Superfusion: Eliminating Intermediate Data Structures via Inductive Synthesis

ANONYMOUS AUTHOR(S)

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 is that all holes in the sketch have to be solved jointly, 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 synthesis 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, outperforming rewriting-based fusion systems and achieving comparable performance with specialized approaches to program restructuring on their respective domains.

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 the intermediate data structures.

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

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

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

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

The inefficiency of compositional programs can often be addressed by eliminating intermediate data structures, replacing them with *scalar attributes* that are sufficient to compute the final result. For example, in the `mts` program, we can replace the list of all tails with a pair of attributes: (1) the maximum tail sum, and (2) the sum of the whole list. The intuition is that the `mts` of a non-empty list is either the `mts` of its tail, or the sum of the tail and the head. Using this observation, we can rewrite `mts` into an efficient program `mts'`, shown on

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

2024. XXXX-XXXX/2024/2-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

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

The input to superfusion is a reference program annotated with data structures to be eliminated; *e.g.* the nested list returned by `tails` in the `mts` example. The first step is to turn the reference program into a *sketch* [Solar-Lezama et al. 2006], where any program fragment that consumes or produces the undesirable intermediate data structure is replaced with a *hole*. The second step is to solve the sketch, filling the holes with new expressions that operate only on scalar attributes, while ensuring that the input-output behavior of the whole program is unchanged.[1]

Challenge 1: Scalability. The main technical challenge of superfusion is the scale of the search space. Even if the solution to each single hole has a manageable size, the issue is that all holes have to be solved jointly, since our specification—equivalence to the reference program—is *global*; this makes a direct application of existing inductive synthesizers infeasible.

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

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

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

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 compressed data structure. This appears to be a chicken-and-egg

[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.

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 *quantifier elimination* (eliminating second-order existential quantifiers with the definition of functions), allowing us to synthesize *?compress* independently.

Evaluation. We implement superfusion in a tool called SuFu and evaluate it on a suite of 290 benchmarks collected from prior work. Our first source of benchmarks are fusion tasks from the deductive fusion literature [Bird 1989; Bird and de Moor 1997; Gill et al. 1993; Hu et al. 1997; Wadler 1988]. For our second source of benchmarks, we turns to prior work on *program restructuring* [Acar et al. 2005; Farzan et al. 2022; Farzan and Nicolet 2017, 2021a,b; Ji et al. 2023; 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, which is significantly more than the state-of-the-art deductive fusion technique [Hinze et al. 2010]. Moreover, while being general, SuFu is also competitive with two specialized synthesizers for program restructuring [Farzan et al. 2022; Ji et al. 2023] 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.

- *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 type information.
- *A sketch solving method* (Sec. 4) that efficiently decomposes the global sketch problem into local synthesis problems 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.

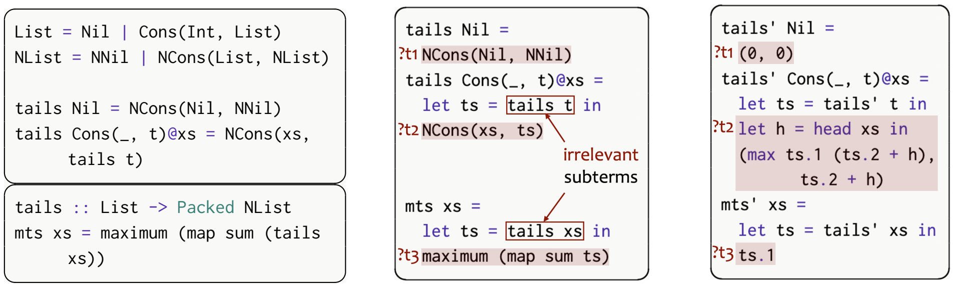
2 OVERVIEW

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 generated by `tails`. To improve efficiency, 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 as $?t_1$, $?t_2$, and $?t_3$ in Fig. 1b: $?t_1$ and $?t_2$ produce an `NList`, while $?t_2$ and $?t_3$ consume an `NList`. Each of the 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 an `NList`.

To make the synthesizer's job easier, SuFu attempts to reuse as much of the original program as possible, moving subterms that don't 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.



```
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.

```
tails Nil =
?t1 NCons(Nil, NNil)
tails Cons(_, t)@xs =
  let ts = tails t in
?t2 NCons(xs, ts)


mts xs =
  let ts = tails xs in
?t3 maximum (map sum ts)
```

(b) Sketch generation.

```
tails' Nil =
?t1 (0, 0)
tails' Cons(_, t)@xs =
  let ts = tails' t in
?t2 let h = head xs in
    (max ts.1 (ts.2 + h),
        ts.2 + h)
mts' xs =
  let ts = tails' xs in
?t3 ts.1
```

(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.

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 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. 4.12).

2.2 Sketch Solving with Compression Functions

Superfusion sketches are challenging to solve due to the combination of two factors. First, our correctness specification is global, so the holes cannot be solved independently. Second, the solutions for individual holes are far from trivial in size: in our dataset, the average number of AST nodes in a hole solution is 46.7 and the maximum is as large as 559. As a result, we cannot use generic sketch solvers [Solar-Lezama et al. 2006; Torlak and Bodík 2013], which can handle global specifications, but do not scale to solution sizes we are interested in: for example, a state-of-the-art sketch solver [Lu and Bodík 2023] can only solve around 30% of tasks in our dataset.

To overcome the scalability challenge, we need to *decompose* the global correctness specification into *local* input-output (IO) examples for each hole. With local examples at hand, we can then use off-the-shelf PBE synthesizers for recursion-free programs [Alur et al. 2017b; Ji et al. 2021] to independently solve each hole.

To generate the local specifications we leverage the reference program. Specifically, we start by observing the IO behavior of the original subterms that were replaced by holes. For example, Tab. 1 illustrates the behavior of the three subterms of the original mts program, corresponding to the holes $?t_1$, $?t_2$, and $?t_3$, when executed on the input [2]. Of course, these IO examples cannot 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.

Table 1. Input-output examples collected by tracing `mts [2]`. From left to right: the name of the hole and the variables it has in scope; the original term for this hole; input-output behavior of the original term (with intermediate data structures marked in blue); symbolic examples obtained by "compressing" intermediate data structures; concrete examples obtained using the definition of *?compress* in Eq. 1.

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

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

As we discussed in the introduction, one valid compression function for `mts` maps a list of tails to the maximum tail sum and the sum of all elements, as shown below:

$$?compress\ ts := \big(maximum\ (map\ sum\ ts), sum\ (head\ ts)\big) \tag{1}$$

Once the definition of *?compress* has been fixed, it can be substituted into the symbolic examples to obtain *concrete* IO examples for the holes, as shown in the last column of Tab. 1. With these examples—given enough inputs to `mts`—an off-the-shelf PBE solver can efficiently synthesize the correct solution to each hole, shown in Fig. 1c.

The remaining challenge then is to synthesize a suitable compression function; the rest of this section is devoted to this task.

2.3 Synthesizing the Compression Function

The only specification we have for *?compress* are the symbolic examples, such as those in Tab. 1. The issue with this specification, of course, is that it also refers to the unknown hole programs, $?t_1$, $?t_2$, and $?t_3$: in other words, it's a *relational synthesis* problem [Wang et al. 2018]. The naive way to approach this problem is to synthesize *?compress* and all sketch holes simultaneously, which defeats the purpose of introducing *?compress* in the first place.

Another problem of synthesizing *?compress* is that the size of *?compress* depend on the number of attributes and may be large, which still imposes a challenge even if we have a local specification that involves only *?compress*.

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 can be further decomposed into small components, which can be classified into the two categories.

(1) components with a *small implementation*, which can be efficiently enumerated;
(2) components that can be quantified over and eliminated from the specification.

For *?compress*, while the whole *?compress* may be large, the component for generating each scalar attribute is small and can be efficiently enumerated. For the sketch holes, while they cannot always be directly eliminated from the specification, they can be further decomposed into components, where some can be eliminated and some can be efficiently enumerated with *?compress*.

This observation leads to the following synthesis algorithm:

- The top-level algorithm is an iterative *fixpoint computation*, where each iteration adds a new scalar attribute to *?compress* to satisfy symbolic examples for a single hole.
- In each iteration, the algorithm uses *quantifier elimination* to obtain an independent specification involving only the current attribute of *?compress* plus at most a small component from a sketch hole, allowing efficient enumeration.

Next, we discuss the fixpoint computation and the quantifier elimination step in more detail.

2.3.1 Fixpoint Computation of Scalar Attributes. Recall the compression function for `mts` in Eq. 1, which computes two scalar attributes from a list of tails—the maximum tail sum and the sum of all elements. Although the computation of each individual attribute is simple, the composition of multiple attributes can get quite large. Luckily, we do not need to synthesize all attributes at once, but instead can add them one by one, using a fixpoint computation inspired by fixpoint solvers in predicate abstraction [Bjørner et al. 2013; Graf and Saïdi 1997; Rondon et al. 2008].

Let us walk through the iterative synthesis of *?compress* for `mts`; in the following, we assume that SuFu is working with multiple traces of `mts`, including `mts` $[2]$ (shown in Tab. 1) and `mts` $[2, -1]$.

Iteration 1. We start with the empty compression function that does not extract any attributes, *i.e.* *?compress ts* $:= ()$. Substituting this definition into the symbolic examples, we find that the constraints for hole $?t_3$ are *unsatisfiable*:

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

Here $\mathcal{L}_{hole}$ is the program space of sketch holes and the two conjuncts originate from the two traces 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 scalar attribute to *?compress* that provides enough information to differentiate between the two traces.

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

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

Our quantifier elimination procedure, detailed below, is able to eliminate $?t_3$ from this specification, and assuming enough `mts` traces are available, will discover the correct definition for $?compress_1$:

$$?compress_1\ ts := maximum\ (map\ sum\ ts)$$

Iteration 2. Since a new attribute has been added in the previous iteration, we need to check whether it made any other sketch holes unsatisfiable. Indeed, substituting our new definition $?compress := ?compress_1$ into the symbolic examples we get the following for hole $?t_2$:

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

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

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

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

Once again, given enough traces, quantifier elimination will discover that the additional attribute required to perform the $?t_2$ computation in constant time is the sum of the list elements:

$$?compress_2\ ts := sum\ (head\ ts)$$

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

Candidate Program	Trace	Example *ts*	Example *out*	Conflicting concrete examples
length of the list of tails $?compress_1\ ts \coloneqq length\ ts$	`mts [2]`	$[[2], []]$	2	$?t_3\ 2 = 2$
	`mts [1]`	$[[1], []]$	1	$?t_3\ 2 = 1$
sum of elements $?compress_1\ ts \coloneqq sum\ (head\ ts)$	`mts [2]`	$[[2], []]$	2	$?t_3\ 2 = 2$
	`mts [−1,3]`	$[[-1,3],[3],[]]$	3	$?t_3\ 2 = 3$
maximum of elements $?compress_1\ ts \coloneqq maximum\ (head\ ts)$	`mts [2]`	$[[2], []]$	2	$?t_3\ 2 = 2$
	`mts [2,−1]`	$[[2,-1],[-1],[]]$	1	$?t_3\ 2 = 1$

Iteration 3. In the third iteration, once we substitute $?compress\ ts \coloneqq (?compress_1\ ts, ?compress_2\ ts)$ into the symbolic examples, we find that all sketch holes are satisfiable, which concludes the synthesis of $?compress$.

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

Iteration 1. Let us consider the specification for $?compress_1$ in Eq. 2. Recall that this is a relational synthesis problem, which we propose to solve via a combination of enumeration and quantifier elimination. Specifically, we can eliminate $?t_3$ from this specification by observing that (1) in order to synthesize $?compress_1$ we do not need to know the definition of $?t_3$, only that such a function exists, and (2) *a function exists iff it maps every input to a unique output.* Using the second property, we transform Eq. 2 into the following equivalent specification:

$$\forall (ts, out), (ts', out') \in E_3. \quad (?compress_1\ ts = ?compress_1\ ts') \to (out = out') \tag{4}$$

where E_3 is the set of original IO behaviors collected for $?t_3$ from the reference implementation (such as the last row, the Example column of Tab. 1).

We can now enumerate candidate solutions for $?compress_1$ from smaller to larger, checking them against the specification Eq. 4. At first glance, this specification seems weak, but in fact, SuFu can efficiently find the desired solution $?compress_1\ ts \coloneqq maximum\ (map\ sum\ ts)$ using only 21 examples. Tab. 2 illustrates three incorrect candidates for $?compress_1$, and for each one gives a pair of IO examples that is sufficient to reject it.

A careful reader might object that Eq. 4 is only equivalent to Eq. 2 when the space of $?t_3$ functions is unrestricted, whereas in our case $?t_3$ is restricted to $\mathcal{L}_{hole}$, the space of $O(1)$-time programs. This is, however, not a problem in practice because $?t_3$'s inputs and outputs are both scalars, and it is reasonable to assume that any function that operates on scalars can be implemented in constant time. More formally, SuFu makes an **assumption** on the search space $\mathcal{L}_{hole}$ that it is expressive enough to implement any needed function whose input and output are both scalar values.

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

Specifically, because $?t_2$ is a $O(1)$-time program, it can only access a constant number of scalar values in the input. Therefore, without loss of generality, we can assume that $?t_2$ has the form:

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

where *?extract* extracts a tuple of scalar values from the input variables and *?comb* combines them into the final result[2]. With this decomposition in place, we can now use our previous technique to *eliminate* *?comb* (which operates on scalars and hence is unrestricted), and then *enumerate* *?extract* jointly with $?compress_2$ (note that *?extract* is small because it does not perform any actual computation). The enumeration yields:

$$\mathit{extract}\ (xs, ts) \coloneqq (ts.1, ts.2, \mathit{head}\ xs) \qquad ?\mathit{compress}_2\ ts \coloneqq \mathit{sum}\ (\mathit{head}\ ts)$$

3 SKETCH GENERATION

Given a reference program, the first step of SuFu is to identify the terms that produce or consume intermediate data structures and replace them with sketch holes. To ensure that the resulting sketch can be solved successfully, we require the holes to satisfy the following three conditions.

(1) **Correctness**: these holes should cover all direct uses of intermediate data structures, including their construction and reading of their content. In the resulting program, these uses of intermediate data structures must be rewritten with scalar attributes.
(2) **Feasibility**: these holes should cover all data structures constructed from the intermediate data structures because it is usually impossible to reconstruct these derived data structures using only scalar attributes. For example, in the `mts` program (Fig. 1a), the list of tail sums produced by `map sum` cannot be reconstructed after the list of tails is replaced with scalar values. Therefore, `map sum` should be covered by a sketch hole, as shown in Fig. 1b.
(3) **Minimality**: these holes should be as small as possible because the larger they are, the more SuFu will need to synthesize, and thus the more difficult the synthesis task will be. For example, we should not replace the whole body of `mts` (Fig. 1a) with a hole because the resulting sketch will have no recursion. Since our sketch solver only synthesizes recursion-free terms, the resulting synthesis problem will be unsolvable.

We design our sketch generation method to ensure these three conditions. Specifically, we design an *annotation language* to capture the conditions of correctness and feasibility (Sec. 3.1). In this language, annotated holes are guaranteed to be correct and feasible whenever the program is well-typed. Hence, the problem of sketch generation is reduced to an optimization problem of finding well-typed and minimal annotations for the reference program. This optimization problem can be solved with a Max-SAT solver (Sec. 3.2).

3.1 Annotation Language

```
T  ::=  Packed T
     |  (other types)
t  ::=  label t
     |  unlabel t
     |  rewrite t
     |  (other terms)
```

Given a surface functional programming language for describing the input program, our annotation language augments it with four constructs, as shown on the right. At the type level, `Packed` is used to annotate a data structure to be eliminated. This is the only annotation the user needs to write, and the other three are inferred automatically. We treat `Packed` as a type constructor, so `Packed` T is a new type, different from T. Consequently, once the user annotates a type in the reference program with `Packed`, the program no longer type-checks.

At the terms level, we introduce `label` and `unlabel`, which serve as the constructor and the destructor of `Packed` values. For example, `label` $[1, 2]$ has the type `Packed List` and `unlabel` (`label` $[1, 2]$) $=$ $[1, 2]$. In a well-typed program, all operations on the intermediate data structure must be explicitly

[2]In this task, we could also define $?t_2\ (xs, ts) \coloneqq\ ?comb\ (?extract\ xs, ts)$, *i.e.* have *?extract* only deal with *xs*, since *ts* is an output of *?compress*, which is already a tuple of scalars. However, in general, we cannot take all values that come from *?compress* as fixed inputs of *?comb* because these values may form a non-scalar data structure. Some superfusion problems require eliminating inner data structures while leaving outer data structures intact.

annotated with label and unlabel. Finally, we introduce rewrite for marking the holes; rewrite t has the same type as t, and rewrite simply marks t as a term to be rewritten.

Example 3.1. The figure on the right shows the inferred annotations for mts. In this program, the output of tails is Packed, so all operations on this data structure are made explicit by label and unlabel. The uses of rewrite in this program correspond to the sketch holes identified in Fig. 1b. The only difference here is that the constant list in tails Nil is also moved out into a let-binding (this binding is eliminated from the synthesis result in a post-processing step).

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

In our annotation language, the conditions of correctness and feasibility are enforced by the type system, using the following two rules:

- **Correctness**: label and unlabel can only be used inside the argument to rewrite.
- **Feasibility**: rewrite terms should not return any non-Packed data structures.

The syntax, semantics, and type system of our annotation language can be found in Appendix A.1.

3.2 Generating Minimal Annotations

Given a program with only Packed annotations, the task of sketch generation is to insert label, unlabel, rewrite, and let-bindings to make the program well-typed in the annotation language, and meanwhile minimize the total size of arguments to rewrite (i.e., the three conditions listed above are satisfied). As shown in the running example (Fig. 1b), inserting let-bindings allows us to move irrelevant subterms out of rewrite, thereby helping to reduce the minimization objective.

More concretely, we would like to solve the following constrained optimization problem:

- **Search Space**: Given a program in the surface language with Packed annotations, we can transform it by repeatedly (1) inserting label, unlabel, or rewrite anywhere in the program, or (2) inserting a let-binding to move a subterm out of a term.
- **Constraint**: The resulting program must be well-typed in our annotation language.
- **Objective**: Minimize the total size of arguments to rewrite.

This optimization problem is difficult because there is an exponential number of possible transformation sequences. We prove that this problem is NP-complete in theory and then solve it using an off-the-shelf Max-SAT solver. Details on this procedure can be found in Appendix A.2.

4 SKETCH SOLVING

Given a reference program in the annotation language, SuFu eliminates intermediate data structures by solving a corresponding sketch problem, where every rewrite term is treated as a sketch hole. In this section, we describe this sketch problem (Sec. 4.1), introduce the synthesis algorithm of SuFu (from Sec. 4.2 to Sec. 4.5), and discuss the properties of SuFu (Sec. 4.6).

For clarity, this section discusses only a simpler case where Packed is used exactly once, which means, there is only one intermediate data structure to be eliminated. The approach in this section can be easily extended to the general case by synthesizing a separate compression function for every Packed. Details can be found in Appendix B.1.

4.1 Synthesis Problem

SuFu eliminates intermediate data structures in the reference program by rewriting the `Packed` type with a scalar type and rewriting `rewrite` terms with new terms that operate scalar attributes. We define a possible way of this rewrite as a *rewrite plan* (Def. 4.1).

Definition 4.1 (Rewrite Plan). Given a space $\mathcal{L}_{hole}$ of candidate terms and a reference program p involving one `Packed` type and n `rewrite` terms, a *rewrite plan* for program p on space $\mathcal{L}_{hole}$ is a pair $(T, \overline{t_i})$ comprising a scalar type T and a sequence $\overline{t_i}$ of n terms in $\mathcal{L}_{hole}$.

The *rewrite result* of rewrite plan $(T, \overline{t_i})$ is a program transformed from the program p, where the `Packed` type is replaced with T, and the ith `rewrite` term is replaced with t_i.

Example 4.2. For `mts_label` (Example 3.1), one expected rewrite plan is to take T as the type of integer pairs and take $\overline{t_i}$ as the three highlighted terms in Fig. 1c.

The synthesis problem of SuFu (denoted as *elimination problems*) is to find a rewrite plan such that the rewrite result has the same input-output behavior as the original program (Def. 4.3).

Definition 4.3 (Elimination Problem). Given a space $\mathcal{L}_{hole}$ of candidate terms, a program p, and a finite set I of inputs, the elimination problem is to find a rewrite plan for program p on space $\mathcal{L}_{hole}$ such that the rewrite result have the same evaluation result as p on every input in the input set I.

In the above definitions, we limit the rewrite plan to use only terms in a pre-defined space $\mathcal{L}_{hole}$. This is to ensure the efficiency of the resulting program. In Sec. 4.6, we shall prove an efficiency guarantee for the resulting program of SuFu when space $\mathcal{L}_{hole}$ includes only terms that can be evaluated within a constant number of steps, i.e., $O(1)$-time terms.

Besides, we consider only a finite set of inputs in the synthesis problem (Def. 4.3). We apply the CEGIS framework [Solar-Lezama 2009] to reduce the general case with an infinite number of possible inputs to this simpler case.

4.2 Workflow

The workflow of SuFu is shown in Algorithm 1. Given an elimination problem, SuFu first evaluates the reference program on every input in the input set and collects IO examples for `rewrite` terms (Line 1). Then, it synthesizes a compression function that maps intermediate data structures to their scalar attributes (Line 2).

With this function, SuFu converts the original examples to IO examples of sketch holes (Line 4) and then independently synthesizes each sketch hole using an existing PBE solver (Line 5). SuFu constructs its rewrite plan from these synthesis results (Line 7).

Algorithm 1: The workflow of SuFu.

```
Input: An elimination problem ($\mathcal{L}_{hole}, p, I$).
Output: A rewrite plan ($\overline{T_i}, \overline{t_i}$).
1 $\mathbb{E}_{orig} \leftarrow \bigcup_{in \in I}$ CollectExamples($p, in$);
2 $?compress \leftarrow$ CompressSynthesis($\mathbb{E}_{orig}$);
3 foreach sketch hole with index i do
4     $\mathbb{E}_{io} \leftarrow$ GetIOExample($i, \mathbb{E}_{orig}, ?compress$);
5     $t_i \leftarrow$ PBESolver($\mathcal{L}_{hole}, \mathbb{E}_{io}$);
6 end
7 return (output type of $?compress, \overline{t_i}$);
```

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

Please note that algorithm 1 stands only for a single CEGIS iteration. The rewrite plan it returns will be verified by an external verifier, and if incorrect, this algorithm will be invoked again with one additional counterexample to generate other candidate results.

4.3 Example Collection

Function `CollectExamples` collects examples by tracing the evaluation. Specifically, we utilize the *big-step environment semantics* [Dikotter 1990] of our annotation intermediate language (details in

Appendix A.1). The evaluation judgments of this semantics have the form of $E \vdash t \Downarrow v$, where E is an environment recording the values available in the context, t is the term to be evaluated, and v denotes the evaluation result.

The key advantage of this semantics is that its evaluation is conducted along the syntax, making it easy to collect examples of subterms. Specifically, given the reference program and an input, `CollectExamples` first constructs the derivation tree of evaluating the reference program on this input. Then, for every judgment on `rewrite` terms (i.e., $E \vdash \texttt{rewrite}\ t \Downarrow v$), it takes the environment E and the resulting value v as an IO example of the involved `rewrite` term. Here, we will exclude all functions from the environment because of the limitation of the PBE solver we use.

Example 4.4. The following is a part of the derivation tree of evaluating `mts_label` (Example 3.1) on input list [2], where E_{func} is environment storing available library functions (e.g., `tails` and `map`), and `label` v represents an intermediate data structure with content v.

$$\dfrac{\dfrac{\cdots}{\begin{array}{l}(E_{func}, \texttt{xs} \mapsto [2]) \vdash \\ \quad \texttt{tails xs} \Downarrow \texttt{label}\ [[2],[]]\end{array}} \qquad \dfrac{\cdots}{\begin{array}{l}(E_{func}, \texttt{xs} \mapsto [2], \texttt{ts} \mapsto \texttt{label}\ [[2],[]]) \vdash \\ \quad \texttt{rewrite (maximum (map sum (unlabel ts)))} \Downarrow 2\end{array}}}{(E_{func}, \texttt{xs} \mapsto [2]) \vdash \texttt{let ts = tails xs in rewrite (maximum (map sum (unlabel ts)))} \Downarrow 2}$$

From the right branch, SuFu collects environment ($\texttt{xs} \mapsto [2], \texttt{ts} \mapsto \texttt{label}\ [[2],[]]$) (where functions in E_{func} are excluded) and value 2 as an IO example of the involved `rewrite` term.

The collected examples serve a sufficient condition to ensure a correct rewrite result. In Example 4.4, the collected example is also satisfied by term `head xs`. We can safely replace the `rewrite` term with `head xs` without changing the full evaluation result because this replacement will only affect the derivation inside the right branch.

4.4 Synthesizing the Compression Function

Function `CompressSynthesis` synthesizes the compression function *?compress* to eliminate intermediate data structures in the collected examples (denoted as *raw examples*). With this function, every raw example (E, v) can be converted to an IO example (*convert ?compress* E, *convert ?compress* v) of a sketch hole, where *convert* replaces intermediate data structures in an environment (or value) with the corresponding outputs of *?compress*.

Example 4.5. We have seen the expected *?compress* for the `mts` task in Eq. 1. With this function, the raw example collected in Example 4.4 will be converted as follows.

$$\Big(\big(\texttt{xs} \mapsto [2], \texttt{ts} \mapsto \mathit{?compress}\ [[2],[]]\big), 2\Big) \xrightarrow{\text{Eq. 1}} \Big(\big(\texttt{xs} \mapsto [2], \texttt{ts} \mapsto (2,2)\big), 2\Big)$$

A valid *?compress* should ensure that its converted examples can be satisfied by sketch holes. Its formal specification is as follows, where n denotes the number of `rewrite` terms and $\mathbb{E}_i$ denotes the available raw examples of the ith `rewrite` term.

$$\bigwedge_{i=1}^{n} \exists ?t_i \in \mathcal{L}_{hole}, \forall (E, v) \in \mathbb{E}_i, \big(\mathit{convert}\ \mathit{?compress}\ E\big) \vdash ?t_i \Downarrow \big(\mathit{convert}\ \mathit{?compress}\ v\big) \tag{5}$$

SuFu synthesizes *?compress* from this specification by iteratively generating necessary attributes, as shown in algorithm 2. It maintains two values during the iteration (Line 1), a temporary result *compress* that records all known attributes and a queue Q storing those attributes that are already known but have not been processed yet.

Since `rewrite` terms are not allowed to return non-`Packed` data structures (Sec. 3.1), the output of each raw example can be divided into two parts, intermediate data structures and other scalar

Algorithm 2: Function `CompressSynthesis`.

```
Input: A set $\mathbb{E}_{orig}$ of original examples.
Output: A compression function.
1  $compress \leftarrow \lambda x.()$;  $Q \leftarrow \emptyset$;
2  FindAttributes(ExampleScalar($\mathbb{E}_{orig}$));
3  while $\neg Q$.Empty() do
4      $attr \leftarrow Q$.Pop();
5      $\mathbb{E}_{attr} \leftarrow$ ExampleAttr($\mathbb{E}_{orig}$, $attr$, $compress$);
6      FindAttributes($\mathbb{E}_{attr}$);
7  end
8  return $compress$;
9  Function FindAttributes($\mathbb{E}$):
10     foreach sketch hole with index $i$ do
11         $\mathbb{E}_i \leftarrow$ examples in $\mathbb{E}$ for the $i$th hole;
12         $compress_i \leftarrow$ SolveSingleHole($\mathbb{E}_i$);
13     end
14     foreach new attribute $attr$ in $\overline{compress_i}$ do
15         $compress \leftarrow \lambda x.(compress\ x, attr\ x)$;
16         $Q$.Push($attr$);
17     end
```

Table 3. The examples transformed by algorithm 2 when solving the *mts* task.

Location		Sample Example 1[†] Input	Sample Example 1[†] Out	Sample Example 2[†] Input	Sample Example 2[†] Out
Original		xs ↦ [2], t ↦ [], ts ↦ [[2], []]	2	xs ↦ [2], t ↦ [], ts ↦ [[]]	[[2], []]
Line 2		xs ↦ [2], t ↦ [], ts ↦ [[2], []])	2	xs ↦ [2], t ↦ [], ts ↦ [[]]	()
Line 6	#1	xs ↦ [2], t ↦ [], ts ↦ (2, [[2], []])	()	xs ↦ [2], t ↦ [], ts ↦ (0, [[]])	2
	#2	xs ↦ [2], t ↦ [], ts ↦ (2, 2, [[2], []])	()	xs ↦ [2], t ↦ [], ts ↦ (0, 0, [[]])	2

[†] To save space, we use a blue list *ts* to denote an intermediate list `label` *ts*.

values. SuFu first synthesizes attributes for calculating these scalar values (Line 2). It removes all intermediate data structures from the outputs of examples (via `ExampleConcrete`) and synthesizes from the transformed examples (via `FindAttributes`).

Function `FindAttributes` (Lines 9-17) synthesizes necessary attributes for a set of examples that involve only scalar outputs. For such examples, the choice of scalar attributes will not affect their outputs. Therefore, we can separately synthesize the attributes required by each hole (Lines 10-13, via `SolveSingleHole`) and directly merge all found attributes into the result[3] (Lines 14-17). We shall introduce function `SolveSingleHole` in Sec. 4.5.

Then, SuFu synthesizes new attributes for intermediate data structures in the output (Lines 3-6). It iteratively considers each known attribute *attr* and synthesizes new attributes for calculating it. To achieve this goal, SuFu transforms raw examples as follows (Line 5, via `ExampleAttr`).

- In the input, SuFu supplies the known attributes for every intermediate data structure (i.e., the output of *compress*), since the goal here is to synthesize new attributes.
- In the output, SuFu removes all scalar values and replaces intermediate data structures with their attributes specified by *attr*, since the goal here is to synthesize attributes for *attr*.

After this transformation, the output of every example will become concrete values. Therefore, we can apply function `FindAttributes` again to synthesize new attributes (Line 6).

Example 4.6. This example demonstrates how algorithm 2 synthesizes *?compress* for program `mts_label` (Example 3.1). We assume that enough raw examples are provided and shall illustrate using two sample examples (Tab. 3). Both samples are collected from `mts_label` [2] and they correspond to the third and the second `rewrite` terms, respectively.

[3]SuFu use observational equivalence to exclude duplicated attributes (Line 14). Two attributes will be regarded as the same if their values are the same for all intermediate data structures in the raw examples.

(1) First, SuFu applies `ExampleScalar` to transform examples (Row "Line 2" in Tab. 3) and then finds the attribute $attr_1$ of the maximum tail sum from the transformed examples. This attribute will be merged into the temporary result *compress* and pushed into queue Q.
(2) Then, in the first iteration, $attr_1$ is taken out from queue Q. SuFu applies `ExampleAttr` to transform examples (Row "Line 6/#1" in Tab. 3) and then finds the attribute $attr_2$ of the sum of all elements, which will also be merged with *compress* and pushed into queue Q.
(3) After that, in the second iteration, $attr_2$ is taken out from queue Q. SuFu applies `ExampleAttr` again to transform examples (Row "Line 6/#2" in Tab. 3). At this time, no extra attribute is found from the transformed examples, hence the iteration stops.

4.5 Synthesizing Necessary Attributes for a Single Sketch Hole

In algorithm 2, function `SolveSingleHole` needs to synthesize necessary attributes for a special set of examples, where (1) only a single sketch hole is involved, and (2) all outputs are scalar values. The synthesis task here is a special case of Eq. 5, as shown below.

$$\exists ?t \in \mathcal{L}_{hole}, \forall (E, v) \in \mathbb{E}, \left(\mathit{convert}\ ?\mathit{compress}\ E\right) \vdash ?t_i \Downarrow v \tag{6}$$

The main challenge of this task comes from the second-order quantifier $\exists ?t \in \mathcal{L}_{hole}$. We address this challenge by making two assumptions. First, since $\mathcal{L}_{hole}$ includes only $O(1)$-time terms (Sec. 4.1), which can only access a constant number of scalar values in the input, we assume that $\mathcal{L}_{hole}$ is specified by two orthogonal subspaces $\mathcal{L}_{extract}$ and $\mathcal{L}_{comb}$, which specify how to access scalar values in the input and how to accomplish the calculation using these scalar values, respectively.

Assumption 4.7. *$\mathcal{L}_{hole}$ is specified by two sub-spaces $\mathcal{L}_{extract}$ and $\mathcal{L}_{comb}$, where (1) $\mathcal{L}_{hole}$ comprises term* `let` x `=` *extract* `in` *comb for every extract $\in \mathcal{L}_{extract}$ and comb $\in \mathcal{L}_{comb}$, (2) every term in $\mathcal{L}_{extract}$ has a scalar type, and (3) terms in $\mathcal{L}_{comb}$ can only access variable x in the environment.*

Second, since most of the common scalar calculations can be accomplished efficiently, we assume that $\mathcal{L}_{comb}$ is expressive enough to implement all possible scalar functions on finite inputs.

Assumption 4.8. *For any function f whose input and output are both scalar values and any finite number of its input-output examples $(in_i, f\ in_i)$, there exists a term comb in $\mathcal{L}_{comb}$ satisfying these input-output examples, that is, $\forall i, (x \mapsto in_i) \vdash comb \Downarrow (f\ in_i)$.*

With these assumptions, we can transform the original specification (Eq. 6) in several steps. First, by Asm. 4.7, "$\exists ?t \in \mathcal{L}_{hole}$" can be replaced with "$\exists(?extract, ?comb) \in (\mathcal{L}_{extract}, \mathcal{L}_{comb})$". Then, by Asm. 4.8, *?comb* exists when there is a function that can produce the outputs of the examples from the values extracted by *?extract*. At last, since a function exists only when it never needs to produce different outputs from two identical inputs, we derive the lemma below.

Lemma 4.9. *When Asm. 4.7 and Asm. 4.8 both hold, Eq. 6 is equivalent to the formula below.*

$$\exists ?extract \in \mathcal{L}_{extract}, \forall (E_1, v_1), (E_2, v_2) \in \mathbb{E}, x_1 = x_2 \to v_1 = v_2 \tag{7}$$

where x_i denotes the evaluation result of ?extract on E_i, i.e., $(\mathit{convert}\ ?\mathit{compress}\ E_i) \vdash ?\mathit{extract} \Downarrow x_i$.

SuFu treats Eq. 7 as a joint specification for *?compress* and *?extract* and synthesizes both programs by enumeration. Specifically, it requires another program space $\mathcal{L}_{compress}$ specifying candidate *?compress*. Then, it enumerates all pairs of candidate *?compress* and *?extract* in the increasing order of the total size, until a pair satisfying Eq. 7 is found.

This enumeration method, though straightforward, is effective in practice because the target *?compress* and *?extract* are both usually small: *?compress* can be constructed compactly from library functions since its efficiency is not important, and *?extract* needs only to access scalar values in the input instead of performing any scalar calculation.

4.6 Properties

Soundness. SuFu ensures the soundness of the result. The terms found by SuFu must satisfy all collected raw examples, which are sufficient to ensure a resulting program with correct outputs.

Theorem 4.10 (Soundness). *For any elimination problem with reference program p and input set I, the rewrite result produced by SuFu must output the same as p on all inputs in I if the underlying PBE solver for sketch holes is sound.*

Proof. Due to the space limit, we move the proofs of theorems in this paper to Appendix E. □

Completeness. The completeness of SuFu relies on the expressiveness of $\mathcal{L}_{compress}$. Since the raw examples are collected from the evaluation of `rewrite` terms, there is always enough information in the inputs to calculate any output value and any scalar attribute (of intermediate data structures in the outputs). Consequently, SuFu can always find a proper *?compress* when $\mathcal{L}_{compress}$ is expressive enough and thus can find a rewrite result.

Theorem 4.11 (Completeness). *For any elimination problem satisfying Asm. 4.7 and Asm. 4.8, SuFu can find a rewrite plan if the underlying PBE solver for sketch holes is complete and $\mathcal{L}_{compress}$ can implement any function mapping from intermediate data structures to scalar values.*

Efficiency of the result. In theory, SuFu ensures that the resulting program will never have a larger time complexity. SuFu optimizes by replacing `rewrite` terms with terms in $\mathcal{L}_{hole}$. This replacement will never increase the time complexity when $\mathcal{L}_{hole}$ includes only $O(1)$-time terms.

Theorem 4.12 (Efficiency). *Let $cost(p, in)$ be the size of the derivation tree of evaluating program p on input in. For any elimination problem where $\mathcal{L}_{hole}$ includes only $O(1)$-time terms, the following formula is always satisfied by the reference program p and the rewrite result p' produced by SuFu.*

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

In practice, the resulting program of SuFu is usually strictly more efficient because the original `rewrite` terms usually involve time-consuming operators on intermediate data structures, resulting in a time complexity much larger than $O(1)$.

5 APPLICATIONS TO PROGRAM RESTRUCTURING

Aside from fusion, there is another line of prior work on synthesizing efficient programs, denoted as *program restructuring*, which aims to rewrite a reference program into a specific target form that is known as efficient. Many specialized approaches have been proposed for various target forms. In this paper, we find that for several target forms studied in the literature, such as divide-and-conquer [Farzan and Nicolet 2021b; Ji et al. 2023; Morita et al. 2007] and single-pass [Farzan et al. 2022; Pu et al. 2011], program restructuring can be reduced to a fusion problem for eliminating intermediate data structures in a template and then can be solved by SuFu.

Let us illustrate this reduction by taking divide-and-conquer (D&C) as an example. For tasks on lists, D&C suggests dividing the input list into two halves, recursively calculating for each half, and then combining the recursive results for the full list. Figure 2a shows a D&C program for `mts`. To combine the `mts` of the two halves, this program introduces the sum of all elements as an auxiliary value. The intuition here is that the `mts` of the whole list is either the `mts` of the right half or the sum of all elements in the right half plus the `mts` of the left half. As we can see, it is challenging to rewrite a program into D&C. To achieve this, we not only need to determine how to combine the recursive results but also need to discover necessary auxiliary values sometimes.

To convert this program into a fusion problem for eliminating intermediate data structures, we write a template program `dac_id` as shown in Fig. 2b. This template program is a simple D&C version

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

(a) A D&C program for `mts`.

```
dac_id Elt(_)@xs = xs
dac_id Cons(_, _)@xs =
  let (ls, rs) = split xs in
  concat (dac_id ls) (dac_id rs)
```

(b) A D&C program for the identity function.

Fig. 2. Programs related to divide-and-conquer (D&C). For simplicity, we consider only non-empty lists and assume a function `split` exists that can equally divide a list into two halves within $O(1)$ time.

of the identity function. It traverses the data structure the same as a standard D&C algorithm, but directly re-constructs and returns the whole input list.

Since `dac_id` performs only identity mapping, we can compose other programs with `dac_id` without changing their input-output behaviors. For example, the right figure shows a program `dac_mts` that still behaves the same as `mts`.

```
dac_id :: List -> Packed List
dac_mts xs = mts (dac_id xs)
```

Since `dac_id` traverses the data structure in the standard D&C manner, eliminating the intermediate data structure produced by `dac_id` for `dac_mts` will result in a program that keeps the D&C form and is also equivalent to `mts`. In this way, we can rewrite `mts` into the form of D&C. By applying SuFu to `dac_mts`, we can get the same D&C program for `mts` as shown in Fig. 2a.

The above procedure can be generalized to other target forms. First, we implement an identity function in the target form as the template. Then, we compose the template program with the program to be restructured, and at last, apply SuFu to eliminate intermediate data structures. More template programs can be found in Appendix D.1. Please note that the template needs only to be written once for each target form since it is irrelevant to the program to be restructured.

6 IMPLEMENTATION

Our implementation of SuFu is in C++ and is available in the supplementary material.

Program spaces. SuFu requires two program spaces $\mathcal{L}_{extract}$ and $\mathcal{L}_{comb}$ for specifying candidate terms of sketch holes, where $\mathcal{L}_{extract}$ comprises terms that extract from the environment and $\mathcal{L}_{comb}$ comprises terms for scalar calculations. We construct these two spaces as follows.

- $\mathcal{L}_{extract}$ includes the projection operator for accessing tuple (e.g., `?.1`) and the pattern match operator for accessing data structures (e.g., `match ? with Nil -> ? | Cons(h,t) -> ?` for lists).
- $\mathcal{L}_{comb}$ is the program space for conditional integer arithmetic in SyGuS-Comp [Alur et al. 2017a]. It includes the branch operator *if-then-else*, basic arithmetic operators such as $+$, basic Boolean operators such as *and*, and comparison operators such as $\leq$.
  This program space is expressive enough to satisfy Asm. 4.8. In the worst case, it can satisfy a finite set of examples by encoding all input-output pairs via nested *if-then-else*.

SuFu also requires a program space $\mathcal{L}_{compress}$ for specifying candidate compression functions. In our implementation, this program space includes all functions available in the reference program[4], the *DeepCoder*'s library [Balog et al. 2017] for operating lists (details in Appendix C.1), and the fold operator of every involved data structure for implementing customized recursions.

[4]The original programs in our evaluation are in terms collected from previous studies without introducing new functions.

Verification. SuFu is based on the CEGIS framework and requires an external verifier to generate counterexamples for incorrect results. Our implementation uses bounded verification (details in Appendix C.2) and it is future work to combine SuFu with a more sophisticated verifier.

Besides, to reduce the time cost of verification, SuFu will initialize the example set of CEGIS with 10^3 random examples. These examples can exclude most of the incorrect results and thus greatly reduce the number of CEIGS iterations (i.e., the number of invocations of the verifier).

Others. We take *Z3* [de Moura and Bjørner 2008] as the Max-SAT solver for sketch generation (Sec. 3.2) and take *PolyGen* [Ji et al. 2021] as the PBE solver for sketch holes (Sec. 4.2).

7 EVALUATION

To evaluate SuFu, we report three experiments to answer the following research questions.

- **RQ1**: How effective is SuFu in eliminating intermediate data structures?
- **RQ2**: How effective is SuFu in program restructuring problems that can be reduced to fusion?
- **RQ3**: How effective is the synthesizer of SuFu (algorithm 1) in solving its sketch problem?

7.1 Experimental Setup

Baseline Solvers. The first baseline solver we considered is a state-of-the-art deductive fusion system [Hinze et al. 2010], denoted as Trans. It unifies and generalizes previous transformation systems using a concept of *recursive co-algebras.*

Besides, since some program restructuring problems can be reduced to fusion, we also compare SuFu with specialized tools for two of such program restructuring problems.

- Synduce [Farzan et al. 2022] is an approach that rewrites a program into the form of structural recursion. A structural recursion takes an inductive data structure as the input and calculates by traversing the input at most once. Synduce rewrites a reference program by unfolding the recursion via a specialized technique namely *counter-example guided partial bounding.*
- AutoLifter [Ji et al. 2023] is an approach that rewrites a program into a divide-and-conquer program or other forms similar to dvide-and-conquer, e.g., single-pass on lists [Schweikardt 2009] and incrementalization [Acar et al. 2005]. AutoLifter reduces these program restructuring tasks into a specialized synthesis problem, namely *lifting problems*, and efficiently solves it by decomposition.

SuFu is strictly more general than the two baselines and can be applied to all program restructuring synthesis tasks supported by the baselines. We have discussed the case of divide-and-conquer (supported by AutoLifter) in Sec. 5, and another example of structural recursion (supported by Synduce) can be found in Appendix D.1.

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

- *Source 1: fusion.* We investigate previous studies on fusion [Bird 1989; Bird and de Moor 1997; Gill et al. 1993; Hu et al. 1997; Wadler 1988] and collect 16 tasks that require replacing intermediate data structures with scalar attributes[5]. Among them, 8 tasks here are collected from studies for guiding manual optimization [Bird 1989; Bird and de Moor 1997], whose approaches are not automated.
- *Source 2: recursion.* We consider the original dataset of our baseline solver Synduce [Farzan et al. 2022] and include all tasks (178 in total) into our dataset. In this procedure, we find that 60 tasks in this dataset include manually provided auxiliary values due to the limit of

[5]In some cases, the elimination is achieved by re-organizing the data structure instead of replacing with scalar attributes. We exclude such tasks because they are out of the scope of SuFu. One such task can be found in Appendix B.2.

Table 4. The profile of our dataset

Source	#Task	Size	#Packed
Fusion	16	126.5	1.250
Recursion	178	157.4	1.101
D&C	96	252.2	1.010
Total	290	187.1	1.079

Table 5. Comparison between SuFu and Trans.

Source	#Task	SuFu	Trans
Fusion	16	**14**	≤ 11
Recursion	178	**170**	≤ 129
D&C	96	**80**	≤ 10
Total	290	**264**	≤ 150

Table 6. Details on the performance of SuFu.

Source	Sketch Generation		Sketch Solving			
	Time	$S_{rewrite}$	Time	$S_{compress}$	$S_{extract}$	S_{holes}
Fusion	0.015	16.00	14.95	9.071	4.643	21.79
Recursion	0.018	17.65	24.73	7.876	6.759	30.41
D&C	0.037	16.39	48.98	11.64	6.563	84.66
Total	0.021	17.14	38.06	9.080	6.587	46.70

Synduce. For example, in the task of synthesizing a recursion for `mts`, the original reference program in Synduce's dataset actually returns a pair of the maximum tail sum and the sum of all elements. We remove all these auxiliary values when constructing our dataset.

- *Source 3: divide-and-conquer (D&C).* We consider the original dataset of our baseline solver AutoLifter [Ji et al. 2023] and include all tasks (97 in total) into our dataset.

Tab. 4 shows the profile of our dataset, including the number of tasks (Column "#Task"), the average number of AST nodes in the reference program (Column "Size"), and the average number of `Packed` types in the reference program (Column "#Packed").

Others. Our experiments are conducted on Intel Core i7-8700 3.2GHz 6-Core Processor, and every single execution is assigned with a time limit of 10 minutes. Our dataset and the experimental data are available in the supplementary material.

7.2 RQ1: Evaluation on Eliminating Intermediate Data Structures

Procedure. In this experiment, we compare SuFu with Trans on the whole dataset. Since the original study of Trans does not provide an implementation, we calculate only an upper bound of the performance of Trans by characterizing a class of tasks that cannot be solved by Trans.

In detail, all rules in Trans are for fusing the composition of two recursions into a single one that produces *exactly* the same output. Consequently, Trans can never solve tasks where new attributes are necessary. For example, given the `mts` program Fig. 1a, Trans may introduce a recursion constructing the list of tail sums by fusing `map sum` with `tails`, or search for a recursion calculating only the maximum suffix sum when further fusing with `maximum`. However, Trans will never consider the sum of all elements because it is not produced in the input program.

By this analysis, we take the number of tasks that do not require new attributes as an upper bound of the performance of Trans and compare it with the number of tasks solved by SuFu.

Results. The results of this experiment are summarized in Tab. 5, where we report the number of solved tasks for SuFu[6] and report the corresponding upper bound for Trans. These results show that SuFu significantly outperforms Trans in the number of solved tasks.

Tab. 6 supplies more statistics on the performance of SuFu.

[6]Since the verifier in our implementation performs only bounded verification, we manually verify the results of SuFu and confirm that they are all *completely correct.*

Table 7. Comparison between SuFu and specialized approaches for program restructuring.

Baseline	#Task	SuFu		Baseline	
		#Solved	Time	#Solved	Time
Synduce	178	**170**	17.91	125	**1.669**
AutoLifter	96	80	65.94	**82**	**15.61**

- For the stage of sketch generation (Sec. 3), we report the time cost (Column "Time") and the total size of `rewrite` terms in the resulting annotated program (Column "$S_{rewrite}$"),
- For the stage of sketch solving (Sec. 4), we report the time cost (Column "Time") and the sizes of synthesized programs, where Columns "$S_{compress}$", "$S_{extract}$", and "S_{holes}" correspond to *?compress*, *?extract*, and hole solutions, respectively.

We make the following observations from the results reported in Tab. 6.

- The Max-SAT-based sketch generation (Sec. 3.2) is efficient in practice though it is exponential-time in theory. One reason is that, in a practical task, the production and consumption of intermediate data structures are usually direct, making the search space of the optimization problem far smaller than the theoretical upper bound.
- Both *?compress* and *?extract* are much smaller than the programs filled to sketch holes. This result matches our analysis in Sec. 4.5.

At last, as shown in Tab. 5, SuFu fails on 26 out of 290 tasks in our dataset. We investigate these failed tasks and conclude two major reasons as follows.

First, SuFu may fail when the target programs are extremely complex. In 17 tasks, the target *?compress* either exceeds the scalability of the enumeration-based synthesizer (Sec. 4.5) or cannot be implemented in the program space $\mathcal{L}_{compress}$ at all; and in 4 more tasks, SuFu succeeds in finding an expected *?compress* but the underlying PBE solver times out in synthesizing sketch holes even when separate input-output examples are provided. The performance of SuFu on these tasks may be improved if a more suitable program space, a more efficient synthesizer for *?compress* and *?extract*, or a more efficient PBE solver for sketch holes is provided.

Second, on the other 5 tasks, SuFu synthesizes an unintended *?compress* whose corresponding hole solutions are extremely complex, making the underlying PBE solver time out. Specifically, the specification of *?compress* (Eq. 5) only ensures that sketch holes exist to satisfy the corresponding examples, but different *?compress* satisfying this specification may correspond to hole solutions with different scales. For example, one reference program in our dataset is `sqrsum (upto n)`, where `upto` constructs a list from 1 to n, `sqrsum` calculates the square sum of elements in a list, and the output of `upto` is annotated with `Packed`. When solving this task, SuFu synthesizes an empty *?compress* because integer n is already enough to determine the final output via function $n(n+1)(2n+1)/6$. However, such a function is so complex that the PBE solver we use cannot synthesize a corresponding program within the time limit. A possible direction to fix this flaw is to further consider the scale of hole solutions in the specification of *?compress*. This is future work.

7.3 RQ2: Comparison with Specialized Approaches for Program Restructuring

Procedure. In this experiment, we compare SuFu with Synduce and AutoLifter on their respective datasets. Specifically, we run Synduce and AutoLifter on the tasks in their original datasets, run SuFu on the corresponding tasks in our dataset, record all solved tasks, and compare the performance of these approaches.

Results. The results of this experiment are summarized in Tab. 7, where we report the number of solved tasks and the average time cost for each approach. These results show that, although

SuFu requires more time in synthesis, it performs well in the number of solved tasks. Concretely, it solves more tasks than Synduce and a close number of tasks when compared with AutoLifter. Please note that SuFu is a general solver that, unlikely the baselines, is not optimized for these specialized types of tasks.

7.4 RQ3: Comparison with Sketch Solvers

Procedure. Since the core synthesis problem of SuFu (Def. 4.3) is in the form of sketch problems, in this experiment, we compare our synthesis algorithm (algorithm 1) with a state-of-the-art sketch solver, Grisette [Lu and Bodík 2023]. Specifically, for each task in our dataset, we use the same method as SuFu to extract the sketch (Sec. 3), then invoke Grisette to solve the corresponding sketch problem, and compare its performance with the original SuFu.

Results. The results of this experiment are summarized in the right table. As shown, our synthesis algorithm significantly outperforms Grisette on both the number of solved tasks and the time cost.

#Task	SuFu		Grisette	
	#Solve	Time	#Solve	Time
290	**264**	**18.21**	85	28.91

8 RELATED WORK

Eliminating intermediate data structures. Many deductive fusion systems have been designed for eliminating intermediate data structures [Bird 1989; Bird and de Moor 1997; Chin 1992; Fokkinga 1992; Gill et al. 1993; Hamilton 2001; Hinze et al. 2010; Hu et al. 1996; Meijer et al. 1991; Takano and Meijer 1995; Wadler 1988]. They iteratively apply pre-defined rules to rewrite an input program toward the direction with fewer intermediate data structures. In comparison, both SuFu and these approaches have their advantages.

- On the one hand, SuFu can better utilize task-specific properties via inductive synthesis.
- On the other hand, SuFu requires more inputs because it needs the user to explicitly specify the intermediate data structures to be eliminated in the input program (via `Packed`).

This shortage of SuFu can be potentially addressed by inferring the intermediate data structures that can be eliminated via program analysis and heuristic rules. This is future work.

Synthesizing efficient programs. Synthesizing efficient programs is an important problem in program synthesis, and there have been different lines of research on this problem. First, *super optimization* [Bornholt et al. 2016; Phothilimthana et al. 2014, 2016; Schkufza et al. 2013; Sharma et al. 2015] aims to find the most efficient implementation for a reference program by inductive synthesis. However, existing approaches in this line consider only loop-free programs formed by low-level instructions and thus cannot be applied to eliminate intermediate data structures.

Second, some existing studies aim to rewrite an inefficient reference program into specific target forms that are known as efficient [Acar et al. 2005; Farzan et al. 2022; Farzan and Nicolet 2017, 2021a; Fedyukovich et al. 2017; Ji et al. 2023; Morita et al. 2007; Pu et al. 2011], which we denoted as *program restructuring* in this paper. Every approach in this category is specifically proposed for a certain target form, and cannot be applied to the general problem of eliminating intermediate data structures, where the synthesized program has no specific form. Besides, as discussed in Sec. 5, program restructuring for some target forms [Farzan et al. 2022; Ji et al. 2023] can be reduced to fusion and thus can also be solved by SuFu.

At last, there is another line of work, namely *type- and resource-aware synthesis* [Hu et al. 2021; Knoth et al. 2019], which 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 can hardly scale to synthesize large programs because they need to 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 [Lu and Bodík 2023; Lubin et al. 2020; Porncharoenwase et al. 2022; Solar-Lezama et al. 2006; Torlak and Bodík 2014]. However, general sketch solvers can hardly scale up to our tasks where the target programs of sketch holes are usually large (46.70 AST nodes on average in our dataset).

Second, SuFu synthesizes a ghost function *?compress* to extract local input-output examples for sketch holes. From a high level, this step synthesizes an invariant for the resulting programs and then decomposes the whole synthesis problem into subtasks on sub-programs. In this sense, SuFu is related to the following two previous approaches.

- Escher [Albarghouthi et al. 2013] takes the reference outputs as the results of recursions and thus reduces a recursive synthesis problem into synthesizing the recursion body. Compared with it, *?compress* in our problem is not given and needs also to be synthesized.
- Natural synthesis [Qiu and Solar-Lezama 2017] synthesizes loop invariants to decompose a loop-related synthesis problem into subtasks without loops. However, this approach still relies on a sketch solver and thus can hardly scale up to our tasks.

At last, SuFu uses the definition of functions to eliminate an existential quantifier over sketch holes (Sec. 4.5). A similar idea has been used by several existing approaches [Ji et al. 2023; Kuncak and Blanc 2013]. Compared with them, the range of our quantifier is limited to $O(1)$-time programs instead of all programs, rendering the infeasibility of using the definition of functions. We address this problem by further decomposing the sketch holes.

9 CONCLUSION

In this paper, we study the problem of eliminating intermediate data structures and propose a novel tool based on inductive program synthesis, namely SuFu. Given a reference program annotated with data structures to be eliminated, SuFu will first generate a sketch by transforming the reference program into an annotation language and then fill this sketch by synthesizing proper $O(1)$-time expressions. To achieve an efficient synthesis, SuFu will synthesize a ghost function *?compress* and use it to decompose the sketch problem into independent synthesis problems for each hole. We implement SuFu and evaluate it on a dataset of 290 tasks collected from previous studies. The results demonstrate the effectiveness of SuFu compared with previous related approaches.

We believe there should be more interesting applications of SuFu left for exploration, given that eliminating intermediate data structures is one fundamental way of improving the efficiency of functional programs. Besides, SuFu can potentially be integrated into a compiler to offer complex optimizations if the annotation on intermediate data structures can be automatically generated, and a complete verifier is available.

REFERENCES

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

Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. 2013. Recursive Program Synthesis. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 8044)*, Natasha Sharygina and Helmut Veith (Eds.). Springer, 934–950. https://doi.org/10.1007/978-3-642-39799-8_67

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

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

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

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

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

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

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

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

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

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

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

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

Azadeh Farzan and Victor Nicolet. 2017. Synthesis of divide and conquer parallelism for loops. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*. 540–555. https://doi.org/10.1145/3062341.3062355

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

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

Grigory Fedyukovich, Maaz Bin Safeer Ahmad, and Rastislav Bodík. 2017. Gradual synthesis for static parallelization of single-pass array-processing programs. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 572–585. https://doi.org/10.1145/3062341.3062382

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

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

Susanne Graf and Hassen Saïdi. 1997. Construction of Abstract State Graphs with PVS. In *Proceedings of the 9th International Conference on Computer Aided Verification (CAV '97)*. Springer-Verlag, Berlin, Heidelberg, 72âĂŞ83.

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

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

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

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

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

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

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

Ruyi Ji, Yuwei Zhao, Yingfei Xiong, Di Wang, Lu Zhang, and Zhenjiang Hu. 2023. Divide and Conquer Divide-and-Conquer – Inductive Synthesis for D&C-Like Algorithmic Paradigms. arXiv:2202.12193 [cs.PL]

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

Rajeev Kohli, Ramesh Krishnamurti, and Prakash Mirchandani. 1994. The Minimum Satisfiability Problem. *SIAM J. Discret. Math.* 7, 2 (1994), 275–283. https://doi.org/10.1137/S0895480191220836

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

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

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

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

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

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

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

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

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

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

Xiaokang Qiu and Armando Solar-Lezama. 2017. Natural synthesis of provably-correct data-structure manipulations. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 65:1–65:28. https://doi.org/10.1145/3133889

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

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

Nicole Schweikardt. 2009. One-Pass Algorithm. In *Encyclopedia of Database Systems*, Ling Liu and M. Tamer Özsu (Eds.). Springer US, 1948–1949. https://doi.org/10.1007/978-0-387-39940-9_253

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

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

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

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

Emina Torlak and Rastislav Bodík. 2013. Growing solver-aided languages with rosette. In *ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2013, part of SPLASH '13, Indianapolis, IN, USA, October 26-31, 2013*, Antony L. Hosking, Patrick Th. Eugster, and Robert Hirschfeld (Eds.). ACM, 135–152. https://doi.org/10.1145/2509578.2509586

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

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

Yuepeng Wang, Xinyu Wang, and Isil Dillig. 2018. Relational program synthesis. *PACMPL* 2, OOPSLA (2018), 155:1–155:27.

$$
\begin{array}{lcl}
t \in \text{Term} & ::= & \text{constants} \mid x \mid \mathsf{app}(t_1,t_2) \mid \lambda(x,T,t) \mid \mathtt{fix}(t) \mid \mathtt{let}(x,t_1,t_2) \mid \mathtt{if}(t_1,t_2,t_3) \mid C(t) \\
 & \mid & \mathsf{tuple}(\overline{t_i}) \mid \mathsf{proj}(t,i) \mid \mathsf{match}(t,\overline{C_i(x_i).t_i}) \mid \mathtt{rewrite}(t) \mid \mathtt{label}(t) \mid \mathtt{unlabel}(t) \\
v \in \text{Value} & ::= & \text{constants} \mid (\mathsf{closure}\ x \to t, E) \mid \mathtt{fix}(v) \mid C(v) \mid \mathsf{tuple}(\overline{v_i}) \mid \mathtt{label}(v) \\
E \in \text{Environment} & ::= & \emptyset \mid E, x \mapsto v \\
B \in \text{BaseType} & ::= & \mathsf{Unit} \mid \mathsf{Int} \mid \mathsf{Bool} \mid B_1 \times \cdots \times B_n \mid \mathsf{ind}(\overline{C_i : (B_i, m_i)}) \mid \mathtt{Packed}(B) \\
T \in \text{Type} & ::= & B \mid T \to T \mid T_1 \times \cdots \times T_n \qquad \Gamma \in \text{TypeContext} \quad ::= \quad \emptyset \mid \Gamma, x : T
\end{array}
$$

Fig. 3. The problem definition language $\mathcal{L}$ of SuFu presented via *abstract binding trees* [Harper 2016], where C denotes a constructor of some inductive data structure, and $\overline{e_i}$ denotes a sequence of some elements.

$$\boxed{E \vdash t \Downarrow v}$$

$$
\text{(E-Var)}\ \frac{x \in E}{E \vdash x \Downarrow E(x)} \qquad
\text{(E-Abs)}\ \frac{}{E \vdash \lambda(x,T,t) \Downarrow (\mathsf{closure}\ x \to t, E)} \qquad
\text{(E-Let)}\ \frac{E \vdash t_1 \Downarrow v_x \qquad E, x \mapsto v_x \vdash t_2 \Downarrow v}{E \vdash \mathtt{let}(x, t_1, t_2) \Downarrow v}
$$

$$
\text{(E-App-Abs)}\ \frac{E \vdash t_1 \Downarrow (\mathsf{closure}\ x \to t_x, E_x) \qquad E \vdash t_2 \Downarrow v_2 \qquad E_x, x \mapsto v_2 \vdash t_x \Downarrow v}{E \vdash \mathsf{app}(t_1, t_2) \Downarrow v}
$$

$$
\text{(E-Rewrite)}\ \frac{E \vdash t \Downarrow v}{E \vdash \mathtt{rewrite}(t) \Downarrow v} \qquad
\text{(E-Label)}\ \frac{E \vdash t \Downarrow v}{E \vdash \mathtt{label}(t) \Downarrow \mathtt{label}(v)} \qquad
\text{(E-Unlabel)}\ \frac{E \vdash t \Downarrow \mathtt{label}(v)}{E \vdash \mathtt{unlabel}(t) \Downarrow v}
$$

Fig. 4. Selected evaluation rules of our intermediate language.

A APPENDIX: SKETCH GENERATION

A.1 Language

Syntax. Figure 3 shows the intermediate language we considered in this paper, including the surface language and the additional constructs introduced in Sec. 3.1. We highlight several details in this language as follows.

- Value $(\mathsf{closure}\ x \to t, E)$ denotes a *function closure* [Dikotter 1990] and is used for defining semantics. Intuitively, a closure is a lambda expression $\lambda(x,T,t)$ paired with an environment E recording the values bound to local variables when the lambda expression is defined.
- `Packed`, `rewrite`, `label`, and `unlabel` are the four additional constructs we introduced for generating the sketch. We limit `Packed` to annotating only basic types because this paper does not consider the elimination of functions.
- Type $\mathsf{ind}(\overline{C_i : (B_i, m_i)})$ defines an inductive data type. In detail, it denotes recursive type $\mu X.\overline{C_i : B_i \times X \times \cdots \times X}$ (where X appears m_i times). For example, `List` (Figure 1a) can be declared as $\mathsf{ind}(\mathsf{Nil} : (\mathsf{Unit}, 0), \mathsf{Cons} : (\mathsf{Int}, 1))$.

Semantics. We use a *big-step environment semantics* [Dikotter 1990] of our language to collect input-output examples for syntactic subterms. The evaluation judgments have the form of $E \vdash t \Downarrow v$, representing that term t is evaluated to value v under environment E.

Figure 4 shows some selected evaluation rules of this semantics.

- For lambda expressions, Rule E-Abs evaluates a lambda expression to a closure. Each time a lambda expression is applied, Rule E-App-Abs switches the environment to the one in the closure and thus evaluates the function body under the correct environment.

$$\boxed{\Gamma \vdash_s t : T}$$

$$\text{(T-VAR)}\ \frac{x \in \Gamma}{E \vdash_s x : \Gamma(x)} \qquad \text{(T-ABS)}\ \frac{\Gamma, x : T_1 \vdash_s t : T_2}{\Gamma \vdash_s \lambda(x, T_1, T_2) : T_1 \to T_2} \qquad \text{(T-LET)}\ \frac{\Gamma \vdash_s t_1 : T_1 \qquad \Gamma, x : T_1 \vdash_s t_2 : T_2}{\Gamma \vdash_s \texttt{let}(x, t_1, t_2) : T_2}$$

$$\text{(T-REWRITE)}\ \frac{\Gamma \vdash_{\text{in}} t : S \qquad S \in \text{ScalarType}^+}{\Gamma \vdash_s \texttt{rewrite}(t) : S} \qquad \text{(T-LABEL)}\ \frac{\Gamma \vdash_{\text{in}} t : B \qquad B \in \text{BaseType}}{\Gamma \vdash_{\text{in}} \texttt{label}(t) : \texttt{Packed}(B)} \qquad \text{(T-UNLABEL)}\ \frac{\Gamma \vdash_{\text{in}} t : \texttt{Packed}(T)}{\Gamma \vdash_{\text{in}} \texttt{unlabel}(t) : T}$$

Fig. 5. Selected typing rules of our intermediate language, where scope s represents whether the current term is enclosed by `rewrite`, BasicType comprises types that do not involve functions, and ScalarType$^+$ comprises basic types that do not involve non-intermediate data structures.

- For our additional constructs, `rewrite` performs no calculation as it is only a label on terms (Rule E- REWRITE), and `label` and `unlabel` convert between intermediate and normal data structures by adding and removing a label on values (Rules E-LABEL and E-UNLABEL).

Typing rules. As discussed in Sec. 3.1, we designed a type system to ensure that `rewrite` indeed labels all terms producing or consuming intermediate data structures. Fig. 5 presents some selected typing rules of $\mathcal{L}$. The typing judgments have the form of $\Gamma \vdash_s t : T$, representing that term t has type t under context Γ when it is inside `rewrite` (represented as s = in) or outside `rewrite` (represented as s = out). The typing rules related to the surface language are all standard, and the typing rules of our additional constructs are designed to ensure the two rules in Sec. 3.1.

- According to Rule 1, both `label` and `unlabel` can only be typed inside `rewrite`.
- According to Rule 2, the output type of `rewrite` should not include non-intermediate data structures. Valid output types of `rewrite` include scalar types, types of intermediate data structures, and their tuples. All these types will become scalar after the elimination.

A.2 Translation Problem

SUFU generates the sketch by translating the reference program into a well-typed program in our intermediate language. As discussed in 3.2, we treat this translation problem as an optimization task, where the action is to insert `label`, `unlabel`, `rewrite`, and `let`-bindings to the reference program, and the goal is to get a well-typed program while minimizing the total size of `rewrite` terms. The following is the formal description of this problem.

Definition A.1 (Translation Problem). Given a reference program p in the surface language with some data structures annotated by `Packed`, we call a well-typed program p' in our intermediate language as a *direct extension* of a p if (*rmlabel* t') and t are exactly the same in syntax, where function *rmlabel* removes all uses of `rewrite`, `label`, and `unlabel` from term t'.

Then, the *translation problem* is to find a direct extension p' of p such that objective value (*optsize* (*addlet* p')) is minimized, where function *optsize* calculates the total size of `rewrite` terms, and function *addlet* moves subterms irrelevant to intermediate data structures (i.e., subterms do not involve any `label` and `unlabel`) out of `rewrite` by inserting `let`-bindings.

Note that in this definition, we treat the insertion of `let`-bindings as a part of the objective function instead of an action. The point here is that `let`-bindings can be greedily inserted to minimize `rewrite` terms (i.e., move out all subterms that are irrelevant to intermediate data structures) once the uses `label`, `unlabel`, and `rewrite` have been determined.

Example A.2. Let us consider the translation for term t := `app(g,app(f,0))`, where `f` and `g` are two functions with types `Int` → `Packed(List)` and `List` → `Int`, respectively.

Term t_1 ≔ `rewrite(app(g,unlabel(app(f,0))))` is a direct extension of t since it is well-typed and is the same as t after removing `rewrite` and `unlabel`. To calculate the objective value of t_1, *addlet* is first applied and transforms t_1 to t_1' ≔ `let(x,app(f,0),rewrite(app(g,unlabel(x))))`, where `app(f,0)` is moved out as it does not involve any `label` or `unlabel`. Then, *optsize* returns an objective value equal to 3 because `rewrite` encloses three non-annotation constructors in t_1', including two variables, and one `app`. One can verify that t_1 already achieves the minimum objective value on this task. Therefore, our generator will return t_1' (i.e., *addlet* t_1) as the final annotated term.

The translation problem is difficult because it aims at the optimal one among all direct extensions, whose number can be exponential. Specifically, we prove that there is no polynomial-time algorithm for this problem unless $\mathsf{P} = \mathsf{NP}$, as shown below.

Theorem A.3. *The translation problem is NP-complete.*

Given this difficulty, we leverage the power of constraint solvers to solve the translation problem. Specifically, we encode the space of direct extensions as hard constraints, encode the objective function as soft constraints, and then find the optimal extension via a Max-SAT solver.

In the remainder of this section, we demonstrate this procedure using the sample problem in Example A.2. To encode the space of direct extensions, we introduce two binary variables f_i and g_i for every sub-term whose type is in BaseType, where f_i denotes whether `label` or `unlabel` is used to flip[7] the type, and e_i denotes whether a sub-term is directly enclosed by `rewrite`. These variables specify a symbolic term as follows, where $[x]$`C` represents that constructor `C` is inserted when x is true.

$$[e_1]\texttt{rewrite}\left([f_1]\texttt{label}\left(\texttt{app}\left(\texttt{g}, [e_2]\texttt{rewrite}([f_2]\texttt{unlabel}(\texttt{app}(\texttt{f}, [e_3]\texttt{rewrite}([f_3]\texttt{label}(\texttt{0})))))\right)\right)\right)$$

Hard constraints are generated to ensure this symbolic term is well-typed. For example, f_3 must be *false* to match the input type of `f`, e_2 must imply $\neg f_2$ because `List` does not belong to ScalarType$^+$, and f_3 must imply $e_1 \vee e_2 \vee e_3$ to ensure the use of `label` is enclosed by `rewrite`. All these constraints are collected through a symbolic version of type-checking.

Soft constraints are generated to encode the objective function. In brief, for every constructor, we extract its condition to be not counted in the objective value as a soft constraint. For example, $(\neg e_1 \wedge \neg e_2) \vee (\neg f_3 \vee e_3)$ is extracted for the inner `app`, where $(\neg e_1 \wedge \neg e_2)$ represents the case that this constructor is initially outside `rewrite`, and the remainder represents the case that this constructor can be extracted out by *addlet*. Through this encoding, the more soft constraints are satisfied, the smaller the objective value will be, and thus the optimal labeling can be found by Max-SAT solving.

After feeding these constraints to a Max-SAT solver, the optimal assignments to the variables are obtained, where e_1 and f_2 are set to true. By instantiating the symbolic term on this assignment, we can get the same labeling result as discussed in Example A.2.

B APPENDIX: SAMPLE ELIMINATION TASKS

B.1 Multiple Intermediate Data Structures

In Sec. 4, we focus on a special case where `Packed` is used only once. In this section, we demonstrate how SuFu deals with the general case where multiple `Packed` annotations are available.

Fig. 6 shows a program with multiple intermediate data structures. In this program, `single_pass` is a general but inefficient template for an algorithm namely *single-pass*. Given an input list, *single-pass* suggests visiting each value in the list one by one, updating a state after each visit, and calculating the results from the final state. Following this idea, `single_pass` converts any target function `f` to

[7]That means, `label` is inserted when the type is intermediate, and otherwise, `unlabel` is inserted. Our translator does not consider repetitive intermediate types such as `Packed(Packed(List))` because they are meaningless in practice.

```
single_pass f xs = f (run xs Nil)
where
run :: List -> Packed List -> Packed List
run Nil state = state
run Cons(h, t) state = run t (append h state)
```

```
map :: (List -> Int) -> NList -> Packed List
mts xs = maximum (map sum (tails xs))

mts_sp xs = single_pass mts xs
```

Fig. 6. A reference program for applying *single-pass* to *mts*.

```
single_pass f xs =
  let state = rewrite (label[?i1] Nil) in
  let xs' = run xs state in rewrite (f (unlabel xs'))
where
run :: List -> Packed[?i2] List -> Packed[?i3] List
run Nil state = state
run Cons(h, t) state =
  let state' = rewrite (label[?i4] (append h (unlabel
      state))) in run t state'
```

```
map :: (List -> Int) -> NList -> Packed[?
     i5] List
mts xs = let tsum = map sum (tails xs) in
  rewrite (maximum (unlabel tsum))

mts_sp xs = single_pass mts xs
```

We omit the details in `map` for simplicity.

Fig. 7. The translated program with symbolic kind labels for applying *single-pass* to *mts*.

single-pass by reconstructing the whole input in the state (`run`) and then directly applying `f` to get the result. In Fig. 6, we take `mts` as the target function of `single_pass`.

The main difference in the general case is that there may be multiple kinds of intermediate data structures requiring different sets of scalar attributes. In Fig. 6, there are three data structures annotated to be eliminated, the state taken by `run`, the final state produced by `run`, and the list of all tail sums produced by `map sum`. From the perspective of human optimization, we know the uses of `Packed` here correspond to two kinds of intermedia data structures, where the two in `single_pass` correspond to the state in *single-pass*, and the one in `mts` corresponds to the list of tail sums. These two kinds should be replaced with different scalar attributes, states with the maximum suffix sum and the list of tail sums with the maximum.

To eliminate all these intermediate data structures, after the translation, SuFu will distinguish different kinds of intermediate data structures by attaching *kind labels* to the uses of `Packed` and `label`. Specifically, `Packed`$[x]$ `List` with different kind label x will be treated as different types, and `label`$[x]$ will be treated as the constructor specialized for the intermediate data structures in kind x. To infer these labels, SuFu first assigns symbolic labels to `Packed` and `label` (Fig. 7) and collects constraints on these labels via a round of type-checking. For example, $?i_1$ and $?i_2$ must be the same because the intermediate data structure constructed by `label`$[?i_1]$ will be used as the second input of `run`, whose type is `Packed`$[?i_1]$ `List`. The constraints collected in this task include $?i_1 = ?i_2, ?i_2 = ?i_3$, and $?i_2 = ?i_4$, and thus divide symbolic labels into two equivalent classes $\{?i_1, ?i_2, ?i_3, ?i_4\}$ and $\{?i_5\}$. SuFu assigns different labels 1 and 2 to these two classes and thus distinguishes the intermediate data structures constructed in `single-pass` and `mts` into two kinds.

The synthesis procedure of SuFu can be easily extended to support two kinds of intermediate data structures. SuFu will synthesize two compression functions $?compress_1$ and $?compress_2$ for respectively specifying attributes for the two kinds. Every raw example collected for `rewrite` terms can be converted into an input-output example of a sketch hole by replacing intermediate data structures constructed by `label`[1] and `label`[2] with the outputs of $?compress_1$ and $?compress_2$, respectively. Our synthesis algorithm for $?compress$ (algorithm 2) can be straightforwardly extended to synthesize $?compress_1$ and $?compress_2$ simultaneously.

$$
\begin{array}{rccl}
\text{Integer expr} & N_{\mathbb{Z}} & \to & \text{IntConst} \mid N_{\mathbb{Z}} \oplus N_{\mathbb{Z}} \mid \texttt{sum}\ N_{\mathbb{L}} \mid \texttt{len}\ N_{\mathbb{L}} \mid \texttt{head}\ N_{\mathbb{L}} \\
 & & \mid & \texttt{last}\ N_{\mathbb{L}} \mid \texttt{access}\ N_{\mathbb{Z}}\ N_{\mathbb{L}} \mid \texttt{count}\ F_{\mathbb{B}}\ N_{\mathbb{L}} \mid \texttt{neg}\ N_{\mathbb{L}} \\
 & & \mid & \texttt{maximum}\ N_{\mathbb{L}} \mid \texttt{minimum}\ N_{\mathbb{Z}} \\
\text{List expr} & N_{\mathbb{L}} & \to & \text{Input} \mid \textit{take}\ N_{\mathbb{Z}}\ N_{\mathbb{L}} \mid \texttt{drop}\ N_{\mathbb{Z}}\ N_{\mathbb{L}} \mid \texttt{rev}\ N_{\mathbb{L}} \\
 & & \mid & \texttt{map}\ F_{\mathbb{Z}}\ N_{\mathbb{L}} \mid \texttt{filter}\ F_{\mathbb{B}}\ N_{\mathbb{L}} \mid \texttt{zip} \oplus N_{\mathbb{L}}\ N_{\mathbb{L}} \mid \texttt{sort}\ N_{\mathbb{L}} \\
 & & \mid & \texttt{scanl} \oplus N_{\mathbb{L}} \mid \texttt{scanr} \oplus N_{\mathbb{L}} \\
\text{Binary Operator} & \oplus & \to & + \mid - \mid \times \mid \texttt{min} \mid \texttt{max} \\
\text{Integer Function} & F_{\mathbb{Z}} & \to & (+\ \text{IntConst}) \mid (-\ \text{IntConst}) \mid \texttt{neg} \\
\text{Boolean Function} & F_{\mathbb{B}} & \to & (< 0) \mid (> 0) \mid \texttt{isodd} \mid \texttt{iseven}
\end{array}
$$

Fig. 8. The program space formed by operators in *DeepCoder*'s library.

B.2 Non-Scalar Elimination

As mentioned in Sec. 7.1, there are some fusion tasks in the previous study that do not require a complete replacement with scalar values. For example, `id xs = rev (rev xs)` is a program reversing a list twice and produces an intermediate data structure in its inner reversion. One way to eliminate this data structure is to directly calculate the final result in the inner recursion, resulting in a more efficient program as `id' Nil = Nil` and `id' Cons(h,t) = Cons(h,id' t)`.

SuFu cannot solve this task because it focuses only on replacing intermediate data structures with their scalar attributes. One possible extension of SuFu here is to remove all requirements on scalars while extending our synthesizer correspondingly. We leave it to future work.

C APPENDIX: IMPLEMENTATION

C.1 Library for Lists

Our implementation supplies the *DeepCoder*'s library to the program space of *?compress* for implementing complex list-related programs. Fig. 8 shows the program space formed by this library. Many non-trivial programs can be implemented using the operators in this library. For example, the maximum tail sum of a list `xs` can be implemented as `maximum (scanr + xs)`, where `scanr` constructs all non-empty tails of a list and reduces each tail to an integer via a given associative binary operator, as shown below.

$$\texttt{scanr}\ (\oplus)\ [xs_1, \dots, xs_n] \coloneqq [xs_1 \oplus \cdots \oplus xs_n, \dots, xs_{n-1} \oplus xs_n, xs_n]$$

C.2 Probabilistic Verifier

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 will initialize 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.

D APPENDIX: EVALUATION

D.1 Program Restructuring for Structural Recursion

```
Tree = Leaf(Int) | Node(Tree, Tree)
tree_rec ref t = ref (rec t)
where
rec :: Tree -> Packed Tree
rec Leaf(v)@t = t
rec Node(l, r) = Node(rec l, rec r)
```

(a) A recursion template for Tree.

```
tree_rec' _ t =
  let tres = rec' t in tres.1
where
rec' Leaf(v) = (max v 0, v)
rec' Node(l, r)) =
  let (lmts, lsum) = rec' l in
  let (rmts, rsum) = rec' r in
  (max lmts (lsum + rmts), lsum + rsum)

mits_rec' t = tree_rec' mits t
```

(b) The synthesis result of SuFu, where variable names are assigned manually for readability.

Fig. 9. Synthesize a structural recursion for `mips` by SuFu

```
Tree = Leaf(Int) | Node(Tree, Tree)
rec Leaf(v) = ?f1 v1
rec Node(l, r) = ?f2 (rec l) (rec r)
```

Fig. 10. A recursion skeleton for trees.

Structural recursions are a special kind of recursions where the input data structure is traversed at most once and recursion calls are only made on sub-structures of the input. The program restructuring problem for structural recursion is to rewrite a reference program using a given recursion skeleton while keeping the semantics unchanged [Farzan et al. 2022; Farzan and Nicolet 2021a]. For example, Fig. 10 shows a common recursion skeleton for trees, and the synthesis problem is to complete it into a program semantically equivalent to the reference program.

SuFu can be applied to this problem by (1) constructing a program that composes the reference program and a recursion template, and (2) eliminating the intermediate data structure in this constructed program. Fig. 9a shows the program constructed for trees. In detail, `rec` traverses the tree in the same way as the skeleton (Fig. 10) but performs no change, and `tree_rec` applies the reference program `ref` to the input tree after the traverse. The program in Fig. 9a can be automatically generated from the skeleton.

When applying SuFu to eliminate the intermediate data structure in `tree_rec ref`, SuFu will search for a program performing the calculation of the reference program step-by-step along the recursion of `rec`, resulting in a structural recursion. For example, let us consider a reference program `mits t = mts (flatten t)` which first collects all values on the input tree in order into a list (`flatten`) and then returns the maximum tail sum of the value list (`mts`). SuFu can generate a corresponding structural recursion from `tree_rec mits`, shown in Fig. 9b[8].

D.2 Failure Analysis

In our first experiment, Tab. 5 shows that SuFu fails in 26 out of 290 tasks in our dataset. We investigate these failed tasks and conclude two major reasons as follows.

First, SuFu may fail when the target programs are extremely complex. In 17 tasks, the target *?compress* either exceeds the scalability of the enumeration-based synthesizer (Sec. 4.5) or cannot be implemented in the program space $\mathcal{L}_{compress}$ at all; and in 4 more tasks, SuFu succeeds in finding an expected *?compress* but *PolyGen* (the PBE solver we use) times out in synthesizing sketch holes even when separate input-output examples are provided. The performance of SuFu on these tasks may be improved if a more suitable program space, a more efficient synthesizer for *?compress* and *?extract*, or a more efficient PBE solver for sketch holes is provided.

[8]The reference program is no longer used in `tree_rec` because `rec'` has been rewritten specialized for `mits`. In practice, such unused parameters can be easily removed at compile time

Second, SuFu may synthesize an unintended *?compress* of which the corresponding sketch holes are much more complex, making the PBE solver time out when synthesizing sketch holes. Specifically, the specification of *?compress* (Eq. 5) only ensures that sketch holes exist to satisfy the corresponding examples, but different *?compress* satisfying this specification may correspond to sketch holes with different complexity. For example, one input program in our dataset is `sqrsum (upto n)`, where `upto` constructs a list from 1 to n, `sqrsum` calculates the square sum of elements in a list, and the output of `upto` is marked with `Packed`. When solving this task, SuFu synthesizes an empty *?compress* because integer n is already enough to determine the final output through function $n(n+1)(2n+1)/6$. However, such a function is so complex that *PolyGen* cannot synthesize a corresponding program within the time limit. One way to fix this flaw is to further consider the complexity of sketch holes in the specification of *?compress*. This is future work.

E APPENDIX: PROOFS

Theorem E.1 (Theorem Thm. 4.10). *For any elimination problem with reference program p and input set I, the rewrite result produced by SuFu must output the same as p on all inputs in I if the underlying PBE solver for sketch holes is sound.*

Proof. Since the underlying PBE solver is sound, the terms in the resulting rewrite plan must satisfy the input-output examples converted by *?compress*. Therefore, to prove this theorem, it is enough to prove that a rewrite result must be correct if its rewrite plan $(T, \overline{t_i})$ satisfies all examples converted by *?compress*. To achieve this, let us first introduce some auxiliary functions and notations.

- For term t, we use *rewrite* t to denote the rewrite result of replacing `rewrite` subterms in t with the corresponding terms in the rewrite plan.
- For environment E, we use *convert* E to denote the converted environment where (1) every involved intermediate data structure is replaced with the corresponding output of *?compress*, and (2) the function body t in every involved function closure is replaced with *rewrite* t. Similarly, we use *convert* v to denote the counterpart for values.

We prove the correctness of the rewrite result of $(T, \overline{t_i})$ by proving the following claim.

- For any environment E, term t, and value v, the formula below holds if $(T, \overline{t_i})$ and *?compress* satisfies all examples collected from the derivation tree of $E \vdash t \Downarrow v$.

$$E \vdash t \Downarrow v \implies (\mathit{convert}\ E) \vdash (\mathit{rewrite}\ t) \Downarrow (\mathit{convert}\ v)$$

The correctness of the rewrite result can be obtained from this claim by taking E as empty, taking t as $\mathsf{app}(p, v_{in})$ for $v_{in} \in I$, and taking v as the corresponding evaluation result.

The above claim can be proved straightforwardly by induction on the derivation tree of $E \vdash t \Downarrow v$. We discuss several representative cases as follows.

Case 1: $t = \mathtt{rewrite}(t')$. Let i be the index of this `rewrite` term. At this time, a raw example will be collected from the derivation tree of $E \vdash t \Downarrow v$, which ensures our claim.

Case 2: $t = \lambda(x, T, t')$. At this time, the original evaluation result is $v := (\mathtt{closure}\ x \rightarrow t', E)$, and the evaluation result of $(\mathit{rewrite}\ t)$ is $v' := (\mathtt{closure}\ x \rightarrow (\mathit{rewrite}\ t'), \mathit{convert}\ E)$. At this time, v' is equal to $(\mathit{convert}\ v)$, hence out claim holds.

Case 3: $t = \mathsf{app}(t_f, t_p)$. At this time, $E \vdash t \Downarrow v$ is derived from the following three facts.

$$E \vdash t_f \Downarrow (\mathtt{closure}\ x \rightarrow t_x, E_x) \qquad E \vdash t_p \Downarrow v_p \qquad E_x, x \mapsto v_p \vdash t_x \Downarrow v$$

Since the set of hole-examples collected for (E, t) is the union of these three branches, $(T, \overline{t_i})$ and *?compress* must also satisfy all examples collected from these branches. Then, by applying the inductive hypothesis and using the definition of related functions, we can get the following facts related to the rewrite result, where we abbreviate the rewrite result (*rewrite* t) for term t as t', the conversion result (*convert* E) for environment E as E', and the conversion result of a value v as v'.

$$E' \vdash t'_f \Downarrow (\texttt{closure}\ x \to t'_x, E'_x) \qquad E \vdash t'_p \Downarrow v'_p \qquad E'_x, x \mapsto v'_p \vdash t'_x \Downarrow v'$$

Then, the fact $E' \vdash \texttt{app}(t'_f, t'_x) \Downarrow v'$ can be obtained by applying Rule E-App-Abs. Therefore, our claim still holds in this case because $\texttt{app}(t'_f, t'_x)$ is equal to t', i.e., the rewrite result of the term t. □

Theorem E.2 (Thm. 4.11). *For any elimination problem satisfying Asm. 4.7 and Asm. 4.8, SuFu can find a rewrite plan if the underlying PBE solver for sketch holes is complete and $\mathcal{L}_{compress}$ can implement any function mapping from intermediate data structures to scalar values.*

Proof. Recall the workflow of SuFu (algorithm 1). By Lemma 4.9, the synthesized *?compress* ensures that the synthesis tasks of sketch holes are realizable. Consequently, the invocation of `PBESolver` must terminate when the synthesis tasks of sketch holes are realizable. Therefore, we only need to prove that `CompressSynthesis` (algorithm 2) will always terminate.

We achieve this by constructing a *killer* compression function $compress^*$ that is always a valid result of `SolveSingleHole` in every iteration. Let us start with the definition of $compress^*$. Since the set of raw examples is finite, there is only a finite number of intermediate data structures involved. Let m be the maximum size of these intermediate data structures. Then, there exists an injection mapping from intermediate data structures of size at most m to tuples of $O(m)$ scalar values. For example, we can map a list $[x_1, \dots, x_n]$ with $n \leq m$ to tuple $(n, x_1, \dots, x_n, 0, \dots)$ (which includes $m + 1$ integers). We take $compress^*$ as any of such injection.

Now, we show why $compress^*$ must be a valid result of `SolveSingleHole`. Suppose there is an arbitrary invocation of `SolveSingleHole`, where the set of examples $\overline{(E_i, v_i)}$. There must exist a function f satisfying $f\ E_i = v_i$ because the environment of raw examples must include enough information to calculate any scalar attribute of output intermediate data structures. After converting by $compress^*$, the ith example will become $(convert\ compress^*\ E_i, v_i)$. Since there is only a finite number of examples, there exists an *?extract* that can extract all values from the environments of these examples. Let the semantics of this term as g, which is again an injection from $(convert\ compress^*\ E_i)$ to scalar values. At this time, the input of *?comb* is $g\ (convert\ compress^*\ E_i)$ and the output is v_i, which is equal to $f\ E_i$. Therefore, *?comb* can be constructed as follows.

$$?comb := f \circ (convert\ compress^*)^{-1} \circ g^{-1}$$

On the one hand, *?comb* is a well-defined function since both g and $(convert\ compress^*)$ are injective. On the other hand, since the input and the output of *?comb* are both scalar values, *?comb* can be implemented in $\mathcal{L}_{comb}$ by Asm. 4.8. Therefore, $compress^*$ is a valid solution for this arbitrary invocation of `SolveSingleHole`.

Given that $compress^*$ is always a valid solution, no new scalar attribute will be necessary once $compress^*$ is found, and at that time, `CompressSynthesis` will terminate after clearing the queue. Moreover, since `SolveSingleHole` synthesizes *?compress* by enumeration and there is only a finite number of programs in $\mathcal{L}_{compress}$ before $compress^*$, this killer compression function will ultimately be found after all its previous programs have been returned as attributes. Therefore, we prove that `CompressSynthesis` always terminates, which implies the target theorem. □

Theorem E.3 (Thm. 4.12). *Let $cost(p, in)$ be the size of the derivation tree of evaluating program p on input in. For any elimination problem where $\mathcal{L}_{hole}$ includes only $O(1)$-time terms, the following formula is always satisfied by the reference program p and the rewrite result p' produced by SuFu.*

$$\exists c > 0, \forall in, cost(p', in) \leq c \cdot cost(p, in) \tag{8}$$

Proof. Let $(\overline{t_i}, T)$ be the rewrite plan found by SuFu. Since $\mathcal{L}_{hole}$ includes only $O(1)$-time terms, we can take constant c as a large enough integer such that the derivation tree of every t_i will never be larger than c. The task remaining is to prove that Eq. 8 holds for this constant.

For any input *in*, let T and T' be the derivation tree of evaluating p and p' on *in*, respectively. Then, T' can always be transformed from T by replacing all subtrees related to `rewrite` terms with derivation trees of evaluating some t_i. 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.

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

□

Theorem E.4 (Thm. A.3). *The translation problem is NP-complete.*

Proof. The annotation generation problem is clearly inside NP, and we prove its NP-hardness by reducing a limited version of the Min-SAT problem [Kohli et al. 1994], which is still NP-complete, to the annotation problem. An instance of this Min-SAT problem is specified by n Boolean variables $\overline{x_i}$ and m clauses, each limited to either a variable x_i or the form of $x_i \vee \neg x_j$, and the goal is to find an assignment to $\overline{x_i}$ such that the number of satisfied clauses is minimized.

To solve a given instance of the Min-SAT problem in Horn clauses, we construct an annotation problem in the following way.

First, for every variable x_i, we introduce a function f_i as $\lambda(x, \texttt{List}, x)$. In the annotation, one can choose to either keep the body unchanged or to annotate the body as `rewrite(label(x))`. The output types of f_i in these two cases are `List` and `Packed(List)`, respectively. We regard the former case as setting x_i to *true*, and the later case as setting x_i to *false*, and according to this construction, setting a variable x_i to *false* will enlarge the objective value by 1.

Then, to encode the objective of minimizing the number of satisfied clauses, the key is to put penalties on our objective value when a clause is satisfied.

- For clause x_i, we construct a term as $\texttt{app}(\lambda(x, \texttt{Packed}(\texttt{List}), 1), \texttt{app}(f_i, \texttt{Nil}))$. At this time, when x_i is satisfied (i.e., the output type of f_i is not intermediate), the inner application will be annotated as $\texttt{let}(x, \texttt{app}(f_i, \texttt{Nil}), \texttt{rewrite}(\texttt{label}(x)))$, enlarging the objective value by 1. Otherwise, no annotation needs to be inserted so that the objective value will not change.
- For clause $x_i \vee \neg x_j$, we construct the following term parameterized by an integer c.

  $$\texttt{app}(\lambda(x, T, 1), \texttt{tuple}(t, t', \dots, t'))$$

  where t' repeats c times, T is defined as $(\texttt{List} \to \texttt{List}) \times \texttt{Packed}(\texttt{List}) \times \cdots \times \texttt{Packed}(\texttt{List})$, term t is defined $\lambda(x, \texttt{List}, \texttt{app}(f_j, x))$, and term t' is defined as $\texttt{app}(f_i, \texttt{Nil})$.
  At this time, there are three possible cases.
  - When $\neg x_j$ (i.e., the output type of f_j is `Packed(List)`), the function body in t must be annotated with `unlabel`, and `rewrite` can only be annotated to the whole term. Because t' can be extracted out by *addlet*, the objective value will be enlarged by $c + 8$ in this case.
  - When $x_i \wedge x_j$, t' will be annotated as $\texttt{let}(x, \texttt{app}(f_i, \texttt{Nil}), \texttt{rewrite}(\texttt{label}(x)))$, so the objective value will be enlarged by c in this case.
  - Otherwise, no annotations will be inserted, so the objective value will not change.

  In summary, each usage of this construction can put a penalty in range $[c, c + 8]$ on the objective value when clause $x_i \vee \neg x_j$ is satisfied.

We finish the reduction by combining the above construction. Specifically, we take c as $10m + 2n$, and repeatedly insert terms to a program including all definitions of f_i. For each clause in the form of x_i, we insert the corresponding term c times into the program, and for each clause in the form of $x_i \vee x_j$, we insert the corresponding term with parameter c once into the program. According to the above analysis, for each assignment to x_i satisfying k clauses, the minimum objective value among corresponding annotation plans must be inside range $[ck, ck + 8m + n]$. Therefore, the minimum number of satisfied clauses must be $\lfloor w/c \rfloor$, where w is the minimum objective value obtained by solving the constructed annotation generation problem. □