithm 2 does not terminate. Then, there must be infinite invocations of \( \text{Refine} \).

• On the one hand, the results of these invocations must be pairwise different because, in each invocation, those attributes found previously have been merged into the input, hence they will never be returned again by the bottom-up enumeration.

• On the other hand, the results of these invocations cannot be larger than \( \text{compress}^* \) because the bottom-up enumeration will return \( \text{compress}^* \) as the result once visiting it.

However, the number of programs no larger than \( \text{compress}^* \) must be finite, so there cannot be an infinite number of different results, raising a conflict.

Now the task remaining is to prove the claim, i.e. construct \( \text{compress}^* \) for a finite set of local examples. Recall that the specification of \( \text{'compress'} \) (the result of \( \text{Refine} \)) is to make sketch holes possible to satisfy concrete examples, and there are two restrictions on the sketch holes.

1. The time complexity of the sketch hole should be \( O(1) \).

2. The sketch hole should have a scalar type, which induces a requirement that the output type of the compression function should be scalar.

If ignore both restrictions, the identity function \( \text{id in} = \text{in} \) must be a valid compression function. Note that the output of local examples includes only (1) the scalar outputs of the original \( \text{rewrite} \) term, and (2) some known attributes (specified by a known function \( \text{compress} \)) of intermediate data structures constructed by the original term. When the compression function is \( \text{id} \), the input of local examples will include the whole input of the original term, hence we can construct the sketch hole by (1) using the original \( \text{rewrite} \) term to produce the original outputs, and (2) using \( \text{compress} \) to calculate the attributes for each intermediate data structure.

Moreover, \( \text{id} \) is still valid even if we bring back the restriction on the time complexity. This is because the sketch hole is only constrained by a finite set of local examples, hence we can always remove the recursions/loops in the sketch hole by unrolling them enough times, thus generating a constant-time program that has the same IO behavior on the given set of local examples.

Finally, when both restrictions are present, we tune the output type of \( \text{id} \) to scalar using the example set again. Specifically, in Algorithm 2, the compression function will only be evaluated on intermediate data structures in the local examples. Therefore, a truncation function \( \text{trunc}_d \) that returns only the first \( d \) levels of the input will have the same effect as the identity function, where \( d \) is the maximum depth of the intermediate data structures in the examples. The output of \( \text{trunc}_d \) includes only a constant number of values and thus can be trivially encoded into a scalar type. \( \square \)

**Theorem C.3 (Thm. 4.11).** Let \( \text{cost}(p, \text{in}) \) be the size of the derivation tree of evaluating program \( p \) on input \( \text{in} \). For any sketching problem, the following formula is always satisfied by the reference program \( p \) and the program \( p' \) synthesized by SuFu.

\[
\exists c > 0, \forall \text{in}, \text{cost}(p', \text{in}) \leq c \cdot \text{cost}(p, \text{in}) \quad (8)
\]

**Proof.** Let \( (\overline{T_i}, \overline{t_h}) \) be the sketch solution found by SuFu. Since SuFu ensures \( \overline{t_h} \) to be constant-time, we can take constant \( c \) as a large enough integer such that the derivation tree of every \( t_h \) will never be larger than \( c \). The task remaining is to prove that Eq. 8 holds for this constant.

For any input \( \text{in} \), let \( T \) and \( T' \) be the derivation tree of evaluating \( p \) and \( p' \) on \( \text{in} \), respectively. Then, \( T' \) can always be transformed from \( T \) by replacing all subtrees related to \( \text{rewrite} \) terms with derivation trees of evaluating some \( t_h \). Therefore, the target theorem is implied by the inequality
below, where $m$ denotes the number of subtrees in $T$ that are related to rewrite terms.

$$cost(p', \text{ in}) \leq c \cdot m + (cost(p, \text{ in}) - m) \leq c \cdot m + c \cdot (cost(p, \text{ in}) - m) = c \cdot cost(p, \text{ in})$$

\[ \square \]