# Computation of Alias Sets from Shape Graphs for Comparison of Shape Analysis Precision

Viktor Pavlu
*Institute of Computer Languages*
*Vienna University of Technology*
*Vienna, Austria*
*vpavlu@complang.tuwien.ac.at*

Markus Schordan
*Institute of Computer Science*
*University of Applied Sciences Technikum Wien*
*Vienna, Austria*
*schordan@technikum-wien.at*

Andreas Krall
*Institute of Computer Languages*
*Vienna University of Technology*
*Vienna, Austria*
*andi@complang.tuwien.ac.at*

***Abstract*—Shape analysis is a static pointer analysis technique that models heap-allocated data structures in greater detail than the typical pointer analysis does. To a compiler, this information is crucial for deciding wether two expressions refer to the same memory location, as a great number of analyses depend on this aliasing information.**

**Various shape analyses have been introduced but their relation in terms of precision often remains unclear as analyses cannot be compared directly due to different representations of analysis results.**

**The aim of this work is to make the precision of shape analysis results comparable. Our solution is based on extracting alias information from shape analysis results. We also propose a significant improvement in precision over an existing 3-valued logic based algorithm to compute alias sets from shape graphs.**

**Using this method we are able to compare the precision of graph-based shape analyses. We demonstrate our algorithm by comparing the precision of two classic examples of graph-based shape analyses, proposed by Sagiv, Reps & Wilhelm (SRW) and Nielson, Nielson & Hankin (NNH) which were implemented for C++ in the SATIrE program analysis framework. Comparison of the computed alias sets gives a measure of quality by which NNH is more precise than SRW shape analysis for each of our benchmarks.**

***Keywords*-pointer aliasing; shape analysis; pointer analysis; shape graph; alias set; precision;**

## I. Introduction

A program analysis can be evaluated with respect to performance and precision. The performance relates to the computational complexity of the algorithm used for computing the desired properties of a given program, whereas precision can be considered a challenge for answering questions about programs and reasoning on whether those questions can be answered at all. Within recent years different approaches have been presented for taming the problem of analyzing programs that create heap-allocated data structures. These objects are linked via pointers and can form arbitrary data structures. Programs that operate on these data structures use and modify variables and refer to specific objects within data structures or share single objects. Here the well-known aliasing problem comes into play.

The question of whether two variables form an alias pair has been attacked by approaches like points-to analysis and shape analysis. Both approaches allow to extract alias information from the computed analysis results. Points-to analyses focus more on the connections between pointer variables on the stack and model the heap very coarsely, while shape analyses model the connections between heap-allocated data structures in great detail. Points-to analysis was tailored towards scalability and being applied to larger programs by providing an acceptable level of precision, whereas shape analysis aims at establishing precise properties of a program at an acceptable level of run-time. Recently notable advances have been made in making shape analysis scalable to larger programs as well [1], [2]. Therefore the interesting question arises how precise a given shape analysis is compared to another shape analysis or to some points-to analysis.

We present a significant improvement to the algorithm given by Reps, Sagiv, and Wilhelm [3][Formula (12.23), p. 12.32.] for computing alias information from shape analysis results and show that the results are indeed useful for determining a measure of precision by which given shape analyses differ in quality. We selected two well-known shape analyses for demonstrating the usefulness of such an evaluation and give an exact method for determining the relative precision factor for a given program.

The shape analysis published by Sagiv, Reps, and Wilhelm [4], [5] determines for each program location a single shape graph which approximates the possible shapes of heap-allocated data structures. The other shape analysis of interest, published by F. Nielson, H.R. Nielson, and C. Hankin [6], determines a *set* of shape graphs for each program point. We shall term the first analysis SRW analysis and the latter NNH analysis.

The comparison turns out to be interesting because even though the representation of null-pointers is different, the union of the set of shapes computed by NNH, is very similar to the single graph computed by SRW analysis. Notwithstanding we are able to show that there exists substantially more precise aliasing information in the NNH-computed set of shape graphs.

To shed more light on the subject of precision we determine alias pair sets that can be computed from both analysis results and compare those. This method of comparison can be applied to all kinds of pointer analyses from which aliasing information can be extracted.

The rationale behind comparing alias sets is that the effectiveness of many compiler optimizations depends on the quality of aliasing information. In this way, the shape analysis that produces fewer conservative results is enabling more optimizations and thus offers the better analysis information. Still, this metric sets aside that some aliasing pairs (i. e., in loops) are more important than others.

We propose an improved algorithm to extract the aliasing information inherent in the computed shape analysis results. Our algorithm obtains more precise aliasing information than previously presented algorithms. Using this algorithm to leverage all available information in shape analysis results and compute alias pair sets, we are able to determine a relative measure of alias set sizes. The differences are significant and suggest that the question "how precise is the analysis" is best answered in relation to another analysis and can be pinned down to a specific factor of precision for any given program.

## II. Computing 3-Valued Alias Sets

Alias sets have traditionally been grouped into must- and may-alias sets, where must-aliases are a subset of the may-aliases. With the use of 3-valued alias sets a finer grained classification is possible. It allows to separate those may-aliases which are not must-aliases into another set, which we shall call *strict may-alias* set. This set represents all expression pairs for which an analysis was not able to produce a better result than the most conservative answer, i. e., that the expressions *may* alias. Two expressions that definitely alias (expressions that are must-aliases) no longer add to the size of the set that includes the conservative answer. Our evaluation in section IV benefits from this distinction because it allows a more detailed comparison between shape analysis results. We first introduce terminology and continue with the presentation of our algorithm afterwards.

### A. Terminology

Three-valued alias sets make differences in precision more directly visible than the traditional must/may-alias classification. We therefore add the term "strict may-alias set" to the traditional terms.

**must-alias** Two pointer expressions $e_v$ and $e_w$ are said to be *must-aliases* at program point *pt* if all execution paths in program $\mathbf{P}$ ending at *pt* produce a structure in run-time memory in which both $e_v$ and $e_w$ refer to the same concrete location.

**no-alias** Two pointer expressions $e_v$ and $e_w$ are said to be *not aliased* at program point *pt* if there exists no execution path in program $\mathbf{P}$ ending at *pt* that produces a structure in run-time memory in which $e_v$ and $e_w$ refer to the same concrete location.

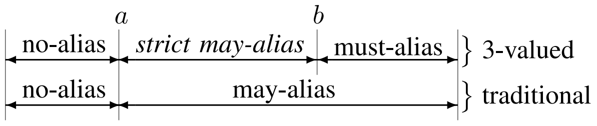

Figure 1. Relation between traditional and 3-valued alias sets.

**strict may-alias** Two pointer expressions $e_v$ and $e_w$ are said to be *strict may-aliases* at program point *pt* if they are neither must-aliases nor no-aliases.

**may-alias** Two pointer expressions $e_v$ and $e_w$ are said to be *may-aliases* at program point *pt* if they are either must-aliases or strict may-aliases.

Using above definitions, two pointer expressions $e_v$ and $e_w$ are strict may-aliases only if they cannot be identified as no-aliases or must-aliases. This is always the case when there exist some execution paths in which both $e_v$ and $e_w$ refer to the same concrete location, and some in which they do not; and, even more importantly, when the information obtained by the analysis does not include enough information to decide wether two expressions referring to the same *abstract location* also refer to the same *concrete location*.

Figure 1 graphically shows the differences between the traditional alias sets and our three-valued alias sets. The most conservative result in both representations is that every expression is strictly may-aliased with every other expression. But we can improve this result by identifying expression pairs as unaliased or must-aliased expressions by interpreting what a shape analysis collected about the heap structure. The more is known about the heap, the smaller the set of strict may-aliases becomes. In the traditional may-alias classification, however, the may-alias term subsumes both strict may-aliases and must-aliases and hence fails to represent improvements in precision that allow to turn strict may-aliases into must-aliases. In Fig. 1 such improvements correspond to a move of barrier *b* to the left. Using 3-valued alias sets this is seen as a reduction of the strict may-alias set while in the traditional partitioning no change can be observed.

### B. Extracting Alias-Sets from Shape Graphs

The alias computation algorithm for which we contribute a significant increase in precision is due to [3][Formula (12.23), p. 12.32.]. For two pointer expressions and a given set of compatible shape graphs [6] it computes the aliasing relation of the expression pair at that program point. To obtain alias sets, the algorithm is applied to all combinations of pointer expressions in the program. Therefore the algorithm can be directly applied to the results of the NNH analysis. SRW shape graphs can be easily converted to NNH compatible sets of shape graphs. Our efficient implementation [7] fuses conversion and alias computation for SRW shape graphs.

We present the algorithm as Functions ALIAS_TYPE (Alg. 1) and ALIAS_TYPE_SG (Alg. 2), showing the differences between the original algorithm and our proposed improvement side-by-side in Alg. 2 line 15.

In the following we assume availability of these functions:

NODES(SG, Expr)
: Ordered list of heap nodes in shape graph $SG$ that lie on the access path described by the pointer expression *Expr*

SELS(Expr)
: List of selectors in the pointer expression *Expr*

FIRST(List)
: First element of *List*

SECOND(List)
: Second element of *List*

LAST(List)
: Last element of *List*

REST(List)
: Sub-list starting with the second element of *List*

REV(List)
: Elements of *List* in reversed order

### C. Imprecision in the Original Algorithm

The original algorithm (cf. Alg. 1 and 2) returns 0, $\frac{1}{2}$, or 1 meaning that the queried expressions are unaliased, strictly may-aliased, or must-aliased, respectively. It has two sources of imprecision that can cause the conservative result $\frac{1}{2}$: First, different execution paths leading to *pt* may produce different heaps. Only if $e_v$ and $e_w$ refer to the same location in *every* (no) execution path, $e_v, e_w$ are must-aliases (not aliased). In every other case the answer has to be $\frac{1}{2}$ as it cannot be decided at compile-time which of the execution paths will be taken at run-time (see the early exits in lines 7 and 13 in procedure ALIAS_TYPE).

The other reason for $\frac{1}{2}$-results lies in the abstraction mechanism of SRW and NNH shape graphs. Abstract locations directly pointed to by a variable get labeled with that variable. As these abstract locations have names they are easily discernable from other abstract locations, so whenever two pointer expressions end at the same abstract location it is also clear that the pointer expressions refer to the same *concrete location* if, and only if, the edges actually exist in the concrete shape graph. For heap locations not directly pointed to by a variable this is different as these locations are represented by the single abstract summary location $n_\emptyset$. This limits abstract shape graphs to a finite size bounded by the number of variables in the program and ensures termination of the shape analysis, but also introduces imprecision: two expressions having the summary location $n_\emptyset$ as their final node could be aliased or not, irrespective of the summary location's sharing. Two expressions ending at $n_\emptyset$ therefore must be classified as strict may-aliases (cf. line 15) in the original algorithm shown in the box to the right in Algorithm 2, ALIAS_TYPE_SG.

**Algorithm 1** Determine Aliasing between pointer expressions $e_v$ and $e_w$ for all shape graphs $SGS_{pt}$ at $pt$

```
procedure ALIAS_TYPE($SGS_{pt}, e_v, e_w$)
    $SG \leftarrow$ any graph in $SGS_{pt}$
    $Alias \leftarrow$ ALIAS_TYPE_SG($SG, e_v, e_w$)
    if $Alias = \frac{1}{2}$ then
        no further tests required if
        $\frac{1}{2}$ in at least one SG
        return $\frac{1}{2}$
    end if
    for all $SG \in SGS_{pt}$ do
        if $Alias \neq$ ALIAS_TYPE_SG($SG, e_v, e_w$) then
            no further tests required if aliasing
            differs in two SGs
            return $\frac{1}{2}$
        end if
    end for
    return $Alias$
end procedure
```

### D. More Precision with Common Tails

The "common tails" improvement we present is able to reduce the imprecision caused by the summary location abstraction. Instead of looking only at the final node of two pointer expressions to decide their aliasing, our algorithm takes a sequence of selectors at the end of both access paths into account. For the improved algorithm we replaced line 15 of the original algorithm with a call to the CTAIL function given in Alg. 3.

If two pointer access paths described by expressions $e_v$ and $e_w$ meet at some intermediate node that is not the summary location and from there on have the same sequence of selectors (i. e. a "common tail"), then the paths not only have the same subsequent nodes including the final node, but the corresponding expressions $e_v$ and $e_w$ also refer to the same concrete location, even if the final node is the summary location $n_\emptyset$. This follows from an invariant of compatible shape graphs: source node and selector of a named location uniquely determine the target, i. e., the 'sel' field of an object in memory always points to the same address, no matter through what pointer expression the object was reached.

In the absence of such a "common tail", i. e., when there is no intermediate node where the expressions meet, the two paths could still meet at their final node. For named nodes this is easily detected even in the original algorithm; if the final node is the summary location $n_\emptyset$ we have two cases: (i) The summary location is *unshared* and represents only heap nodes pointed to by at most one heap node, so this last indirection cannot introduce aliasing. Knowing that

**Algorithm 2** Determine Aliasing between pointer expressions $e_v$ and $e_w$ in a single shape graph $SG$

```
 1: procedure ALIAS_TYPE_SG($SG, e_v, e_w$)
 2:     $nodes_v \leftarrow$ NODES($SG, e_v$)
 3:     $nodes_w \leftarrow$ NODES($SG, e_w$)
 4:     if $[\,] = nodes_v \vee [\,] = nodes_w$ then
 5:         at least one expression does not point to
 6:         a node in SG
 7:         return 0
 8:     else if LAST($nodes_v$) = LAST($nodes_w$) then
 9:         if LAST($nodes_v$) $\neq n_\emptyset$ then
10:             both expressions end at the same
11:             named location
12:             return 1
13:         else
14:             both expressions end at the summary location
15:  | our "common tails" improvement                   | original
     | $n_v \leftarrow$ REV(NODES($SG, e_v$))           | algorithm
     | $n_w \leftarrow$ REV(NODES($SG, e_v$))           |
     | $sels_v \leftarrow$ REV(SELS($e_v$))             |
     | $sels_w \leftarrow$ REV(SELS($e_w$))             |
     | return CTAIL($n_v, sels_v, n_w, sels_w$)         | return $\frac{1}{2}$
16:         end if
17:     else
18:         expressions end at different locations
19:         return 0
20:     end if
21: end procedure
```

**Algorithm 3** Common Tails Algorithm

```
 1: procedure CTAIL($nodes_v, sels_v, nodes_w, sels_w$)
 2:     if FIRST($sels_v$) = FIRST($sels_w$)
        $\wedge$ SECOND($nodes_v$) = SECOND($nodes_w$) then
 3:         selector and source equal on both paths...
 4:         if SECOND($nodes_v$) = $n_\emptyset$ then
 5:             ...but sources could be different
 6:             concrete locations, keep looking
 7:             return CTAIL(
                    REST($nodes_v$), REST($sels_v$),
                    REST($nodes_w$), REST($sels_w$))
 8:         else
 9:             ...and both paths have a named node
10:             as common source
11:             return 1
12:         end if
13:     else
14:         reached $n_\emptyset$ via different paths
15:         if IS_SHARED($n_\emptyset$) then
16:             return $\frac{1}{2}$
17:         else
18:             return 0
19:         end if
20:     end if
21: end procedure
```

|  | Algorithm: ORIG |  | Algorithm: CTAIL |  |
|---|---|---|---|---|
|  | $n_\emptyset \notin is$ | $n_\emptyset \in is$ | $n_\emptyset \notin is$ | $n_\emptyset \in is$ |
| with | $\frac{1}{2}$ | $\frac{1}{2}$ | 1 | 1 |
| without | $\frac{1}{2}$ | $\frac{1}{2}$ | 0 | $\frac{1}{2}$ |

$n_\emptyset \in (\notin)\ is$: the summary location is shared (unshared).
*with (without)*: there is a (no) intermediate node from which both expressions share a common tail of selectors.

Figure 2. Alias classification computed by original (ORIG) and our improved algorithm (CTAIL) under different conditions.

there was no common tail, the improved algorithm concludes that the paths end at different concrete locations and hence are not aliased. (ii) If the summary location is *shared* it means that at least one of the concrete nodes represented by $n_\emptyset$ has two (or more) incoming edges originating at a heap node. From the analysis information, however, it is not recoverable whether the paths end at one of the shared or unshared concrete nodes represented by $n_\emptyset$, so $\frac{1}{2}$ must be the answer whenever expressions end at the shared summary location without a "common tail".

Figure 2 compares the results obtained by the original (ORIG) and our improved alias computation algorithm (CTAIL). The "common tail" extension improves what can be said about pointer expression pairs in all cases, except when the expressions reach the *shared* summary location without a "common tail".

## III. Implementation

The shape analyses and alias computation algorithms were implemented for a subset of C++ using the SATIrE program analysis framework.

The data flow analyzers were built using PAG [8], a tool for generating analyzers from high-level functional specifications. The source-to-source architecture used for the analysis was ROSE [9]. These systems were integrated using the SATIrE static program analysis framework being developed at Vienna University of Technology[1] and University of Applied Sciences Technikum Wien[2]. SATIrE is described in [10]. Recent releases of SATIrE[3] include the implementation of the shape analyses as examples.

In their original formulation [4] [6], the algorithms were formulated for an intra-procedural language. We extended the algorithms to be used inter-procedurally for a subset of C++ that is focused on pointer expressions operating on the

[1]`http://www.tuwien.ac.at/`
[2]`http://www.technikum-wien.at/`
[3]SATIrE 0.8.5 available at `http://www.complang.tuwien.ac.at/satire/`

heap.

The shape analyses are used to demonstrate the extraction of alias information from shape graphs and to experimentally compare the precision of their abstractions. Within a compiler, the shape analyses would be used in conjunction with other alias analyses focusing on the various other C++ constructs that could possibly introduce aliasing. Our implementation specifically does not cover aliasing introduced by the following language constructs, which therefore lie outside our language subset:

- Stack-based aliasing introduced by expressions of the form `x = &y`. The shape analyses do not support the address-of operator.
- Aliasing that is the product of pointer arithemtic.
- Aliasing introduced through the use of references.
- Aliasing introduced by unions and anonymous unions.

A detailed description of the implementation itself and the C++ subset that it covers is given by Pavlu [7].

## IV. Evaluation

To evaluate the precision of the algorithms, we analyzed C++ procedures that operate on linked lists.[4] The same set of list operations was used as benchmark in Rinetzky and Sagiv [11]. The programs 'insert' and 'reverse' previously analyzed by Sagiv, Reps & Wilhelm [5] were also added to our benchmark programs.

For every list operation we have two programs: Each operation is applied to linked lists created by unrolled code (*nb*), and to lists created in a loop (*lp*). The *nb* case does not contain branches; the shape analysis only needs to approximate the list creation for a single program execution path. In the *lp* case, infinitely many execution paths need to be approximated.

The analyzed C++ procedures and running times on a single core of a quad-core Intel® Xeon® E5450 (12MB Cache, 3.00 GHz, 1333 MHz FSB, 24GB RAM) machine using 8GB RAM running Linux (Debian 4.3.2-1.1, Kernel 2.6.30-perfctr) are listed in Tables II and III. Running times were averaged over five runs to guard against measuring side-effects.

### A. Precision: Original vs. Common Tails Algorithm

The columns labeled 0, $\frac{1}{2}$ and 1 in Table II show the number of unaliased pointer expressions, *strict* may-aliases and must-aliases, respectively. The column labeled $N$ shows the number of possible alias pairs over all expressions for all statements in the program.

If no aliases are found in a program, $N$ equals the size of the no-alias set. Generally, the set sizes for 0, $\frac{1}{2}$ and 1 add up to $N$.

In SRW shape graphs each edge is possibly null, i. e., nonexistent. No must-alias information can therefore be extracted from SRW shape graphs, so the 1-column shows 0 occurrences for all benchmarks analysed with SRW.

Note that CTAIL provides a more precise classification of aliases for each of our benchmarks, i. e., some pointer expression pairs that ORIG identifies as strict may-aliases can be more precisely identified as not aliased or must-aliases by our "common tails" algorithm. The number of strict may-aliases is thus reduced; columns $\delta_{srw}$ and $\delta_{nnh}$ of Table II give the factor by which the set size of strict may-aliases can be turned into more precise results when using CTAIL. The better the shape analysis results (cf. next section), the greater the gain of replacing ORIG with CTAIL: while CTAIL improves strict may-alias classifications in SRW by a factor ranging from 1.14 to 2.03, the improvement for NNH is in the range of 1.31 to 5.05.

Note also that every pointer expression pair identified by ORIG as 0 or 1 can also be found in the respective set computed by CTAIL. CTAIL is always at least as precise as ORIG, and often succeeds in computing a more precise classification of aliasing in pointer expressions.

### B. Precision: SRW vs. NNH

Using the number of strict may-aliases obtained with the CTAIL algorithm as measure of analysis precision we argue that the NNH shape analysis is more precise ($\Delta_{\text{CTAIL}}$) than the SRW shape analysis; in our benchmarks, NNH is more precise by a factor of 1.62 on average, as can be seen in Table I.

When comparing SRW and NNH by means of may-aliases (instead of strict may-aliases) extracted from shape graphs by the original algorithm, the shape analyses show different levels of precision in only a small number of benchmarks, and when they do, it's only by a small amount: a factor of 1.03 on average.

Table I
Comparison of SRW and NNH shape analysis results based on may- and strict may-aliases. $\Delta_{\text{CTAIL}}$ ($\Delta_{\text{ORIG}}$) is the factor that NNH is more precise than SRW using the CTAIL (ORIG) algorithm.

| Bench | May ($1 \cup \frac{1}{2}$) | | Strict May ($\frac{1}{2}$) | |
|---|---|---|---|---|
| | $\Delta_{\text{ORIG}}$ | $\Delta_{\text{CTAIL}}$ | $\Delta_{\text{ORIG}}$ | $\Delta_{\text{CTAIL}}$ |
| delall$_{nb}$ | 1.00 | 1.32 | 1.07 | 1.52 |
| delall$_{lp}$ | 1.00 | 1.34 | 1.01 | 1.38 |
| insert$_{nb}$ | 1.00 | 1.22 | 1.05 | 1.33 |
| insert$_{lp}$ | 1.00 | 1.23 | 1.01 | 1.25 |
| remove$_{nb}$ | 1.02 | 1.23 | 1.09 | 1.50 |
| remove$_{lp}$ | 1.02 | 1.26 | 1.03 | 1.26 |
| search$_{nb}$ | 1.03 | 2.21 | 1.08 | 2.68 |
| search$_{lp}$ | 1.04 | 2.32 | 1.05 | 2.40 |
| append$_{nb}$ | 1.00 | 1.33 | 1.13 | 1.77 |
| append$_{lp}$ | 1.00 | 1.34 | 1.01 | 1.36 |
| merge$_{nb}$ | 1.16 | 1.31 | 1.21 | 1.39 |
| merge$_{lp}$ | 1.00 | 1.23 | 1.00 | 1.24 |
| reverse$_{nb}$ | 1.05 | 1.61 | 1.10 | 1.91 |
| reverse$_{lp}$ | 1.05 | 1.64 | 1.06 | 1.67 |
| Average | 1.03 | 1.47 | 1.06 | 1.62 |

[4]All benchmarks available at:
`http://www.complang.tuwien.ac.at/vpavlu/`

Note that every pointer expression pair identified as unaliased using SRW shape analysis results is also unaliased in the information extracted from NNH shape graphs. But NNH can also identify additional pointer expression pairs as unaliased or even as must-aliased. NNH is strictly more precise than SRW for all of our benchmarks.

### *C. Running Time*

As can be seen in Figure 3, overall runtime of shape analysis and alias extraction does not increase significantly when replacing ORIG with CTAIL analyzing NNH shape graphs. And for SRW shape graphs the cost added by CTAIL is even below the precision of our measurements (column "$p_1$" in Table III), as in two cases (search$_{nb}$, append$_{lp}$) this cost turns out to be negative. Table III also gives the absolute running time in seconds for the shape analyses (columns labeled "shape") and alias extraction algorithms (columns labeled "alias").

Extracting the aliasing information from shape graphs is expensive already in the original algorithm. This is due to the approach of querying all pointer expression pairs avaialable in a program for their aliasing at every program point. Let $|\mathbf{Exp}_\star|$ be the number of pointer expressions in the given benchmark, and $|\mathbf{Stmt}_\star|$ the number of statements, then the number of considered alias pairs $N$ is:

$$N = \frac{|\mathbf{Exp}_\star|^2}{2} \cdot |\mathbf{Stmt}_\star|$$

Note that the differences in running time of the shape analyses (comparing the columns labeled "shape" in ORIG and CTAIL, cf. Table III) is, again, due to measuring imprecisions only, as the underlying shape analysis algorithm is in no way altered when the alias computation – performed as separate post-processing pass – is performed using CTAIL instead of ORIG.

Switching the underlying shape analysis from SRW to NNH, however, increases running time significantly. Not only does the NNH shape analysis take longer ($P_{sh}$ in Table III), but the alias computation is also much more time-consuming ($P_{sh+al}$ in Table III). This is a result of the vast number of shape graphs contained in the shape graph sets used by NNH. The choice between ORIG and CTAIL, however, is negligeable here.

## V. Related Work

A vast amount of work has been published on pointer analysis, Hind [12] gives a survey of the field. Our work compares two of the most precise shape analysis algorithms known [5], [6] for their relative quality. We do this by comparing alias pairs extracted from shape graphs.

Shapiro and Horwitz [13] compared pointer analyses in combination with subsequent analyses that rely on alias analysis' results. It was found that using a more precise pointer analysis (Andersen's) not only leads in general to "transitively" more precise results, i. e., more precise results of the subsequent analysis, but also causes the client analysis to run faster.

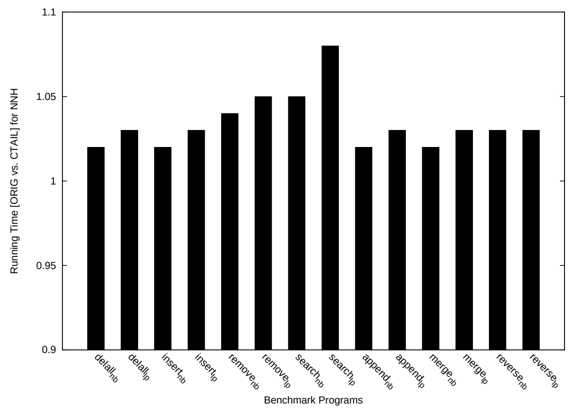

Figure 3. Running times of CTAIL performed on NNH shape graphs (cf. Column $p_2$ in Table III), scaled to the running times of ORIG.

Our method of comparison also uses alias information as required by following optimization passes in a compiler. The proposed method can be applied to all pointer analyses from which aliasing information can be extracted. We are therefore able to compare the precision of shape analyses to that of pointer analyses and give an overview of related work in the fields of alias- and shape analysis.

Emami et. al. [14] suggest that the aliasing problem for statically allocated data (typically on the stack) and dynamically allocated data (typically on the heap) should be decoupled. Analysis of pointers in statically allocated data is easier because the set of locations in static memory is finite, known at compile time, and is usually already named (variables). Analysis of heap-directed pointers is complicated by the fact that locations in the heap don't have names and recursive data structures give rise to a theoretically unbounded number of locations. It has been shown [15], [16] that the may-alias and must-alias problems are undecidable for programs with dynamic storage and recursive data structures. The must-alias problem is not even recursively enumerable for the same class of programs. An algorithm trying to solve the aliasing problem must therefore come up with a good approximation that also ensures termination.

The simplest form of alias analysis uses an "address-taken" approach: all pointers are said to alias with all variables whose address "was taken" in the program, i. e. all variables that the address-of operator (`&`) has been applied to. This also includes all heap-allocated objects. While this analysis is very simple and linear in the size of the program, it is also very imprecise as it uses a single solution set ( [17], [18]).

Steensgaard [19] describes a points-to analysis that takes almost linear time but is substantially more precise than the address-taken analysis [18]. Steensgaard's algorithm uses a type system to describe the store: the type of a variable represents locations possibly pointed to by the variable. Types of memory locations that may be pointed to by the same pointer are unified (merged). This kind of analysis is therefore also called *unification-* or *equality-based* pointer analysis. It is one of the fastest algorithms for finding aliases but is less precise than other flow-insensitive analyses.

Andersen [20] extracts subset constraints from a program reflecting which locations must be included in the points-to set of which variable. The set of constraints is then solved. Algorithms based on this technique are called *Andersen-style*, *inclusion-* or *subset-based* pointer analyses. Andersen's algorithm does not perform the merging found in Steensgaard's algorithm. Inclusion-based analyses are the most precise flow-insensitive, context-insensitive pointer alias analyses.

Recent work on points-to analysis either tries to make unification-based (Steensgaard) analyses more precise or inclusion-based (Andersen) analyses more efficient. Manuvir Das [21] gives a hybrid algorithm that is almost as precise as Andersen's algorithm but scales almost as well as Steensgaard's on real programs. Hardekopf and Lin [22] considerably improve the efficiency of the state of the art in inclusion-based analyses ( [23]–[25]) by introducing two online cycle detection techniques. Online cycle detection looks for cycles in the constraint graph and collapses their components into single nodes to reduce complexity.

Results in [17], [18] suggest that flow-sensitivity alone does not offer much gain in precision over Andersen-style flow-insensitive analyses. For context-sensitive analyses without flow-sensitivity the results are not as clear. According to Michael Hind [12] and Foster et. al. [26] context-sensitivity brings little or no improvements to Andersen-style analyses, but can be beneficial for simpler equality-based analyses.

The algorithm described in [27], [28] combines context-sensitivity by cloning and an explicit heap model with an efficient flow-insensitive unification-based algorithm called data structure analysis (DSA).

Jones and Muchnick [29] were the first to study the shape analysis problem for languages with destructive updating. Each program point has attached a set of finite shape graphs to model the heap. The same concept of sets of shape graphs is also used in NNH [6], but the mechanism to make the shape graphs finite differs: Jones and Muchnick [29] limit paths in their shape graphs to a fixed length $k$ ($k$-limiting), while NNH [6] uses the naming scheme also found in SRW [4], [5] that labels nodes with the names of those variables that directly point to them. As the number of variables in a program is finite it follows that the number of named nodes in a shape graph using this naming scheme is also finite. All heap locations not directly pointed to by a variable give rise to an additional node called the summary node.

Due to the possible exponential blow-up in $k$-limited graphs, a small $k$ is often chosen; beyond $k$ no information is retained and conservative assumptions have to be made. To remedy this problem Chase et. al. [30] do not use $k$-limiting, but summarize shape graph nodes in different summary nodes according to their allocation site instead. They follow Jones and Muchnick [31] in that they also use a single shape graph instead of sets of shape graphs. Their algorithm is able to perform strong updates only under certain conditions.

The analysis of Sagiv, Reps, and Wilhelm [4], [5] was the first shape analysis to always perform strong updates for the elementary transfer functions. Their labelling scheme accounts for a worst case graph size of $2^{|\mathbf{Var}_\star|}$ ($\mathbf{Var}_\star$ is the set of variables in the program), whereas in NNH [6] sets of shape graphs are used to represent the heap, further increasing the number of shape graph nodes stored at each program point to $2^{2^{|\mathbf{Var}_\star|}}$ in the worst case. In previous sections we showed how these two algorithms can be evaluated w. r. t. computed alias pair sets.

Deutsch [32] uses access paths to abstract the structure of the heap. Instead of $k$-limiting the paths, regular expressions are used to make them finite. It is not clear whether Deutsch [32] or SRW [4] produce more precise results.

Sagiv, Reps, and Wilhelm present a parametric framework for shape analyses [33], [34] based on 3-valued logic [35]. Instantiations of this framework use 3-valued logical structures instead of shape graphs to carry the analysis information. The authors claim that use of this framework brings several advantages: the abstract semantics are easier to derive from the concrete semantics and there is no need for a proof as the soundness of all instantiations of the framework follows from the single Embedding Theorem. But instrumentation predicates required for the instantiation need to be defined and proven correct: "It is open to debate whether these are more or less burdensome tasks than those one faces with more standard approaches to abstract interpretation." [34, p. 278]. Previous work on shape analysis (i. e., graph-based shape analyses) could be implemented as instantiation of the 3-valued logic analysis (TVLA) framework, but some of these algorithms are more efficient than instantiations of the framework would be [34, p. 279]. Recent advances [2] show that the run-time of TVLA based analyses can be further improved. Another approach is based on separation logic and analyzes each procedure independently of its caller [1], increasing the potential to scale.

## VI. Conclusion

We have presented a novel algorithm for the extraction of alias information from shape graphs. Instead of looking only at the final node of two pointer expressions, our algorithm takes a sequence of selectors at the end of two pointer expressions into account to decide their alias relation. This "common tails" algorithm is significantly more precise than the previously known method [3] to extract alias information from shape graphs.

We argued in favor of separating must-aliases from the traditional notion of may-aliases. Expression pairs for which only the conservative answer (that the expressions possibly alias) can be given, should be isolated in a separate category that we called strict may-aliases.

For a standard set of graph-based shape analysis benchmarks we have observed that our "common tails" algorithm accounts for improved precision by a factor of 1.21 to 5.05 over the original algorithm, as measured by a reduction of the strict may-alias set.

Our work then experimentally compared two of the most precise shape analysis algorithms for their relative quality. The goal was to determine a measure of precision for a given program which allows to rate the quality of one shape analysis in relation to the other. As program property we selected the aliasing information that is available in shape graphs of these two analyses, as many compiler optimizations also depend on aliasing information.

The experimental results allowed us to derive a novel measure of precision for each benchmark, showing that NNH is always more precise than SRW for each of our benchmarks. The size of strict may-alias sets differs by a factor of 1.62 on average for our benchmarks.

We also presented measurements of the run-times of the SRW- and NNH analysis, as well as for the computation of the alias information. The measurements showed that we can determine the precision factor used for our comparison within reasonable time.

The proposed technique of comparing shape analyses by the size of derived strict may-alias sets can be applied to all kinds of pointer analyses that allow the extraction of alias information. It is therefore also possible to compare the precision of graph-based shape analyses to logic-based shape analyses or even to points-to analyses.

## References

[1] C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang, "Compositional shape analysis by means of bi-abduction," *SIGPLAN Notices*, vol. 44, no. 1, pp. 289–300, 2009.

[2] I. Bogudlov, T. Lev-Ami, T. W. Reps, and M. Sagiv, "Revamping TVLA: Making parametric shape analysis competitive," in *CAV*, ser. Lecture Notes in Computer Science (LNCS), W. Damm and H. Hermanns, Eds., vol. 4590. Springer, 2007, pp. 221–225.

[3] T. W. Reps, M. Sagiv, and R. Wilhelm, "Shape analysis and applications," in *The Compiler Design Handbook: Optimizations and Machine Code Generation*, Y. N. Srikant and P. Shankar, Eds. CRC Press, 2002, pp. 175–218.

[4] M. Sagiv, T. W. Reps, and R. Wilhelm, "Solving shape-analysis problems in languages with destructive updating," in *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA: ACM, 1996, pp. 16–31.

[5] ——, "Solving shape-analysis problems in languages with destructive updating," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 20, no. 1, pp. 1–50, January 1998.

[6] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of Program Analysis*. Springer, 1999, ch. Shape Analysis, pp. 102–129.

[7] V. Pavlu, "Shape-based alias analysis for object-oriented languages," Master's thesis, TU Wien, Department of Computer Science, Vienna, Austria, 2009.

[8] F. Martin, "PAG – an efficient program analyzer generator," *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 1, pp. 46–67, 1998.

[9] M. Schordan and D. Quinlan, "A source-to-source architecture for user-defined optimizations," in *Joint Modular Languages Conference*, 2003.

[10] M. Schordan, "Combining tools and languages for static analysis and optimization of high-level abstractions," in *24. Workshop der GI-Fachgruppe "Programmiersprachen und Rechenkonzepte"*. Department of Computer Science, Christian-Albrechts-Universität zu Kiel, 2007, pp. 72–81.

[11] N. Rinetzky and M. Sagiv, "Interprocedural shape analysis for recursive programs," *Lecture Notes in Computer Science*, vol. 2027, pp. 133–149, 2001.

[12] M. Hind, "Pointer analysis: haven't we solved this problem yet?" in *PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. New York, NY, USA: ACM, 2001, pp. 54–61.

[13] M. Shapiro and S. Horwitz, "The effects of the precision of pointer analysis," in *Proceedings of the 4th International Symposium on Static Analysis*. Springer-Verlag, 1997, pp. 16–34.

[14] M. Emami, R. Ghiya, and L. J. Hendren, "Context-sensitive interprocedural points-to analysis in the presence of function pointers," in *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, 1994, pp. 242–256.

[15] W. Landi, "Undecidability of static analysis," *Letters on Programming Languages and Systems (LOPLAS)*, vol. 1, no. 4, pp. 323–337, 1992.

[16] G. Ramalingam, "The undecidability of aliasing," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 16, no. 5, pp. 1467–1471, 1994.

[17] M. Hind and A. Pioli, "Evaluating the effectiveness of pointer alias analyses," *Science of Computer Programming*, vol. 39, no. 1, pp. 31–55, 2001.

[18] ——, "Which pointer analysis should I use?" in *ISSTA '00: Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*. New York, NY, USA: ACM, 2000, pp. 113–123.

[19] B. Steensgaard, “Points-to analysis in almost linear time,” in *POPL ’96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA: ACM, 1996, pp. 32–41.

[20] L. O. Andersen, “Program analysis and specialization for the c programming language,” Ph.D. dissertation, University of Copenhagen, May 1994.

[21] M. Das, “Unification-based pointer analysis with directional assignments,” *SIGPLAN Notices*, vol. 35, no. 5, pp. 35–46, 2000.

[22] B. Hardekopf and C. Lin, “The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code,” *SIGPLAN Notices*, vol. 42, no. 6, pp. 290–299, 2007.

[23] N. Heintze and O. Tardieu, “Ultra-fast aliasing analysis using CLA: a million lines of C code in a second,” *SIGPLAN Notices*, vol. 36, no. 5, pp. 254–263, 2001.

[24] D. J. Pearce, P. H. J. Kelly, and C. Hankin, “Efficient field-sensitive pointer analysis for C,” in *PASTE ’04: Proceedings of the 5th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. New York, NY, USA: ACM, 2004, pp. 37–42.

[25] M. Berndl, O. Lhoták, F. Qian, L. Hendren, and N. Umanee, “Points-to analysis using BDDs,” in *PLDI ’03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*. New York, NY, USA: ACM, 2003, pp. 103–114.

[26] J. S. Foster, M. Fähndrich, and A. Aiken, “Polymorphic versus monomorphic flow-insensitive points-to analysis for C,” in *SAS ’00: Proceedings of the 7th International Symposium on Static Analysis*. London, UK: Springer-Verlag, 2000, pp. 175–198.

[27] C. Lattner and V. Adve, “Data structure analysis: An efficient context-sensitive heap analysis,” University of Illinois at Urbana-Champaign, Tech. Rep., 2003.

[28] C. Lattner, A. Lenharth, and V. Adve, “Making context-sensitive points-to analysis with heap cloning practical for the real world,” in *PLDI ’07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*. New York, NY, USA: ACM, 2007, pp. 278–289.

[29] N. D. Jones and S. S. Muchnick, “Flow analysis and optimization of lisp-like structures,” in *POPL ’79: Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. New York, NY, USA: ACM, 1979, pp. 244–256.

[30] D. R. Chase, M. Wegman, and F. K. Zadeck, “Analysis of pointers and structures,” in *PLDI ’90: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*. New York, NY, USA: ACM, 1990, pp. 296–310.

[31] N. D. Jones and S. S. Muchnick, “A flexible approach to interprocedural data flow analysis and programs with recursive data structures,” in *POPL ’82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA: ACM, 1982, pp. 66–74.

[32] A. Deutsch, “Interprocedural may-alias analysis for pointers: beyond k-limiting,” in *PLDI ’94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*. New York, NY, USA: ACM, 1994, pp. 230–241.

[33] M. Sagiv, T. W. Reps, and R. Wilhelm, “Parametric shape analysis via 3-valued logic,” in *POPL ’99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA: ACM, 1999, pp. 105–118.

[34] M. Sagiv, T. Reps, and R. Wilhelm, “Parametric shape analysis via 3-valued logic,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 24, no. 3, pp. 217–298, May 2002.

[35] F. Nielson, H. R. Nielson, and M. Sagiv, “Kleene’s logic with equality,” *Information Processing Letters*, vol. 80, no. 3, pp. 131–137, 2001.

Table II
ALIAS SETS COMPUTED FROM SHAPE ANALYSIS RESULTS. $\delta_{srw}$ ($\delta_{nnh}$) IS THE FACTOR BY WHICH CTAIL HAS FEWER CONSERVATIVE RESULTS ($\frac{1}{2}$) THAN ORIG FOR SRW (NNH).
$N$ IS THE NUMBER OF POTENTIAL ALIAS PAIRS IN THE GIVEN BENCHMARK.

| | | SRW | | | | | | | NNH | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Bench | $N$ | ORIG | | | CTAIL | | | $\delta_{srw}$ | ORIG | | | CTAIL | | | $\delta_{nnh}$ |
| | | 0 | $\frac{1}{2}$ | 1 | 0 | $\frac{1}{2}$ | 1 | | 0 | $\frac{1}{2}$ | 1 | 0 | $\frac{1}{2}$ | 1 | |
| delall$_{nb}$ | 8602 | 8407 | 195 | 0 | 8479 | 123 | 0 | 1.59 | 8407 | 183 | 12 | 8509 | 81 | 12 | 2.26 |
| delall$_{lp}$ | 5313 | 5139 | 174 | 0 | 5196 | 117 | 0 | 1.49 | 5139 | 172 | 2 | 5226 | 85 | 2 | 2.02 |
| insert$_{nb}$ | 13338 | 13054 | 284 | 0 | 13146 | 192 | 0 | 1.48 | 13054 | 271 | 13 | 13181 | 144 | 13 | 1.88 |
| insert$_{lp}$ | 8775 | 8512 | 263 | 0 | 8589 | 186 | 0 | 1.41 | 8512 | 261 | 2 | 8624 | 149 | 2 | 1.75 |
| remove$_{nb}$ | 18270 | 17469 | 801 | 0 | 17780 | 490 | 0 | 1.63 | 17485 | 732 | 53 | 17873 | 326 | 71 | 2.25 |
| remove$_{lp}$ | 15225 | 14331 | 894 | 0 | 14621 | 604 | 0 | 1.48 | 14351 | 872 | 2 | 14744 | 479 | 2 | 1.82 |
| search$_{nb}$ | 9600 | 9295 | 305 | 0 | 9450 | 150 | 0 | 2.03 | 9305 | 283 | 12 | 9532 | 56 | 12 | 5.05 |
| search$_{lp}$ | 5700 | 5443 | 257 | 0 | 5556 | 144 | 0 | 1.78 | 5453 | 245 | 2 | 5638 | 60 | 2 | 4.08 |
| append$_{nb}$ | 12285 | 12007 | 278 | 0 | 12099 | 186 | 0 | 1.49 | 12007 | 246 | 32 | 12145 | 105 | 35 | 2.34 |
| append$_{lp}$ | 7722 | 7465 | 257 | 0 | 7542 | 180 | 0 | 1.43 | 7465 | 255 | 2 | 7588 | 132 | 2 | 1.93 |
| merge$_{nb}$ | 24180 | 22987 | 1193 | 0 | 23131 | 1049 | 0 | 1.14 | 23152 | 988 | 40 | 23377 | 754 | 49 | 1.31 |
| merge$_{lp}$ | 15345 | 14733 | 612 | 0 | 14839 | 506 | 0 | 1.21 | 14733 | 610 | 2 | 14935 | 408 | 2 | 1.50 |
| reverse$_{nb}$ | 14364 | 14031 | 333 | 0 | 14179 | 185 | 0 | 1.80 | 14046 | 303 | 15 | 14249 | 97 | 18 | 3.12 |
| reverse$_{lp}$ | 9450 | 9138 | 312 | 0 | 9271 | 179 | 0 | 1.74 | 9153 | 295 | 2 | 9341 | 107 | 2 | 2.76 |
| Average | | | | | | | | 1.55 | | | | | | | 2.43 |
| Overall | | 162011 | 6158 | 0 | 163878 | 4291 | 0 | 1.44 | 162262 | 5716 | 191 | 164962 | 2983 | 224 | 1.92 |

Table III
RUNNING TIME (IN SECONDS) OF SHAPE-BASED ALIAS ANALYSES.
$p_1$ ($p_2$) IS THE FACTOR BY WHICH CTAIL RUNS LONGER (SHAPE ANALYSIS+ALIAS COMPUTATION) THAN ORIG FOR SRW (NNH).
$P_{sh}$ ($P_{sh+al}$) IS THE FACTOR THAT NNH SHAPE ANALYSIS (+ALIAS COMPUTATION WITH CTAIL) RUNS LONGER THAN SRW.

| | SRW | | | | | NNH | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Bench | ORIG | | CTAIL | | $p_1$ | ORIG | | CTAIL | | $p_2$ | $P_{sh}$ | $P_{sh+al}$ |
| | shape | alias | shape | alias | | shape | alias | shape | alias | | | |
| delall$_{nb}$ | 0.03 | 0.46 | 0.03 | 0.47 | 1.02 | 0.05 | 8.32 | 0.04 | 8.51 | 1.02 | 1.33 | 17.10 |
| delall$_{lp}$ | 0.02 | 0.29 | 0.02 | 0.30 | 1.03 | 0.12 | 29.46 | 0.11 | 30.47 | 1.03 | 5.50 | 95.56 |
| insert$_{nb}$ | 0.02 | 0.80 | 0.02 | 0.80 | 1.00 | 0.06 | 19.41 | 0.06 | 19.82 | 1.02 | 3.00 | 24.24 |
| insert$_{lp}$ | 0.03 | 0.53 | 0.02 | 0.53 | 1.00 | 0.20 | 70.60 | 0.20 | 72.39 | 1.03 | 10.00 | 131.98 |
| remove$_{nb}$ | 0.03 | 1.34 | 0.03 | 1.35 | 1.01 | 0.32 | 22.63 | 0.33 | 23.46 | 1.04 | 11.00 | 17.24 |
| remove$_{lp}$ | 0.02 | 1.10 | 0.02 | 1.10 | 1.00 | 0.42 | 95.54 | 0.42 | 99.91 | 1.05 | 21.00 | 89.58 |
| search$_{nb}$ | 0.03 | 0.71 | 0.04 | 0.70 | 0.99 | 0.07 | 10.12 | 0.08 | 10.62 | 1.05 | 2.00 | 14.46 |
| search$_{lp}$ | 0.03 | 0.42 | 0.02 | 0.42 | 1.00 | 0.27 | 32.11 | 0.28 | 34.59 | 1.08 | 14.00 | 79.25 |
| append$_{nb}$ | 0.03 | 0.71 | 0.03 | 0.72 | 1.01 | 0.05 | 9.73 | 0.05 | 9.93 | 1.02 | 1.67 | 13.31 |
| append$_{lp}$ | 0.02 | 0.46 | 0.03 | 0.45 | 0.98 | 0.12 | 31.03 | 0.11 | 31.96 | 1.03 | 3.67 | 66.81 |
| merge$_{nb}$ | 0.73 | 5.50 | 0.72 | 5.50 | 1.00 | 10.56 | 577.47 | 10.55 | 586.85 | 1.02 | 14.65 | 96.05 |
| merge$_{lp}$ | 0.03 | 1.01 | 0.03 | 1.01 | 1.00 | 5.95 | 478.92 | 5.94 | 491.12 | 1.03 | 198.00 | 477.94 |
| reverse$_{nb}$ | 0.03 | 0.87 | 0.02 | 0.88 | 1.01 | 0.10 | 23.60 | 0.11 | 24.26 | 1.03 | 5.50 | 27.08 |
| reverse$_{lp}$ | 0.02 | 0.58 | 0.02 | 0.58 | 1.00 | 0.37 | 74.44 | 0.37 | 76.97 | 1.03 | 18.50 | 128.90 |