FAKULTÄT FÜR !NFORMATIK

Shape-Based Alias Analysis for Object-Oriented Languages

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering / Internet Computing

eingereicht von

Viktor Pavlu

Matrikelnummer 0425543

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuer: Dipl.-Ing. Dr.techn. Markus Schordan

Wien, 23.09.2009

(Unterschrift Verfasser)

(Unterschrift Betreuer)

Viktor Pavlu
Mühlbachergasse 10
1130 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, September 2009

Viktor Pavlu

Abstract

Shape analysis is a static code analysis technique discovering properties of linked data structures allocated in the heap. It is typically used at compile time to find software bugs or to verify high-level correctness properties.

The aim of this work is to evaluate merits and drawbacks of parametrized versions of two shape analyses: the SRW analysis, as described by "Solving shape-analysis problems in languages with destructive updating" (M. Sagiv, T. Reps, R. Wilhelm), which was the first to achieve strong update of pointer values for statements that modify pointer values. The second analysis, termed NNH, is based on the former but uses sets of shape graphs as abstraction instead of merging the analysis information into summary shape graphs. This shape analysis is described in the book "Principles of Program Analysis" (F. Nielson, H.R. Nielson, C. Hankin).

To facilitate evaluation, both analyses were implemented for C++ using the SATIrE program analysis framework. An algorithm that computes a finite conservative set of may-aliases from a given shape graph has been developed. The precision of the shape analysis algorithms is measured by the size of extracted may-alias sets, where smaller sets indicate a more precise shape analysis. Experimental results show the relative quality of parametrized versions of both shape analyses and the impact that the parameters have on precision and runtime.

Zusammenfassung

Eine Shape Analyse ist eine statische Programmanalyse, die Eigenschaften verketteter Datenstrukturen im dynamischen Speicher untersucht. Üblicherweise wird sie zur Übersetzungszeit eingesetzt um Programmeigenschaften zu überprüfen und Fehler zu finden.

Ziel dieser Arbeit ist es parametrisierte Versionen von zwei Shape Analysen zu vergleichen: die SRW Analyse, wie sie in "Solving shape-analysis problems in languages with destructive updating" (M. Sagiv, T. Reps, R. Wilhelm) beschrieben wird. Sie war die erste Shape Analyse, die für Programmknoten die Zeigerwerte ändern *strong updates* durchführen konnte. Die zweite Analyse, in Folge NNH genannt, basiert auf der vorigen Analyse, verwaltet ihre Information zu jedem Programmpunkt aber in einer Menge von Shape Graphen anstatt die Analyseinformation in einem einzelnen Zusammenfassungsgraphen zu halten. Diese Analyse ist in dem Buch "Principles of Program Analysis" (F. Nielson, H.R. Nielson, C. Hankin) beschrieben.

Für den Vergleich wurden beide Analysen für C++ mit dem SATIrE Programmanalyse Framework implementiert. Ein Algorithmus der eine endliche konservative Menge von May-Aliasen aus einem gegebenem Shape Graphen berechnen kann wurde entwickelt. Die Genauigkeit der beiden Analysen wird an der Qualität der May-Alias Mengen gemessen, die aus den Ergebnissen der Shape Analysen extrahiert werden, wobei kleinere Mengen ein genaueres Ergebnis der Shape Analyse anzeigen. Ergebnisse von Experimenten zeigen die relative Qualität von parametrisierten Versionen beider Shape Analysen und die Auswirkung der Parameter auf Genauigkeit und Analyselaufzeit.

Contents

Introduction **10**

1 Shape Analysis **11**
1.1 Terminology and Background 13
1.2 Shape Analysis in this Work 16

2 Analysis Information **17**
2.1 Program Representation 17
2.2 Objects in Memory 18
2.3 Concrete Representation of Memory 20
2.4 Shape Graphs 21
2.5 Abstract Representation of Memory 23
2.6 Abstract Shape Graphs 24
2.7 Shape Graph Conversions 29

3 Transfer Functions **38**
3.1 NNH Intra-Procedural Transfer Functions 39
3.2 SRW Intra-Procedural Transfer Functions 46
3.3 Transfer Function Wrappers 46
3.4 Normalization of Pointer Expressions 47
3.5 Inter-Procedural Transfer Functions 49

4 Three-Valued Alias Computation from Shape Graphs **52**
4.1 Extraction of the Alias Information 53
4.2 Improved Test with Common Tails 57
4.3 Algorithm 59

5 Implementation **64**
5.1 SATIrE 64
5.2 Shape Analysis Implementation 69
5.3 Visualization 71
5.4 Alias Information 72

5.5 Shape Analyses and C++ 72

6 Evaluation **80**
6.1 Shape Analysis Variations 80
6.2 Alias Analysis Variations 87
6.3 Measurements 90

7 Related Work **95**
7.1 Flow-Insensitive Context-Insensitive 96
7.2 Flow- or Context-Sensitive 97
7.3 Flow-Sensitive Context-Sensitive 98
7.4 Other related work 100

8 Perspectives and Conclusions **101**
8.1 Speed Improvements 101
8.2 Precision and Coverage Improvements 102
8.3 Conclusions 104

Bibliography **105**

List of Tables **113**

List of Figures **113**

Index **115**

Introduction

Use of pointers is common, especially in object-oriented languages like C++ and Java. Ignoring pointers and structures in the heap considerably reduces what can be recovered with any analysis, as every analysis must at least ensure a conservative approximation to the side-effects that pointer operations have.

```
1: a->val = 42;
2: b->val = 0;
3: x = a->val;
```

Figure 1: A short example C++ program with pointers.

Example 1 Consider the program in Fig. 1. Without information about the aliasing of `a` and `b` no constant propagation may conclude that what is assigned to `x` in line 3 was not overwritten in the second statement. Only if `a->val` and `b->val` are known not to alias, i.e. only if they refer to different memory locations, it is certain that the assignment in line 2 does not overwrite the value in `a->val`. Then `x` will be set to 42 in line 3. If `a->val` and `b->val` are guaranteed to be aliases it can be concluded that `x` is set to 0 in line 3. Under these conditions, the precise knowledge about pointers in the program can be used to determine the value of `x` already at compile time. ■

Elimination of unnecessary instructions or the replacement of one sequence of instructions with a faster one that does the same thing is the task of program optimization. Returning to our example Ex. 1, the program could be optimized using the information on pointer structure: if `a->val` and `b->val` are known to refer to the same memory location, the first assignment in line 1 (cf. Fig. 1) is immediately overwritten in line 2 without use in between, so the first assignment could be eliminated; if the expressions are known to never

alias `a->val` and `b->val` are independent of each other and the assignment in line 2 could be moved to any other point in the program without changing the semantics of the code, enabling parallelization and possibly helping with register allocation, for example.

Shape analysis is a static program analysis technique that approximates the structure of the runtime heap. Using this conservative heap representation various questions about linked data structures allocated in the heap can be answered at compile time. This work concentrates on what was for some the original motivation for shape analyses: providing better may-alias information.[1] Aim of this work is to compare parametrized versions of two shape analyses by the size of may-alias sets that are extracted from shape analysis results. Smaller sets of possible aliases are more precise and indicate a more precise underlying shape analysis. We are thus able to experimentally evaluate precision and analysis runtime tradeoffs for parametrized versions of two shape analysis algorithms that differ in the way they represent the heap.

The algorithm described by [SRW98] uses a single graph for each program point, called static shape graph, while the algorithm in [NNH99] uses sets of compatible shape graphs. The set-oriented approach has several advantages; it is conceptually simpler and the analysis results are more precise than what can be found in SRW static shape graphs. The disadvantage lies in the maximum number of shape nodes per program point: SRW static shape graph sizes are exponential in the number of program variables as their nodes are labelled with subsets of variables, i.e. the set of labels is the powerset of $\mathbf{Var}_\star$, the set of variables found in the program. The maximum number of nodes in an SRW static shape graph is therefore $|\mathcal{P}(\mathbf{Var}_\star)| = 2^{|\mathbf{Var}_\star|}$.

The NNH analysis, on the other hand, uses sets of compatible shape graphs with the same naming scheme. The worst case graph size of a single shape graph is then again $2^{|\mathbf{Var}_\star|}$, but this time there could be any subset of such graphs associated with a single program point. The total number of shape nodes per program point in the NNH analysis is therefore $2^{2^{|\mathbf{Var}_\star|}}$.

Example 2 The C++ function in Fig. 2 destructively reverses the acyclic list pointed to by `x` into an acyclic list pointed to by `y` by traversing its elements and changing the direction of each `next`-pointer.

The effect that the ReverseList program has on the heap when actually executed on a four-element acyclic list is shown in Fig. 3 for certain program points before, during, and after reversal of the list. At the beginning of each

[1] "The goal of providing better may-alias information was the motivation for our work that grew into shape analysis"[RSW02, p.12-4].

```
01: List* ReverseList(List* x) {
02:   List* y;
03:   List* t;
04:   y=0;
05:   while(x!=0) {
06:     t=y;
07:     y=x;
08:     x=x->next;
09:     y->next=t;
10:   }
11:   t=0;
12:   return y;
13: }
```

Figure 2: The ReverseList C++ function.

iteration `x` and `y` point to the heads of the unreversed and reversed list, respectively. The old value of `y` is copied to the temporary variable `t` while `y` is made to point to the head of the original list pointed to by `x`. `x` is then moved to the next element of the unreversed list. The final step of each iteration is to detach `y`, the head of the unreversed list, from the remaining elements and append it to the end of the reversed list by making its `next`-field point to `t`, thus "turning" the pointer and reversing the order of elements in the list. This is done until the unreversed list's head is `null`. ■

Using shape analyses we compute from the program's source code the conservative approximation of the heap structure for the general case of an n-element acyclic list. The SRW shape analysis keeps a conservative representation of the heap in SRW static shape graphs associated with every program point and updates these shape graphs with transfer functions according to the pointer-modifying statements found in the source code. The NNH shape analysis works on the same principle but uses a different abstraction and different transfer functions.

The SRW static shape graph representation as computed by the SRW shape analysis is, for the same program points as before, shown in Fig. 4. The sets of compatible shape graphs produced by the NNH shape analysis can be seen in Fig. 5. Note that the union of shape graphs contained in the three NNH shape graph sets are equal to the three SRW static shape graphs.

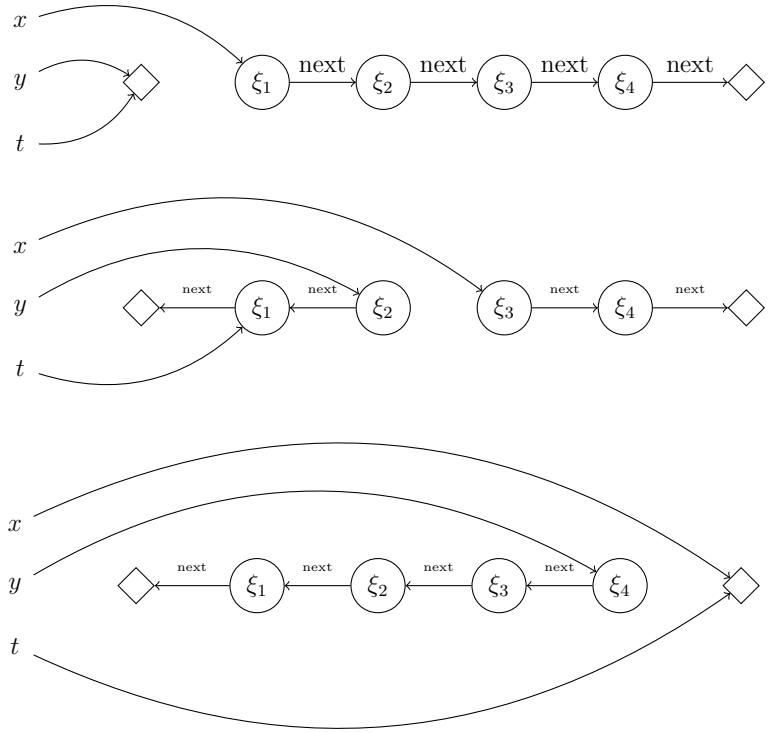


Figure 3: Concrete heap during execution of ReverseList: before (after line 6, iteration 1), during (after line 5, iteration 3), and after reversal (after line 11).

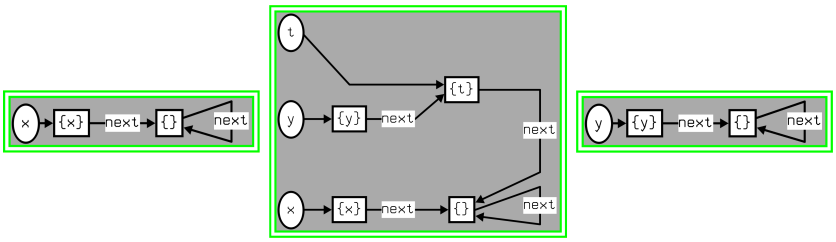


Figure 4: Abstract heap represented by SRW static shape graphs as computed by the SRW analysis and illustrated using our visualization for aiSee: before, during, and after reversal.

Utilizing either heap approximation produced by the shape analyses we can make statements about the aliasing of pointer expressions that refer to addresses in the heap. Two expressions constitute a may-alias pair at program

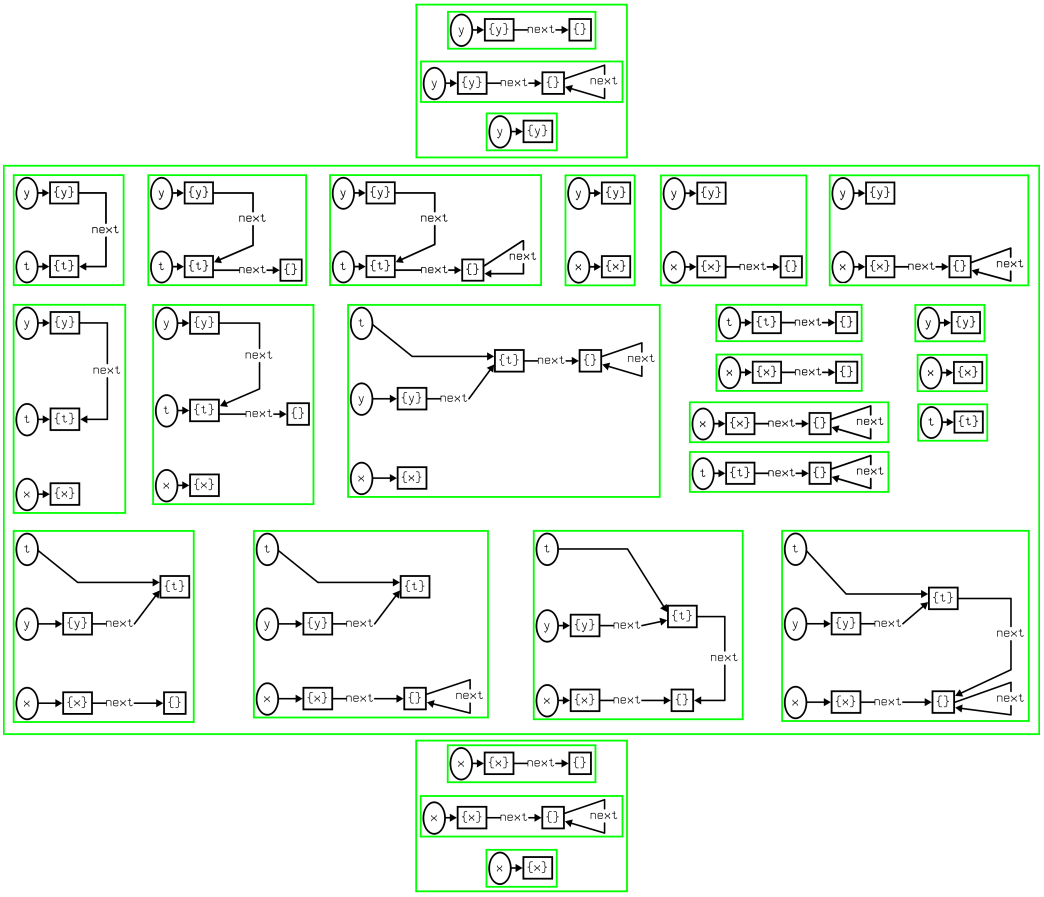


Figure 5: Abstract heap represented by NNH sets of compatible shape graphs as computed by the NNH analysis and illustrated using our visualization for aiSee: before, during, and after reversal.

point ℓ if they describe access paths to the same heap location in at least one execution path of a program that ends at ℓ. Accordingly, two expressions constitute a must-alias pair at program point ℓ if they describe access paths to the same heap location in *all* execution paths of a program that ends at ℓ. At compile time we are thus able to identify pairs of pointer expressions that will refer to the same runtime memory location.

The aliasing information is made available to other program analyses as attributes in the abstract syntax tree from where it may also be fed back to the programmer as annotations to the input program. A program with may-alias annotations generated by our implementation is shown in Fig. 6. The may-

alias pairs immediately before and after each statement are automatically added to the source code as comments.

By looking at the different shape graphs we may not be able to decide which analysis produces more precise results. Converting the NNH shape graph set into a static shape graph even yields an SRW static shape graph that is equal to that obtained directly from the SRW analysis. By comparing the size of may-alias sets extracted from the shape graphs, however, we are able to see that the NNH shape analysis produces a slightly more precise result than SRW as the may-alias set computed for the ReverseList function is smaller when derived from the NNH analysis results.

The metric of may-alias set size is therefore used in the experimental comparison of parametrized SRW- and NNH shape analysis variations.

```
// pre may-aliases : (t,y -> next),
while(x != ((0))) {
  // pre may-aliases : (t,y -> next),
  t = y;
  // post may-aliases : (y,t),

  // pre may-aliases : (y,t),
  y = x;
  // post may-aliases : (y,x), (y -> next,(x -> next)),

  // pre may-aliases : (y,x), (y -> next,(x -> next)),
  x = (x -> List::next);
  // post may-aliases : (x,y -> next),

  // pre may-aliases : (x,y -> next),
  y -> List::next = t;
  // post may-aliases : (t,y -> next),
}
// post may-aliases : (t,y -> next),
```

Figure 6: Part of function ReverseList. May-alias information is computed by our alias analysis and added to the program's source code as annotations.

Structure of This Work

This thesis comprises two main parts. The first deals with the theoretical aspects of shape analyses in general and the merits of the SRW and NNH analyses in particular. The extraction of alias sets from shape analysis results is also presented. The second part treats the practical implementation of parametrized versions of both analyses and discusses the parameters' impact on the precision and runtime behaviour of an alias analysis based on these shape analyses.

In the theoretical part, Chapter 1 gives an introduction to shape analyses and then puts the shape analysis by Sagiv, Reps and Wilhelm (SRW), and the shape analysis described by Nielson, Nielson and Hankin (NNH) in context within the broader field of pointer analysis. Then the analysis information carrier (cf. Ch. 2) and the transfer functions (cf. Ch. 3) are described. Chapter 4 concludes the theoretical part with a description of how alias information is computed from summary shape graphs (SRW) or compatible shape graph sets (NNH). An improved version of the alias computation algorithm given in [RSW02] is also presented in Chapter 4.

The part describing the implementation of the analyses begins with Chapter 5 which introduces the software tools used in the practical work: the source-to-source transformation framework ROSE, the data flow analyzer generator PAG, and the SATIrE system for integrating program analysis and transformation tools. This chapter continues by dealing with issues that arise when applying the shape analyses to C++, and specifying what restrictions are imposed on the input language. Chapter 6 discusses how the analyses compare to each other under varying analysis parameters. The discussion is backed with measurements of shape analysis variants described in this work.

Chapter 7 gives an overview of related work. Chapter 8 concludes by hinting at further improvements to the analyses and then provides a summary of the main points and results.

Acknowledgements

First and foremost, I would like to thank Markus Schordan for his supervision and the inspiring talks we had during my work on the SATIrE project and the preparation of this thesis. The discussions at the early stage of the project contributed a lot to my interest in program analysis and compiler construction.

I would also like to thank my colleagues Gergö Bárány and Adrian Prantl for collaborating on the SATIrE project.

Several people read earlier drafts of this work and made important comments. I am especially indebted to Andreas Bolka whose suggestions greatly helped in improving presentation.

The work on the thesis was in part supported by a studentship granted by the Vienna University of Technology. Work on SATIrE has been partially funded by the ARTIST2 Network of Excellence (http://www.artist-embedded.org/).

Chapter 1

Shape Analysis

Shape analysis is a static program analysis technique that discovers a wide range of properties of linked data structures allocated in the heap. It is closely related to pointer alias analysis but has its focus on the heap with dynamically allocated, recursive data structures and works with explicit heap models that are more powerful than alias- or points-to sets. Shape analysis is typically used at compile time to find software bugs or to verify high-level correctness properties of a program.

The information a shape analysis can recover is valuable for debugging, optimization, program understanding, parallelization, instruction scheduling or compile time garbage collection, to name but a few uses. Information about the structure of the heap greatly improves the accuracy of a wide range of other analyses as the heap is an extensive source for aliases in a program.

With the results of a shape analysis, these questions ([RSW02]) can be answered:

Null pointers: Does a pointer contain `null` at the entry to a statement that dereferences the pointer?

Always: Report an error about the certain `null` dereference.

Sometimes: Warn about the potential `null` dereference.

Never: Eliminate check for `null`.

Aliases: Do two pointer expressions refer to the same location at a program point?

Always: This constitutes a must-alias.

Sometimes: This constitutes a may-alias.

Never: The expressions are not aliased.

Knowing about aliases enhances the precision of other analyses, improves program dependence information, and is necessary for instruction reordering or parallelizing transformations.

Sharing: Is there more than one way to access a heap cell?

Sometimes: Warn about explicit deallocation when there are still references to the cell ("dangling pointers").

Never: Deallocate the heap cell when the last pointer to it ceases to exist.

Reachability: Is a heap cell reachable from a specific variable or any pointer variable?

Always: Use this information for program verification.

Never: Insert code at compile time that collects unreachable cells at runtime.

Disjointness: Do two data structures ever have common elements?

Never: Disjoint data structures may be distributed to different processors for computation.

Cyclicity: Is a heap cell part of a cycle?

Never: Garbage collection could be implemented by reference counting.

Memory leaks: Does a procedure or program leave behind unreachable heap cells when it returns?

Sometimes: Issue a warning about the memory leak.

Structure: Does a program fragment leave the shape of a data structure intact?

Knowing whether a singly linked list, potentially with (or definitely without) a cycle remains a singly linked list after the execution of a procedure helps with program understanding, debugging and program verification.

1.1 Terminology and Background

Shape analysis is a special form of pointer analysis. The terms *pointer analysis*, *alias analysis*, and *points-to analysis* are but sometimes used interchangeably [Hin01]. It is therefore necessary that we clarify our understanding of these terms:

Pointer Analysis The goal of pointer analysis is to discover properties of pointer values in a program. It is used as the general term for all analyses that deal with pointer values and includes all of the following analyses.

Shape Analysis Shape analysis investigates the effects that statements have on the structure, or shape, of the heap. Detailed information about the runtime heap structure is useful in many areas (cf. Sect. 1), including analysis of the heap-induced aliasing.

Alias Analysis The purpose of an alias analysis is to discover expressions that refer to the same storage location and thus are aliases of each other. In C++ there are also other sources of aliases besides pointers on the stack or in the heap, e.g. call-by-reference parameters, references, and unions. The precise results obtained from a shape analysis are useful for alias analysis problems.

Points-To Analysis Points-to analysis is a pointer analysis that computes sets of locations an expression may point to. This is different from the alias sets used in alias analysis. It avoids redundancy and prevents inconsistent configurations. Points-to analysis results are used to answer aliasing questions.

1.1.1 Static vs. Dynamic

Static program analysis is performed without actually executing the program by deriving information from a program's representation (usually source code or object code), e.g. by means of abstract interpretation of the program according to its semantics.

Static program analysis in general and the may-alias and must-alias problems for programs with dynamic storage and recursive data structures in particular have been shown to be undecidable. [Lan92; Ram94]

A static program analysis therefore has to work with a good approximation that also ensures termination. Static analysis can be performed at compile time or link time.

Dynamic program analysis gets additional information about a program by executing it. The behaviour of all but the most trivial programs depends on their inputs, so the program must be executed sufficiently often with test inputs that reflect typical usage. In the form of a profiler, dynamic program analysis is used to identify program parts critical to its performance.

1.1.2 Flow-insensitive vs. Flow-sensitive

A program analysis that ignores the control-flow of the program is called flow-insensitive analysis. It treats the program as unordered set of statements and computes a single solution per function, or the entire program.

Flow-sensitive analysis respects the order of statements. A forward analysis uses the forward edges of the control flow graph, a backward analysis uses the backward edges which can be obtained by swapping source and destination of all forward edges. Analyses that consider the control flow must compute a result for every point in the program. They are therefore more costly, especially in the presence of loops, but they can also be more precise.

1.1.3 Intra-procedural vs. Inter-procedural

An intra-procedural analysis is limited to what can be found within a procedure. Procedure calls are neglected for this kind of analysis. Intra-procedural analysis is also called *monovariant* or *sticky* [And94, p. 118].

An inter-procedural analysis takes the control- and information-flow between procedure calls and called procedures into account, increasing both power and complexity of analysis. Inter-procedural analysis is also called *polyvariant*.

1.1.4 Context-insensitive vs. Context-sensitive

Inter-procedural analysis can be further divided by the precision of their procedure call semantics.

Context-sensitivity determines whether an analysis models the fact that separate invocations of a function are independent of each other. Analyses that are not context-sensitive merge the analysis information of all calling con-

texts when analysing a procedure. The result at the end of the procedure is then returned to all the call sites.

A context-sensitive analysis respects the differences between call sites of a procedure. Procedures are analysed separately for every calling context and the analysis information at the end of procedures is only returned to the corresponding call site. Clearly, this is more precise than a context-insensitive analysis, but also more costly.

One method for storing context information is the use of call strings: at every procedure call the label of the call site is added to the call string. This encodes the path taken through the program and allows to tell call sites apart. If the language of the input program supports recursion, the call strings can become arbitrarily large and need to be truncated to ensure termination of the analysis.

1.1.5 Weak update vs. Strong update

In a weak update, new analysis information is only ever added to the information that was previously associated with a program point.

A strong update not only adds to the analysis information but replaces parts of it. For example, the shape analyses presented here model an assignment to a pointer in two steps: first, all the old bindings of the pointer are removed (strong nullification), then the new bindings are established.

1.1.6 May vs. Must

The certainty of an alias can be characterized to be *may* or *must*. When two expressions are identified as must-aliases at a program point, they are guaranteed to refer to the same memory location, no matter which execution path leads to this program point. Must information is also known as *universal* information.

When two expressions are identified as may-aliases at a program point, there exists at least one execution path leading to that point, in which the expressions are made to point to the same storage location. May-aliases are therefore also known as *existential* aliases.

1.2 Shape Analysis in this Work

For the practical part of this thesis, a flow-sensitive, context-sensitive inter-procedural alias analysis was developed. It uses the analysis results obtained from either the SRW or the NNH shape analysis as input to solve may- and must-alias problems for all pointer-expressions found in a program.

1.2.1 Sagiv-Reps-Wilhelm Analysis

The shape analysis described in [SRW98] is a static, flow-sensitive intra-procedural data flow analysis that uses finite graphs to approximate the possible structure of the heap. It is presented within the framework of abstract interpretation.

The shape analysis by Sagiv, Reps and Wilhelm was the first shape analysis to achieve strong updates for languages with destructive updating. It will be referred to as SRW analysis throughout this work. Sect. 3.5 illustrates how the SRW shape analysis was extended for inter-procedural work.

Shape graphs used in SRW have their nodes labelled with sets of variables that directly point to them. All heap locations not directly pointed to by a variable are coalesced into one additional node called the abstract summary location. This ensures the finite size of SRW static shape graphs. The analysis information carrier will be dealt with in detail in Sect. 2.6.

1.2.2 Nielson-Nielson-Hankin Analysis

A modified shape analysis based on the work of Sagiv, Reps and Wilhelm [SRW96; SRW98] is presented in [NNH99]. The shape analysis by Nielson, Nielson and Hankin will be referred to as NNH analysis throughout this work.

NNH is also a static intra-procedural data flow analysis, but uses sets of compatible shape graphs instead of a single static shape graph for the analysis information. This modification makes it not only better suited as textbook example, but also improves the precision of the analysis while introducing an exponential factor in the cost of the analysis (cf. Sect. 2.6.2).

Chapter 2

Analysis Information

The two shape analyses compared in this work are both given in terms of graph-based memory abstractions and a set of transfer functions that manipulate these shape graphs. The transfer functions are discussed in Sect. 3. In this chapter we define the memory abstractions and show how the SRW and NNH shape graphs are related.

A number of preliminary definitions are needed. Closely following the notation in [NNH99] we use $\mathbf{S}_\star$ to represent the program that we are analysing, $\mathbf{Lab}_\star$ to represent the labels used to identify the statements in $\mathbf{S}_\star$, $\mathbf{Var}_\star$ to represent the variables used in $\mathbf{S}_\star$, and $\mathbf{Exp}_\star$ to represent the expressions in $\mathbf{S}_\star$.

2.1 Program Representation

Definition 1 (Flow Graph)**.** *A* flow graph *is a tuple* $G = (N, E, n_s, n_e)$ *where* N *is a finite set of nodes and* $E \subseteq N \times N$ *is a set of edges. The nodes* $n_s, n_e \in N$ *are the unique* start *(or* initial*) and* end *(or* final*) nodes which are assumed to have no incoming or outgoing edges, respectively. A flow graph with directed edges is a* directed flow graph*.*

Definition 2 (Predecessor, Successor)**.** *For a flow graph* (N, E, n_s, n_e) *we denote the set of* immediate predecessors *and* successors *of a node* n *by* $pred(n) \stackrel{def}{=} \{m \mid (m, n) \in E\}$ *and* $succ(n) \stackrel{def}{=} \{m \mid (n, m) \in E\}$*.*

Definition 3 (Path)**.** *A* path *of length* $k-1$ *in a flow graph* $G = (N, E, n_s, n_e)$ *is a sequence of nodes* $\langle n_1, \ldots, n_k \rangle$ *such that* $(n_i, n_{i+1}) \in E$ *for all* $1 \leq i < k$*.*

A *control flow graph (CFG)* is a directed flow graph that makes the control flow of an intra-procedural program fragment explicit. All nodes of a control flow graph lie on paths from n_s to n_e. CFGs can either be node-labelled or edge-labelled, i.e. the statements of a program can either be associated with the nodes or the edges of the CFG; in terms of granularity they can be basic-block graphs or single-instruction graphs.

We will represent programs as control flow graphs where nodes are associated with program states and edges are associated with the transitions between them. Edges are thus labelled with single program statements or not labelled at all (i.e. labelled with the empty pseudo-statement *skip*), which corresponds to a branch. Control flow graphs like these are called edge-labelled single-instruction CFGs. Compared to the traditional node-labelled basic-block CFGs used in the majority of compiler literature, where nodes represent basic blocks, i.e. maximal sequences of straight-line code, and edges represent the branching structure, this representation simplifies various program analysis tasks.[KKS98]

To model programs that consist of several functions we use *Inter-procedural Control Flow Graphs* (ICFGs) that contain a CFG for every procedure. Function call sites and called functions are connected with special edges that allow the passing of analysis information, increasing both power and complexity of the analysis.

2.2 Objects in Memory

The C++ Standard [Int03] defines an *object* as a region of storage that has a type and a storage duration which influences its lifetime. Functions and references, regardless of whether or not they occupy storage in the way that objects do, are not objects [Int03, 1.8 intro.object, 5.3.4 expr.new].

Objects are created by a definition or by a new-expression [Int03, 1.8 intro.object].

Creating objects by definitions is referred to as *static memory allocation* (cf. Fig. 2.1) because the size of the memory being allocated can be decided at compile time by looking at the text of the program only. These objects either have *static storage duration*, which means that the storage lasts for the duration of the program [Int03, 3.7.1 basic.stc.static], or *automatic storage duration* which ends as soon as the block in which the object was created exits [Int03, 3.7.2 basic.stc.auto].

For languages that use procedures it is common to manage at least part of their runtime memory as a *stack*. Each time a procedure is called, space for its local variables is reserved on the stack, and when the procedure terminates, that space is popped off the stack again [ALSU07, pp. 429ff]. Objects created by definitions are allocated in the stack area of memory.[1]

```
List *p;
List l();
```

Figure 2.1: A pointer-to-list object and a list object are created by definitions.

The other way to create objects is through the use of a new-expression (cf. Fig. 2.2). This is called *dynamic memory allocation* as it allows objects to be created at program runtime. These objects have *dynamic storage duration* which means that they may outlive the call to the procedure that created the object. Dynamically allocated objects last until explicitly destroyed using delete-expressions. They are therefore not kept on the stack but in a separate region of runtime memory called the *heap*[1] area of reusable memory [ALSU07, p. 452].

```
p = new List();
```

Figure 2.2: A list object is created using a new-expression.

Besides definitions and new-expressions, an object can also be created by the implementation as temporary object when needed. In terms of storage duration temporary objects behave like those created through a definition [Int03, 3.7 basic.stc].

An *array* is an object that contains a contiguously allocated non-empty set of N objects of the same type [Int03, 8.3.4 dcl.array].

All objects have a location or address that uniquely determines where they are located in memory. For arrays the address of the first element is used as the address of the array [Int03, 5.3.4 expr.new].

[1] An implementation is free to choose where it allocates objects as long as the storage duration requirements of the C++ Standard [Int03, 3.7 basic.stc] are met. The stack region of memory for automatic objects and the heap region for dynamic objects is but the obvious choice.

A *pointer* is an object that contains the address of an object. A pointer is said to *point to*, or refer to, the object whose address it contains.

A *reference* can be thought of an additional name for an object. There are no references to references, pointers to references or arrays of references, as a reference is not an object. It is unspecified whether or not a reference requires storage [Int03, 8.3.2 dcl.ref].

Stack and heap regions of runtime memory will together be referred to as the *store*.

2.3 Concrete Representation of Memory

To model heap allocated data structures we introduce a set **Loc** of locations or addresses in the heap:

$$\xi \in \mathbf{Loc} \qquad \textit{concrete locations, or addresses in the heap}$$

Every cell that is allocated in the heap is uniquely defined by its address ξ, such that $\xi_i \neq \xi_j$ for all $\xi_i, \xi_j \in \mathbf{Loc}, i \neq j$.

Cells in the heap may contain values as well as pointers to other cells. The fields stored in a cell are accessed through selectors:

$$\mathit{sel} \in \mathbf{Sel} \qquad \textit{selector names, like } \texttt{car} \textit{ or } \texttt{cdr}$$

The concrete state mapping function σ binds variables $v \in \mathbf{Var}_\star$ to their value. A value is either an elementary type (**Integer**, **Float**, ...), an aggregate type (**Class**, **Union**, **Enum**, ...), a pointer to a location in the heap (**Loc**), or the special constant $\diamond$ representing the *nil*-value (i.e. a null-pointer). The same is valid for the fields in a cell. For the shape analysis only pointer values are relevant, so the other types a value can have are only hinted at in the definition. The concrete heap mapping function $\varkappa$ binds fields to their values:

$$\begin{array}{rcl} \sigma \in \mathbf{StateF} & = & \mathbf{Var}_\star \to \mathbf{Value} \\ \varkappa \in \mathbf{HeapF} & = & (\mathbf{Loc} \times \mathbf{Sel}) \hookrightarrow \mathbf{Value} \\ \mathbf{Value} & = & (\mathbf{Integer} + \mathbf{Float} + \ldots + \mathbf{Enum} + \mathbf{Loc} + \{\diamond\}) \end{array}$$

Note that not all selector fields need to be defined. For a newly created cell ξ with yet uninitialized fields the heap will have $\varkappa(\xi, \mathit{sel})$ to be undefined for all selectors $\mathit{sel} \in \mathbf{Sel}$.

The pointer-induced structure of the store can be described by a shape graph.

2.4 Shape Graphs

The definition of the shape graph concept (cf. Definition 4) is taken from [SRW98], p. 13 but given in the notation used throughout this work which follows [NNH99]:

Definition 4 (Shape Graph)**.** *A* shape graph *is a directed graph of finite size that consists of two kinds of nodes,* variables *and* shape nodes*, and two kinds of edges,* variable edges *and* selector edges*. The set of shape graphs is denoted by* $\mathcal{SG}$*.*

A shape graph $SG \in \mathcal{SG}$ is represented by a pair of edge sets $(\mathsf{E_v}, \mathsf{E_s})$:

$$\begin{array}{rcl} \mathsf{E_v} \in \mathbf{VarEdges} & = & \mathcal{P}(\mathbf{Var}_\star \times \mathbf{ShapeNode}) \\ \mathsf{E_s} \in \mathbf{SelEdges} & = & \mathcal{P}(\mathbf{ShapeNode} \times \mathbf{Sel} \times \mathbf{ShapeNode}) \end{array}$$

The variable edges $\mathsf{E_v}$ are sometimes also called *stack edges* as the variables are allocated on the stack, similarly, the selector edges are also called *heap edges*.

Obviously there cannot be isolated nodes in a shapegraph, as only pointers that refer to locations are recorded by the shape graph. It is therefore sufficient that nodes are given implicitly as the endpoints of edges. For a shape graph SG the shape nodes $shape\_nodes(SG)$ can be recovered from the edges:

$$\begin{aligned} shape\_nodes(SG) \stackrel{\text{def}}{=} & \{n \mid (*, n) \in E_v\} \\ & \cup \{n \mid (n, *, *) \in E_s\} \\ & \cup \{n \mid (*, *, n) \in E_s\} \end{aligned}$$

Following [SRW98] we use the symbol E_v to mean the function that, when applied to a variable x returns the shape nodes which are pointed to by x in the shape graph: $E_v(x) \stackrel{\text{def}}{=} \{n \mid (x, n) \in \mathsf{E_v}\}$. Similarly, for a shape node n and selector sel we define $E_s(n, sel) \stackrel{\text{def}}{=} \{m \mid (n, sel, m) \in \mathsf{E_s}\}$.

Definition 5 (Deterministic Shape Graph)**.** *A shape graph* $(\mathsf{E_v}, \mathsf{E_s})$ *is* deterministic *if*

- *every variable points to at most one shape node,*
  $x \in \mathbf{Var}_\star, |E_v(x)| \leq 1$*, and*

– *every shape node and selector points to at most one shape node,* $n \in \mathbf{ShapeNode}, sel \in \mathbf{Sel}, |E_s(n, sel)| \leq 1$.

The set of deterministic shape graphs is denoted by $\mathcal{DSG}$, $\mathcal{DSG} \subseteq \mathcal{SG}$.

This common concept of shape graph is the basis for the different shape graphs used for the concrete and abstract semantics of the SRW and NNH analyses. [SRW98] use deterministic shape graphs $\mathcal{DSG}$ (cf. Sect. 2.4.1) and static shape graphs $\mathcal{SSG}$ (cf. Sect. 2.6.1) for the concrete and abstract semantics, respectively. In [NNH99] compatible shape graphs $\mathcal{CSG}$ (cf. Sect. 2.6.2) are used to model the abstract semantics.

2.4.1 Concrete Representation with Shape Graphs

We represent the actual (or concrete) structure of a store using deterministic shape graphs $(\mathsf{S_C}, \mathsf{H_C}) \in \mathcal{DSG}$ that have concrete locations (i.e. heap memory addresses) $\xi \in \mathbf{Loc}$ or the special value *nil* (or $\diamond$), signifying the null-value, as their shape nodes.

$$\begin{array}{rcrcl} \mathsf{S_C} & \in & \mathbf{State} & = & \mathcal{P}(\mathbf{Var}_\star \times \mathbf{Loc}) \cup \mathcal{P}(\mathbf{Var}_\star \times \{\diamond\}) \\ \mathsf{H_C} & \in & \mathbf{Heap} & = & \mathcal{P}(\mathbf{Loc} \times \mathbf{Sel} \times \mathbf{Loc}) \cup \mathcal{P}(\mathbf{Loc} \times \mathbf{Sel} \times \{\diamond\}) \end{array}$$

A variable edge $(x, \xi) \in \mathsf{S_C}$ denotes that `x` is a pointer referring to an object in the heap at the address ξ. Heap edges reflect the connections between objects in the heap. $(\xi_i, sel, \xi_j) \in \mathsf{H_C}$ means that there is a selector field in the object at address ξ_i that has the name *sel* and the field is a pointer that refers to the object at address ξ_j. Note that pointers to *nil* are also part of *concrete* shape graphs.

Example 3 The concrete shape graph representation of a four-element acyclic list pointed to by `lst` is $(\{(lst, \xi_1)\}, \{(\xi_1, next, \xi_2), (\xi_2, next, \xi_3), (\xi_3, next, \xi_4), (\xi_4, next, \diamond)\})$. The same list is illustrated as concrete shape graph in Fig. 2.3. Unlabelled edges are variable edges, heap edges are labelled with their selector. Oval nodes represent concrete cells of the heap and are labelled with their address, $\diamond$ denotes the null-value.

■

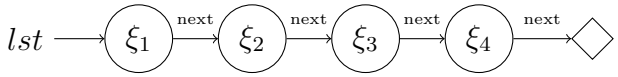


Figure 2.3: A list as concrete shape graph.

2.5 Abstract Representation of Memory

It is evident that there are programs for which the heap can grow arbitrarily large, therefore, for the analysis to be computable, an abstraction is needed that combines the arbitrarily large set of concrete locations (or addresses) **Loc** into a finite number of abstract locations **ALoc**:

$$\mathbf{ALoc} = \{n_X \mid X \subseteq \mathbf{Var}_\star\}$$

Abstract locations are named using a (possibly empty) set of variables. Since the set of variables that actually occur in the program $\mathbf{Var}_\star$ is finite, it follows that the set of abstract locations is also finite.

An abstract location n_X that is labelled with variable-set X represents the concrete location ξ that is simultaneously pointed to by all (and only) the variables in X, i.e. $\sigma(x) = \xi$ for all $x \in X$. If X is non-empty, the abstract location n_X is called a *named* abstract location.

Example 4 If the variables x and y point to the same object, the concrete state σ maps them to the same address (or concrete location) ξ: $\sigma(x) = \sigma(y) = \xi$. The corresponding abstract location is then $n_{\{x,y\}}$ with $\{x, y\}$ as its label. ■

All locations that cannot be reached directly from the state σ without going through the heap $\mathcal{H}$, that is, all locations not directly pointed to by a variable, are collected into a special abstract location called the abstract summary location $n_\emptyset$. As no variables directly point to the summary location it is labelled with the empty set.

Both shape analyses presented in this work use this abstraction to approximate a theoretically infinite number of objects in the heap by a finite representation that is bounded by the number of variables in the program. All cells not directly pointed to by a variable are clustered into a single summary location and therefore only account for one additional shape node. The set of edges originating from variables is called the *abstract state* or *stack* while the edges originating at abstract locations are the *abstract heap*.

Fig. 2.4 illustrates the relation between concrete and abstract representations of memory. Notice that in the concrete representation every variable is mapped to a value by the state σ, while in the abstract representation only pointer values are stored.

Concept	**Code**	**Concrete**	**Abstract**
Location	`new T`	address $\xi \in \mathbf{Loc}$ in the heap	heapnode $n_X \in \mathbf{ALoc}$ in the shape graph
Integer Variable	`i`	$\sigma(i) = 42$	—
Pointer Variable	`x`	$\sigma(x) = \xi$	$(x, n_{\{x\}}) \in \mathsf{S}$
Integer Variable in Heap-Structure	`x->i`	$\sigma(x) = \xi_1$, $\mathcal{H}(\xi_1, i) = 42$	$(x, n_{\{x\}}) \in \mathsf{S}$ —
Pointer Variable in Heap-Structure	`x->sel`	$\sigma(x) = \xi_1$, $\mathcal{H}(\xi_1, sel) = \xi_2$	$(x, n_{\{x\}}) \in \mathsf{S}$ $(n_{\{x\}}, sel, n_Y) \in \mathsf{H}$

Figure 2.4: Stores in concrete and abstract semantics.

2.6 Abstract Shape Graphs

Based on the finite approximation of memory using abstract locations we can define abstract shape graphs:

Definition 6 (Abstract Shape Graph)**.** *An* abstract shape graph *is a pair* (SG, is)*, where* $SG \in \mathcal{SG}$ *is a shape graph with abstract locations as shape nodes, and* is *is a set of abstract locations. The pair* $((\mathsf{S}, \mathsf{H}), \mathsf{is})$ *will be abbreviated as* $(\mathsf{S}, \mathsf{H}, \mathsf{is})$*. The set of abstract shape graphs is denoted by* $\mathcal{ASG}$*.*

An abstract shape graph $(\mathsf{S}, \mathsf{H}, \mathsf{is})$ consists of the abstract state S, an abstract heap H, and a set is of abstract locations that are shared:

$$\begin{array}{rcrcl} \mathsf{S} & \in & \mathbf{AState} & = & \mathcal{P}(\mathbf{Var}_\star \times \mathbf{ALoc}) \\ \mathsf{H} & \in & \mathbf{AHeap} & = & \mathcal{P}(\mathbf{ALoc} \times \mathbf{Sel} \times \mathbf{ALoc}) \\ \mathsf{is} & \in & \mathbf{IsShared} & = & \mathcal{P}(\mathbf{ALoc}) \end{array}$$

The sharing information is is used to decide whether an abstract location is the target of two or more pointers in the heap, or if the abstract location is only pointed to by at most one heap pointer.

When the abstract summary location $n_\emptyset$ points to an abstract location n_X, the explicit sharing information allows to distinguish between the case where only one location in $n_\emptyset$ points to n_X from the cases where many locations in $n_\emptyset$ point to n_X. Similarly it can be recovered whether two (or more) abstract locations pointing to $n_\emptyset$ actually point to distinct locations within the summary node, which then remains unshared, or to the same and therefore shared locations within the summary node. The sharing information thus helps to recover some of the imprecision introduced by clustering all unnamed locations into a single abstract summary location $n_\emptyset$.

Note that the shape graph only records pointers to abstract locations. If a variable or a field has a non-pointer value or is the null-pointer, the variable or field is simply not part of the shape graph. From the perspective of a shape graph it is not possible to discern non-pointer values from uninitialized fields or null-pointers.

Example 5 The abstract shape graph representation of the four-element acyclic list shown in Fig. 2.3 is $(\{(lst, \{lst\})\},\ \{(\{lst\}, next, \{\}), (\{\}, next, \{\})\}, \{\})$. The same list is illustrated as abstract shape graph in Fig. 2.5. Unlabelled edges are variable edges, heap edges are labelled with their selector. Boxed nodes represent abstract heap locations and are labelled with the variables directly pointing to them.

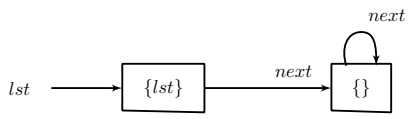


Figure 2.5: A list as abstract shape graph.

■

2.6.1 Analysis Information in SRW

The SRW analysis is specified as a set of transfer functions which operate on abstract shape graphs $SG \in \mathcal{ASG}$ (cf. Sect. 2.6). [SRW98] call their abstract shape graphs *static shape graphs* and denote their set by $\mathcal{SSG}$.

Abstract shape graphs as defined above are nondeterministic: $E_v(x)$ and $E_s(n, sel)$ may each yield a set with more than one abstract location, which means that a pointer actually points to two (or more) locations at the same time. This, of course, cannot be the case in a concrete store, so each of these edges stands for one *possible* pointer value in the concrete store.

SRW static shape graph sizes are exponential in the number of program variables as the abstract heap nodes are labelled with subsets of the variables found in the program, i.e. the set of labels is the powerset of $\mathbf{Var}_\star$. The maximum number of nodes in an SRW static shape graph is therefore $|\mathcal{P}(\mathbf{Var}_\star)| = 2^{|\mathbf{Var}_\star|}$.

2.6.2 Analysis Information in NNH

The analysis information carrier in the NNH analysis is based on the same notion $(\mathsf{S}, \mathsf{H}, \mathsf{is}) \in \mathcal{ASG}$ of abstract shape graphs (cf. Sect. 2.6) as used in the SRW analysis[2], but in an effort to reduce the level of indeterminism the fulfillment of five invariants is demanded:

Invariant 1. *If two abstract locations n_X and n_Y occur in the same shape graph then either $X = Y$ or $X \cap Y = \emptyset$.*

Invariant 2. *If x is mapped to n_X by the abstract state then $x \in X$.*

Invariants 1 and 2 control the labelling of abstract heap nodes. While invariant 2 simply restates what was already specified in the formulation of abstract locations (cf. Sect. 2.5), invariant 1 requires that every variable is included in at most one label, hence the intersection of distinct labels is always empty. This means that there can be no variable pointing to more than one abstract location simultaneously, so the indeterminism of variable edges is eliminated.

Invariant 3. *Whenever (n_V, sel, n_W) and $(n_V, sel, n_{W'})$ are in the abstract heap then either $V = \emptyset$ or $W = W'$.*

[2]Both, the static shape graphs of SRW and the compatible shape graphs of NNH are represented as $(\mathsf{S}, \mathsf{H}, \mathsf{is})$ in this work. Where the type of shape graph is not clear from the context the shape graphs will be labelled $(\mathsf{S}^{\mathcal{S}}, \mathsf{H}^{\mathcal{S}}, \mathsf{is}^{\mathcal{S}})$ and $(\mathsf{S}^{\mathcal{C}}, \mathsf{H}^{\mathcal{C}}, \mathsf{is}^{\mathcal{C}})$, respectively.

Invariant 3 ensures that the heap edges are deterministic for all named nodes, i.e. no selector *sel* of a named location points to two locations simultaneously. $|E_s(n_V, sel)| \leq 1$ for all *sel* and $n_V \neq n_\emptyset$. Indeterminism remains only for heap edges emanating at $n_\emptyset$: The abstract summary location $n_\emptyset$ is allowed to have a selector *sel* point to more than one location at the same time as $n_\emptyset$ represents a potentially unlimited number of concrete heap cells, each with their own selector fields.

Invariant 4. *If* $n_X \in \mathsf{is}$ *then either*

(a) $(n_\emptyset, sel, n_X)$ *is in the abstract heap for some sel, or*

(b) there exists two distinct triples (n_V, sel_1, n_X) *and* (n_W, sel_2, n_X) *in the abstract heap (that is either* $sel_1 \neq sel_2$ *or* $V \neq W$*).*

Invariant 5. *Whenever there are two distinct triples* (n_V, sel_1, n_X) *and* (n_W, sel_2, n_X) *in the abstract heap and* $n_X \neq n_\emptyset$ *then* $n_X \in \mathsf{is}$*.*

The last two invariants keep the explicit sharing information is and the sharing information implicit to the shape graph consistent: If an abstract heap location n_X is explicitly shared $n_X \in \mathsf{is}$, then there must be at least two (concrete) heap edges pointing to n_X; that is, either the abstract summary location points to n_X with one or more selectors, or there are at least two distinct n, sel pairs that point to n_X (Inv. 4).
On the other side, if there are two heap edges pointing to the same named node, then this named node is also included in the explicit sharing information is (Inv. 5).

Abstract shape graphs that meet above requirements (Inv. 1–5) are called *compatible shape graphs*. Their set is denoted by $\mathcal{CSG}$, $\mathcal{CSG} \subseteq \mathcal{ASG}$.

With indeterminism limited to the summary location, the NNH analysis uses sets of compatible shape graphs to represent all possible heap configurations that result from the different execution paths leading to a program point ℓ.

As compatible shape graphs use the same naming scheme as SRW static shape graphs they also share the same worst case graph size of $2^{|\mathbf{Var}_\star|}$ nodes in a single shape graph. But NNH uses sets of compatible shape graphs, so there could be any subset of such graphs associated with a single program point. The total number of shape nodes per program point is therefore $2^{2^{|\mathbf{Var}_\star|}}$.

2.6.3 Taxonomy of Shape Graphs

Fig. 2.6 shows the taxonomy of shape graphs. Derived from the common concept of a shape graph are the concrete and abstract shape graphs that differ in the type of nodes they contain: nodes in a concrete shape graph represent concrete locations labelled with their address ξ_i, while nodes in an abstract shape graph are abstract locations labelled with the set of variables that directly point to them. All abstract locations not directly pointed to by a variable are coalesced into the summary location $n_\emptyset$. This ensures that abstract shape graphs are of finite size with the number of nodes bounded by $2.^{|\mathbf{Var}_\star|}$

Orthogonal to this classification is the division into deterministic and nondeterministic shape graphs. In a deterministic shape graph every variable or field points to at most one location (cf. Sect. 2.4.1) while in nondeterministic shape graphs they may point to multiple locations simultaneously. In practice, however, there are no nondeterministic concrete shape graphs or deterministic abstract shape graphs. Deterministic concrete shape graphs are therefore simply called $\mathcal{DSG}$, and nondeterministic abstract shape graphs are referred to as static shape graphs ($\mathcal{SSG}$) in [SRW98]. A special case of abstract shape graphs are the compatible shape graphs which have five invariants that constrain indeterminism to the summary location. All other locations may only have selectors that point to at most one location, these locations are thus deterministic (cf. Sect. 2.6.2). Static shape graphs ($\mathcal{SSG}$) have no such constraints, $\mathcal{SSG}$ and $\mathcal{ASG}$ therefore contain the same elements. The use of these shape graph representations by the SRW and NNH analysis is shown in Fig. 2.7.

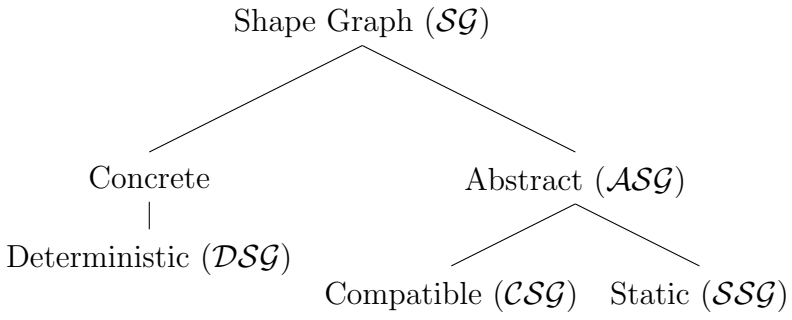


Figure 2.6: Taxonomy of Shape Graphs.

	SRW	**NNH**
concrete modelling	$\mathcal{DSG}$	$\mathcal{DSG}$
abstract modelling	$\mathcal{SSG} = \mathcal{ASG}$	$\mathcal{CSG} \subseteq \mathcal{ASG}$

Figure 2.7: Shape graphs used in SRW and NNH.

2.7 Shape Graph Conversions

After completion of either shape analysis it could be of use to transform static shape graphs into a set of compatible shape graphs or vice versa, due to the their specific advantages: $\mathcal{SSG}$ are compact in representation, while sets of $\mathcal{CSG}$s are more expressive and better suited for post-processing of analysis results, as indeterminism limited to the abstract summary location alleviates the extraction of information from the shape graphs.

2.7.1 $\mathcal{CSG}$-Set to $\mathcal{SSG}$ Conversion

A set of compatible shape graphs $SGS^{\mathcal{C}} = \{SG^{\mathcal{C}}\}$ as used in the NNH analysis can be converted into a single static shape graph $SG^{\mathcal{S}} = (\mathsf{S}^{\mathcal{S}}, \mathsf{H}^{\mathcal{S}}, \mathsf{is}^{\mathcal{S}})$ as used by SRW by forming the union of the shape graphs' components:

$$SGS\_to\_SG(SGS^{\mathcal{C}}) \;=\; SG^{\mathcal{S}} = (\mathsf{S}^{\mathcal{S}}, \mathsf{H}^{\mathcal{S}}, \mathsf{is}^{\mathcal{S}})$$

$$\left.\begin{array}{ll} \mathsf{S}^{\mathcal{S}} & = \bigcup(\mathsf{S}^{\mathcal{C}}) \\ \mathsf{H}^{\mathcal{S}} & = \bigcup(\mathsf{H}^{\mathcal{C}}) \\ \mathsf{is}^{\mathcal{S}} & = \bigcup(\mathsf{is}^{\mathcal{C}}) \end{array}\right\} \text{for all } (\mathsf{S}^{\mathcal{C}}, \mathsf{H}^{\mathcal{C}}, \mathsf{is}^{\mathcal{C}}) \in SGS^{\mathcal{C}}$$

2.7.2 $\mathcal{SSG}$ to $\mathcal{CSG}$-Set Conversion

Conversely, the analysis information computed by the SRW analysis, a static shape graph, can be converted into a set of compatible shape graphs. During this transformation the indeterminism of combining multiple execution paths into a single static shape graph is eliminated as the static shape graph $SG^{\mathcal{S}}$ is split into a set of compatible shape graphs $SGS^{\mathcal{C}} = \{SG^{\mathcal{C}}\}$, each representing one possible configuration of the store. The only form of indeterminism that remains is that of the abstract summary location which is also part of compatible shape graphs and is therefore not eliminated during the conversion.

The transformation is performed in three steps:

1. First, the variable and heap edges are split into their deterministic constituents. This fits the fact that no pointer can point to two addresses simultaneously in a deterministic shape graph. Only selector edges starting at the summary location keep their indeterminism. This is part of the memory abstraction that allows the analyses to represent stores of arbitrary size and is not inhibited by any of the invariants.

   The transformation of the abstract state $\mathsf{S}^{\mathcal{S}}$ into a set of abstract states *States* is the n-ary cartesian product of edge sets $(x_i, n_{X_i}) \in \mathsf{E_v}$ (including the empty set) of all variables $x_i \in \mathbf{Var}_\star$, except that we want the result to be a set, not a tuple. We therefore define the combination operator $\otimes$ as follows:

$$A \otimes B \;=\; \{a \cup b \mid a \in A, b \in B\}$$

   The transformation of the abstract state into its deterministic representatives is thus:

$$\begin{aligned} vars(\mathsf{S}) &\stackrel{\text{def}}{=} \{x \mid (x, n_X) \in \mathsf{S}\} \\ sels(\mathsf{H}) &\stackrel{\text{def}}{=} \{sel \mid (n_X, sel, n_Y) \in \mathsf{H}\} \\ srcs(\mathsf{H}) &\stackrel{\text{def}}{=} \{X \mid (n_X, sel, n_Y) \in \mathsf{H}\} \end{aligned}$$

$$\begin{aligned} var\_edges(x) &= \{\{\}\} \cup \{\{(x, n_X)\} \mid (x, n_X) \in \mathsf{S}^{\mathcal{S}}\} \\ States &= var\_edges(x_0) \otimes \ldots \otimes var\_edges(x_n) \\ &\quad \text{where } x_{0\ldots n} \in vars(\mathsf{S}^{\mathcal{S}}) \end{aligned}$$

   Note that neither of two variable edges $(x, n_{\{x,y\}}), (y, n_{\{x,y\}})$ pointing to the same location in the static shape graph $SG^{\mathcal{S}}$ can be part of a compatible shape graph $SG^{\mathcal{C}}$ if the other edge is not. Either they are together in the shape graph, or they both are not, but clearly there can be no variable edge $(x, n_{\{x,y\}})$ without y pointing to that same abstract location. This is demanded by the naming scheme of abstract locations.

Given that a selector *sel* in a heap cell n can neither refer to more than one heap cell in a compatible shape graph, it is evident that the same transformation as for the variable edges has to be applied to all heap edges that do not start at the summary location, $n \neq n_\emptyset$. Every heap configuration of the resulting set is then combined with all the configurations contained in edges originating at the summary location. Return to mind that the summary location may simultaneously point to several heap nodes even in a compatible shape graph. The set of configurations for edges starting at $n_\emptyset$ is therefore the powerset, while for every named node $n \neq n_\emptyset$ only one target is allowed in a compatible shape graph:

$$\begin{array}{lcl} \mathit{sel\_edges}(X, \mathit{sel}) & = & \{\{\}\} \cup \{\{(n_X, \mathit{sel}, n_Y)\} \mid (n_X, \mathit{sel}, n_Y) \in \mathsf{H}^{\mathcal{S}}\} \\ \mathit{Heaps} & = & \mathit{sel\_edges}(X_0, \mathit{sel}_0) \otimes \ldots \otimes \mathit{sel\_edges}(X_n, \mathit{sel}_m) \otimes \\ & & \mathcal{P}(\mathit{sel\_edges}(n_\emptyset, \mathit{sel}_0)) \otimes \ldots \otimes \mathcal{P}(\mathit{sel\_edges}(n_\emptyset, \mathit{sel}_m)) \\ & & \text{where } X_{0\ldots n} \in \mathit{srcs}(\mathsf{H}^{\mathcal{S}}), \mathit{sel}_{0\ldots m} \in \mathit{sels}(\mathsf{H}^{\mathcal{S}}) \end{array}$$

From these deterministic sets, *States* and *Heaps*, all edge configurations are dropped that violate any of the five compatibility invariants. The result is the set of compatible variable- and heap edge sets, $\mathit{States}^{\mathcal{C}}$ and $\mathit{Heaps}^{\mathcal{C}}$, respectively:

$$\begin{array}{lcl} \mathit{States}^{\mathcal{C}} & = & \{s \mid s \in \mathit{States}, \mathit{is\_compatible}(s)\} \\ \mathit{Heaps}^{\mathcal{C}} & = & \{h \mid h \in \mathit{Heaps}, \mathit{is\_compatible}(h)\} \end{array}$$

Eliminating incompatible state- and heap configurations already at this point is done for efficiency considerations. Every state (or heap) that is incompatible in isolation can only produce incompatible shape graphs when combined with any heap (or state). Eliminating them early thus avoids unnecessary generation of incompatible graphs.

The number of deterministic state edge sets, or states, $|\mathit{States}|$ in a static shape graph $(\mathsf{S}^{\mathcal{S}}, \mathsf{H}^{\mathcal{S}}, \mathsf{is}^{\mathcal{S}})$ is:

$$|\mathit{States}| = \prod_{x_i \in \mathbf{Var}_\star} (1 + \mathit{var\_edges}(x_i))$$

The number of deterministic heap edge sets, or heaps, $|\mathit{Heaps}|$ in a static shape graph $(\mathsf{S}^{\mathcal{S}}, \mathsf{H}^{\mathcal{S}}, \mathsf{is}^{\mathcal{S}})$ is:

$$|Heaps| = \prod_{\substack{x_i \in \mathbf{Var}_\star \\ sel_j \in \mathbf{Sel}}} (1 + heap\_edges(x_i, sel_j)) \ * \prod_{sel_j \in \mathbf{Sel}} 2^{|sel\_edges(n_\emptyset, sel_j)|}$$

2. After dealing with the edges, the sharing information is computed as the powerset of shared heap nodes of the static shape graph $SG^{\mathcal{S}}$. This is necessary because a shared heap node in $SG^{\mathcal{S}}$ may theoretically be unshared in all but one compatible shape graph $SG^{\mathcal{C}}$ in the graph set that resembles the static shape graph $SG^{\mathcal{S}}$. An abstract location that was not shared in $SG^{\mathcal{S}}$ must not be shared in any of the $SG^{\mathcal{C}}$.

$$Sharings^{\mathcal{C}} = \mathcal{P}(\mathsf{is}^{\mathcal{S}})$$

3. The third step is to apply the generate/filter paradigm from above to whole shape graphs: the already compatible sets of variable edges, heap edges and sharing sets are combined to generate the set of shape graphs SGS represented by the single static shape graph $SG^{\mathcal{S}}$, i.e. SGS converted into a static shape graph must result in the original static shape graph $SG^{\mathcal{S}}$:

$$SGS \ = \ States^{\mathcal{C}} \otimes Heaps^{\mathcal{C}} \otimes Sharings^{\mathcal{C}}$$

The number of shape graphs in this set can be very large. Although each of the sets did, in isolation, not violate any of the invariants, many of the resulting shape graphs in SGS are not compatible.

To complete the conversion, all incompatible shape graphs are discarded. This includes the removal of heap edges originating at abstract locations that have no corresponding variable edges. The sharing information also plays a vital role in this process. Every shape graph where explicit sharing information and sharing indicated by the structure of the graph diverge is incompatible and must be discarded.

$$SGS^{\mathcal{C}} \ = \ \{SG \mid SG \in SGS, \ is\_compatible(SG)\}$$

The remaining shape graphs form the set of compatible shape graphs $SGS^{\mathcal{C}}$ that were represented by the static shape graph $SG^{\mathcal{S}}$. Converting $SGS^{\mathcal{C}}$ back into a static shape graph must, again, result in a SG' where $SG' = SG^{\mathcal{S}}$.

$$
\begin{array}{rcl}
SG\_to\_SGS(SG^{\mathcal{S}}) & = & SG\_to\_SGS((\mathsf{S}^{\mathcal{S}}, \mathsf{H}^{\mathcal{S}}, \mathsf{is}^{\mathcal{S}})) \\
SG\_to\_SGS(SG^{\mathcal{S}}) & = & SGS^{\mathcal{C}} \\
SGS^{\mathcal{C}} & = & \{SG \mid SG \in SGS,\ is\_compatible(SG)\} \\
SGS & = & States^{\mathcal{C}} \otimes Heaps^{\mathcal{C}} \otimes Sharings^{\mathcal{C}}
\end{array}
$$

where

$$
\begin{array}{rcl}
var\_edges(x) & = & \{\{\}\} \cup \{\{(x, n_X)\} \mid (x, n_X) \in \mathsf{S}^{\mathcal{S}}\} \\
States & = & var\_edges(x_0) \otimes \ldots \otimes var\_edges(x_n) \\
 & & \text{where } x_{0\ldots n} \in vars(\mathsf{S}^{\mathcal{S}}) \\
States^{\mathcal{C}} & = & \{s \mid s \in States, is\_compatible(s)\}
\end{array}
$$

$$
\begin{array}{rcl}
sel\_edges(X, sel) & = & \{\{\}\} \cup \{\{(n_X, sel, n_Y)\} \mid (n_X, sel, n_Y) \in \mathsf{H}^{\mathcal{S}}\} \\
Heaps & = & sel\_edges(X_0, sel_0) \otimes \ldots \otimes sel\_edges(X_n, sel_m) \otimes \\
 & & \mathcal{P}(sel\_edges(n_\emptyset, sel_0)) \otimes \ldots \otimes \mathcal{P}(sel\_edges(n_\emptyset, sel_m)) \\
 & & \text{where } X_{0\ldots n} \in srcs(\mathsf{H}^{\mathcal{S}}), sel_{0\ldots m} \in sels(\mathsf{H}^{\mathcal{S}}) \\
Heaps^{\mathcal{C}} & = & \{h \mid h \in Heaps, is\_compatible(h)\}
\end{array}
$$

$$
Sharings^{\mathcal{C}} = \mathcal{P}(\mathsf{is}^{\mathcal{S}})
$$

Example 6 Consider the conversion of $SG^{\mathcal{S}} = (\mathsf{S}^{\mathcal{S}}, \mathsf{H}^{\mathcal{S}}, \mathsf{is}^{\mathcal{S}}) =$

$$
\begin{array}{l}
(\quad \{(p, n_{\{p\}}), (x, n_{\{x\}}), (x, n_{\{x,y\}}), (y, n_{\{y\}}), (y, n_{\{x,y\}})\}, \\
\quad \{(n_{\{x\}}, next, n_{\{p\}}), (n_{\{x\}}, next, n_{\{y\}}), (n_{\{y\}}, next, n_{\{p\}}), (n_{\{x,y\}}, next, n_{\{p\}})\}, \\
\quad \{n_{\{p\}}\} \quad)
\end{array}
$$

into a set of compatible shape graphs $SGS^{\mathcal{C}}$ that conservatively represents all shape graphs embodied by the single static shape graph $SG^{\mathcal{S}}$. The shape graph $SG^{\mathcal{S}}$ is illustrated in Fig. 2.8.

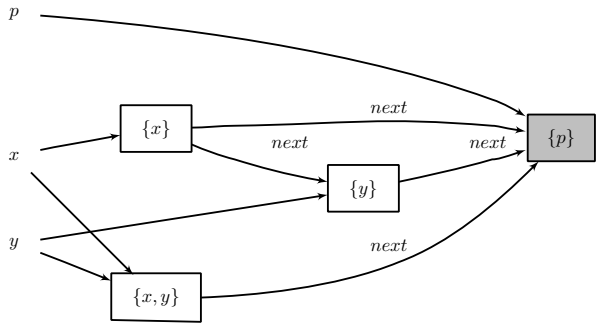


Figure 2.8: An SRW static shape graph $SG^{\mathcal{S}}$.

First, the abstract state $\mathsf{S}^{\mathcal{S}}$ is transformed into a set of abstract states *States* where each variable only points to at most one location:

$$\mathit{States} = \left\{ \begin{array}{c} \{\} \\ \{(x, n_{\{x\}})\} \\ \{(x, n_{\{x,y\}})\} \end{array} \right\} \otimes \left\{ \begin{array}{c} \{\} \\ \{(y, n_{\{y\}})\} \\ \{(y, n_{\{x,y\}})\} \end{array} \right\} \otimes \left\{ \begin{array}{c} \{\} \\ \{(p, n_{\{p\}})\} \end{array} \right\}$$

To complete the transformation of the state, all abstract states that violate the naming scheme are discarded. Recall that there can be no edge $(x, n_{\{x,y\}})$ without y pointing to the same $n_{\{x,y\}}$ location. The remaining abstract states are the compatible abstract states $\mathit{States}^{\mathcal{C}}$:

$$\mathit{States}^{\mathcal{C}} = \left\{ \begin{array}{ll} \{\}, & \{(p, n_{\{p\}})\}, \\ \{(y, n_{\{y\}})\}, & \{(y, n_{\{y\}}), (p, n_{\{p\}})\}, \\ \cancel{\{(y, n_{\{x,y\}})\}}, & \cancel{\{(y, n_{\{x,y\}}), (p, n_{\{p\}})\}}, \\ & \\ \{(x, n_{\{x\}})\}, & \{(x, n_{\{x\}}), (p, n_{\{p\}})\}, \\ \{(x, n_{\{x\}}), (y, n_{\{y\}})\}, & \{(x, n_{\{x\}}), (y, n_{\{y\}}), (p, n_{\{p\}})\}, \\ \cancel{\{(x, n_{\{x\}}), (y, n_{\{x,y\}})\}}, & \cancel{\{(x, n_{\{x\}}), (y, n_{\{x,y\}}), (p, n_{\{p\}})\}}, \\ & \\ \cancel{\{(x, n_{\{x,y\}})\}}, & \cancel{\{(x, n_{\{x,y\}}), (p, n_{\{p\}})\}}, \\ \cancel{\{(x, n_{\{x,y\}}), (y, n_{\{y\}})\}}, & \cancel{\{(x, n_{\{x,y\}}), (y, n_{\{y\}}), (p, n_{\{p\}})\}}, \\ \{(x, n_{\{x,y\}}), (y, n_{\{x,y\}})\}, & \{(x, n_{\{x,y\}}), (y, n_{\{x,y\}}), (p, n_{\{p\}})\} \end{array} \right\}$$

Equally, no selector of any heap cell may point to more than one location in a deterministic heap, so we apply a similar transformation to the abstract heaps $\mathsf{H}^{\mathcal{S}}$ to compute the set of compatible abstract heaps $\mathit{Heaps}^{\mathcal{C}}$:

$$\begin{aligned} \mathit{Heaps} &= \left\{ \begin{array}{c} \{\} \\ \{(n_{\{x\}}, \mathit{next}, n_{\{p\}})\} \\ \{(n_{\{x\}}, \mathit{next}, n_{\{y\}})\} \end{array} \right\} \\ &\otimes \left\{ \begin{array}{c} \{\} \\ \{(n_{\{y\}}, \mathit{next}, n_{\{p\}})\} \end{array} \right\} \\ &\otimes \left\{ \begin{array}{c} \{\} \\ \{(n_{\{x,y\}}, \mathit{next}, n_{\{p\}})\} \end{array} \right\} \end{aligned}$$

Again, we get the set of compatible heap edges $Heaps^{\mathcal{C}}$ by elimination of incompatible edges:

$$
Heaps^{\mathcal{C}} = \left\{ \begin{array}{l}
\{\}, \\
\{(n_{\{x,y\}}, next, n_{\{p\}})\}, \\
\\
\{(n_{\{y\}}, next, n_{\{p\}})\}, \\
\cancel{\{(n_{\{y\}}, next, n_{\{p\}}), (n_{\{x,y\}}, next, n_{\{p\}})\}}, \\
\\
\{(n_{\{x\}}, next, n_{\{y\}})\}, \\
\cancel{\{(n_{\{x\}}, next, n_{\{y\}}), (n_{\{x,y\}}, next, n_{\{p\}})\}}, \\
\\
\{(n_{\{x\}}, next, n_{\{y\}}), (n_{\{y\}}, next, n_{\{p\}})\}, \\
\cancel{\{(n_{\{x\}}, next, n_{\{y\}}), (n_{\{y\}}, next, n_{\{p\}}), (n_{\{x,y\}}, next, n_{\{p\}})\}}, \\
\\
\{(n_{\{x\}}, next, n_{\{p\}})\}, \\
\cancel{\{(n_{\{x\}}, next, n_{\{p\}}), (n_{\{x,y\}}, next, n_{\{p\}})\}}, \\
\\
\{(n_{\{x\}}, next, n_{\{p\}}), (n_{\{y\}}, next, n_{\{p\}})\}, \\
\cancel{\{(n_{\{x\}}, next, n_{\{p\}}), (n_{\{y\}}, next, n_{\{p\}}), (n_{\{x,y\}}, next, n_{\{p\}})\}}
\end{array} \right\}
$$

As there are no compatibility requirements to the sharing information, we get $Sharings^{\mathcal{C}}$ from the sharing information $\mathsf{is}^{\mathcal{S}}$ simply by computing the set of all subsets of $\mathsf{is}^{\mathcal{S}}$:

$$
\begin{array}{rcl}
Sharings^{\mathcal{C}} & = & \mathcal{P}(\{n_{\{p\}}\}) \\
& = & \{\{\}, \{n_{\{p\}}\}\}
\end{array}
$$

Next we combine the results from above to generate all shape graphs entailed by the static shape graph $SG^{\mathcal{S}}$ we started with.

$$
\begin{aligned}
SGS &= \mathit{States}^{\mathcal{C}} \otimes \mathit{Heaps}^{\mathcal{C}} \otimes \mathit{Sharings}^{\mathcal{C}} \\
&= \left\{ \begin{array}{l}
\{\}, \\
\{(p, n_{\{p\}})\}, \\
\{(y, n_{\{y\}})\}, \\
\{(y, n_{\{y\}}), (p, n_{\{p\}})\}, \\
\{(x, n_{\{x\}})\}, \\
\{(x, n_{\{x\}}), (p, n_{\{p\}})\}, \\
\{(x, n_{\{x\}}), (y, n_{\{y\}})\}, \\
\{(x, n_{\{x\}}), (y, n_{\{y\}}), (p, n_{\{p\}})\}, \\
\{(x, n_{\{x,y\}}), (y, n_{\{x,y\}})\}, \\
\{(x, n_{\{x,y\}}), (y, n_{\{x,y\}}), (p, n_{\{p\}})\}
\end{array} \right\} \\
&\otimes \left\{ \begin{array}{l}
\{\}, \\
\{(n_{\{x,y\}}, \mathit{next}, n_{\{p\}})\}, \\
\{(n_{\{y\}}, \mathit{next}, n_{\{p\}})\}, \\
\{(n_{\{x\}}, \mathit{next}, n_{\{y\}})\}, \\
\{(n_{\{x\}}, \mathit{next}, n_{\{y\}}), (n_{\{y\}}, \mathit{next}, n_{\{p\}})\}, \\
\{(n_{\{x\}}, \mathit{next}, n_{\{p\}})\}, \\
\{(n_{\{x\}}, \mathit{next}, n_{\{p\}}), (n_{\{y\}}, \mathit{next}, n_{\{p\}})\},
\end{array} \right\} \\
&\otimes \left\{ \begin{array}{l}
\{\} \\
\{n_{\{p\}}\}
\end{array} \right\}
\end{aligned}
$$

The final step is to discard every shape graph of the 140 shape graphs in SGS that violates any of the five invariants of compatible shape graphs. What remains is the set of compatible shape graphs $SGS^{\mathcal{C}}$ that describes the same store as the original static shape graph $SG^{\mathcal{S}} = (\mathsf{S}^{\mathcal{S}}, \mathsf{H}^{\mathcal{S}}, \mathsf{is}^{\mathcal{S}})$, but with all indeterminism removed from heap edges that originate at named locations and variable edges in general:

$$SGS^{\mathcal{C}} = \left\{ \begin{array}{l} (\{\},\{\},\{\}), \\ \big(\{(p,n_{\{p\}})\},\{\},\{\}\big), \\ \big(\{(y,n_{\{y\}})\},\{\},\{\}\big), \\ \big(\{(y,n_{\{y\}}),(p,n_{\{p\}})\},\{\},\{\}\big), \\ \big(\{(y,n_{\{y\}}),(p,n_{\{p\}})\},\{(n_{\{y\}},next,n_{\{p\}})\},\{\}\big), \\ \big(\{(x,n_{\{x\}})\},\{\},\{\}\big), \\ \big(\{(x,n_{\{x\}}),(p,n_{\{p\}})\},\{\},\{\}\big), \\ \big(\{(x,n_{\{x\}}),(p,n_{\{p\}})\},\{(n_{\{x\}},next,n_{\{p\}})\},\{\}\big), \\ \big(\{(x,n_{\{x\}}),(y,n_{\{y\}})\},\{\},\{\}\big), \\ \big(\{(x,n_{\{x\}}),(y,n_{\{y\}})\},\{(n_{\{x\}},next,n_{\{y\}})\},\{\}\big), \\ \big(\{(x,n_{\{x\}}),(y,n_{\{y\}}),(p,n_{\{p\}})\},\{\},\{\}\big), \\ \big(\{(x,n_{\{x\}}),(y,n_{\{y\}}),(p,n_{\{p\}})\},\{(n_{\{x\}},next,n_{\{p\}})\},\{\}\big), \\ \big(\{(x,n_{\{x\}}),(y,n_{\{y\}}),(p,n_{\{p\}})\},\{(n_{\{x\}},next,n_{\{y\}})\},\{\}\big), \\ \big(\{(x,n_{\{x\}}),(y,n_{\{y\}}),(p,n_{\{p\}})\},\{(n_{\{y\}},next,n_{\{p\}})\},\{\}\big), \\ \big(\{(x,n_{\{x\}}),(y,n_{\{y\}}),(p,n_{\{p\}})\},\{(n_{\{x\}},next,n_{\{p\}}),(n_{\{y\}},next,n_{\{p\}})\},\{n_{\{p\}}\}\big), \\ \big(\{(x,n_{\{x\}}),(y,n_{\{y\}}),(p,n_{\{p\}})\},\{(n_{\{x\}},next,n_{\{y\}}),(n_{\{y\}},next,n_{\{p\}})\},\{\}\big), \\ \big(\{(x,n_{\{x,y\}}),(y,n_{\{x,y\}})\},\{\},\{\}\big), \\ \big(\{(x,n_{\{x,y\}}),(y,n_{\{x,y\}}),(p,n_{\{p\}})\},\{\},\{\}\big), \\ \big(\{(x,n_{\{x,y\}}),(y,n_{\{x,y\}}),(p,n_{\{p\}})\},\{(n_{\{x,y\}},next,n_{\{p\}})\},\{\}\big), \end{array} \right\}$$

■

Chapter 3

Transfer Functions

Both shape analyses are specified as instances of a monotone framework with the complete lattice of properties being $\mathcal{P}(\{\mathcal{CSG}\})$ in the Nielson-Nielson-Hankin case and $\mathcal{P}(\mathcal{SSG})$ for the analysis by Sagiv, Reps, and Wilhelm. For every statement that modifies the structure of the store the statement's transfer function updates the heap representation accordingly. For the other statements, those statements that do not have an effect on the store, the transfer function is simply the identity function.

Six elementary pointer operations are formulated in [SRW98] and [NNH99]. The statements covered by elementary transfer functions are:

$[\![$`x = null`$]\!]$	assignment of *nil* or non-pointer
$[\![$`x = new T`$]\!]$	allocation of a new object
$[\![$`x = y`$]\!]$	assignment of a pointer value
$[\![$`x = y->sel`$]\!]$	assignment of an heap object's field
$[\![$`x->sel = null`$]\!]$	assignment *nil* or non-pointer value to a field
$[\![$`x->sel = y`$]\!]$	assignment of a pointer value to a field

These elementary transfer functions are sufficient for intra-procedural analysis of already normalized programs. Sect 3.1 examines them in detail for the NNH analysis. For an explanation of the same elementary transfer functions for the SRW analysis, please refer to [SRW98], pp. 21–31. The following sections clarifies what is meant with normalizing and covers the composition of elementary transfer functions into transfer functions that can deal with special cases (cf. Sect. 3.3), arbitrarily nested pointer expressions (cf. Sect. 3.4), and with inter-procedural analysis (cf. Sect. 3.5).

3.1 NNH Intra-Procedural Transfer Functions

The transfer functions $f_\ell^{\mathsf{NNH}} : SGS^{\mathcal{C}} \to SGS^{\mathcal{C}}$ associated with program labels ℓ have the form:

$$f_\ell^{\mathsf{NNH}}(SGS) = \bigcup\{\phi_\ell^{\mathsf{NNH}}((\mathsf{S},\mathsf{H},\mathsf{is})) \mid (\mathsf{S},\mathsf{H},\mathsf{is}) \in SGS\}$$

where $\phi_\ell^{\mathsf{NNH}} : SG^{\mathcal{C}} \to SGS^{\mathcal{C}}$ specifies how a single shape graph $SG_\circ^{\mathcal{C}}$ of the input shape graph set $SGS_\circ^{\mathcal{C}}$ is transformed into a set of shape graphs that represent the structure of the store after the execution of statement ℓ. The union of these results for all $SG_\circ^{\mathcal{C}}$ in $SGS_\circ^{\mathcal{C}}$ is then the set of shape graphs $SGS_\bullet^{\mathcal{C}}$ that represents the structure of the store after execution of ℓ.

3.1.1 Transfer Function $[\![$`x = null`$]\!]$

This covers the assignment of *nil* and non-pointer values to a pointer. As the shape analysis is only interested in the pointer-induced structure of the store, the effect of the transfer function $[\![$`x = null`$]\!]$ is to remove the binding that x has, updating the heap locations' names accordingly.

The renaming of abstract locations is specified by the function:

$$k_x(n_Z) \;=\; n_{Z\setminus\{x\}}$$

The transfer function is then:

$$\phi_\ell^{\mathsf{NNH}}((\mathsf{S},\mathsf{H},\mathsf{is})) \;=\; \{kill_x((\mathsf{S},\mathsf{H},\mathsf{is}))\}$$

where the function $kill_x((\mathsf{S},\mathsf{H},\mathsf{is}))$ is given by:

$$\begin{array}{rcl} kill_x((\mathsf{S},\mathsf{H},\mathsf{is})) & = & (\mathsf{S}',\mathsf{H}',\mathsf{is}') \\ \mathsf{S}' & = & \{(z, k_x(n_Z)) \mid (z,n_Z) \in \mathsf{S} \wedge z \neq x\} \\ \mathsf{H}' & = & \{(k_x(n_V), sel, k_x(n_W)) \mid (n_V, sel, n_W) \in \mathsf{H}\} \\ \mathsf{is}' & = & \{k_x(n_X) \mid n_X \in \mathsf{is}\} \end{array}$$

When $(x, n_{\{x\}})$ is in the abstract state S then the two abstract location $n_{\{x\}}$ and $n_\emptyset$ are merged when the variable edge originating at x is removed. Only

when both $n_{\{x\}}$ and $n_\emptyset$ were unshared in the original shape graph can $n_\emptyset$ be unshared in the updated shape graph.

3.1.2 Transfer Function $[\![\texttt{x = new T}]\!]$

The allocation transfer function has the effect that a variable edge to a fresh unshared heap location is created. If the variable that should point to that new cell was part of a variable edge already, that old binding needs to be removed first.

$$\begin{aligned}\phi_\ell^{\mathsf{NNH}}((\mathsf{S},\mathsf{H},\mathsf{is})) &= \{(\mathsf{S}' \cup \{(x, n_{\{x\}}), \mathsf{H}', \mathsf{is}')\} \\ (\mathsf{S}',\mathsf{H}',\mathsf{is}') &= kill_x(\mathsf{S},\mathsf{H},\mathsf{is})\end{aligned}$$

3.1.3 Transfer Function $[\![\texttt{x = y}]\!]$

If $x = y$ then the transfer function is the identity. In the case where $x \neq y$ the effect of the transfer function is to remove the old binding to x and establish a new binding to the same location that y is pointing to. As x and y then point to the same location, the shape graph node with y in its label needs to be renamed to also include x after the transfer function. The renaming is specified by the function:

$$g_x^y(n_Z) = \begin{cases} n_{Z\cup\{x\}} & \text{if } y \in Z \\ n_Z & \text{otherwise} \end{cases}$$

The transfer function for $[\![\texttt{x = y}]\!]$ thus is:

$$\phi_\ell^{\mathsf{NNH}}((\mathsf{S},\mathsf{H},\mathsf{is})) = \{(\mathsf{S}'',\mathsf{H}'',\mathsf{is}'')\}$$

where $(\mathsf{S}',\mathsf{H}',\mathsf{is}') = kill_x((\mathsf{S},\mathsf{H},\mathsf{is}))$ and

$$\begin{aligned}\mathsf{S}'' &= \{(z, g_x^y(n_Z)) \mid (z, n_Z) \in \mathsf{S}'\} \\ &\quad \cup \{(x, g_x^y(n_Y)) \mid (y', n_Y) \in \mathsf{S}' \wedge y' = y\} \\ \mathsf{H}'' &= \{(g_x^y(n_V), sel, g_x^y(n_W)) \mid (n_V, sel, n_W) \in \mathsf{H}'\} \\ \mathsf{is}'' &= \{g_x^y(n_Z) \mid n_Z \in \mathsf{is}'\}\end{aligned}$$

3.1.4 Transfer Function ⟦`x = y->sel`⟧

This transfer function is the most complex one in the NNH shape analysis. It is responsible for the materialisation of named locations out of the summary location. In [NNH99] it is given in terms of the reversal of function $kill_x((\mathsf{S}, \mathsf{H}, \mathsf{is}))$, which is to be computed by rejection of certain graphs of the full set of compatible shape graphs.

The effect of ⟦`x = y->sel`⟧ is to remove the old binding x had and let it point to the location that the *sel* field of y points to, thereby giving this location the label $\{x\}$.

Depending on the input shape graph we can identify three cases for the edges with y and *sel*:

1. (a) There is no variable edge that originates at y, or (b) there is one, but then there is no heap edge $(n_{\{y\}}, sel, m)$ in the shape graph.

   This means that either (a) y does not hold a valid pointer value , in which case it would be a run-time error in the program being analysed to dereference y, or that (b) y is a pointer to a location but the *sel* field of that location is no pointer. In either case the value of `y->sel` is null, any non-pointer value or undefined.

2. y and *sel* have corresponding variable- and heap edges in the shape graph so that `y->sel` points to a named abstract location. In this case, the target of `y->sel` is also the target of a variable edge that provides the name of the location.

3. y and *sel* have corresponding variable- and heap edges in the shape graph so that `y->sel` points to the abstract summary location $n_\emptyset$. This is the materialisation case where the assignment of `y->sel` to x provides one of the abstract locations clustered into $n_\emptyset$ with the label $\{x\}$.

Case 1. As there is no abstract location for `y->sel` and hence no abstract location to rename and no binding to establish we take the transfer function as:

$$\phi_\ell^{\mathsf{NNH}}((\mathsf{S}, \mathsf{H}, \mathsf{is})) \quad = \quad \{kill_x((\mathsf{S}, \mathsf{H}, \mathsf{is}))\}$$

As noted above, this case includes the case where y must not be dereferenced as it is not a valid pointer but null, uninitialized, or not of pointer type.

Case 2. This covers the case where `y->sel` points to a named abstract location, so there exists a heap edge $(n_{\{Y\}}, sel, n_{\{T\}})$. The purpose of this transfer function is then to rename the location $n_{\{T\}}$ to include x in its label and to establish a variable binding $(x, n_{\{T\}})$.

The renaming of abstract location $n_{\{T\}}$ is specified by the function $r_x^T(n_Z)$:

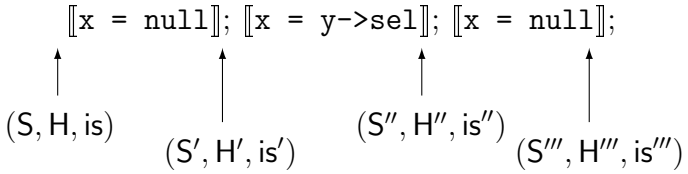
$$r_x^T(n_Z) \;=\; \begin{cases} n_{T\cup\{x\}} & \text{if } Z = T \\ n_Z & \text{otherwise} \end{cases}$$

The transfer function is then:

$$\begin{aligned} \phi_\ell^{\mathsf{NNH}}((\mathsf{S},\mathsf{H},\mathsf{is})) &= \{(\mathsf{S}'',\mathsf{H}'',\mathsf{is}'')\} \\ (\mathsf{S}',\mathsf{H}',\mathsf{is}') &= kill_x((\mathsf{S},\mathsf{H},\mathsf{is})) \\ \mathsf{S}'' &= \{(z, r_x^T(n_Z)) \mid (z,n_Z) \in \mathsf{S}'\} \cup \{(x, r_x^T(n_T))\} \\ \mathsf{H}'' &= \{(r_x^T(n_V), sel', r_x^T(n_W)) \mid (n_V, sel', n_W) \in \mathsf{H}'\} \\ \mathsf{is}'' &= \{r_x^T(n_Z) \mid n_Z \in \mathsf{is}'\} \end{aligned}$$

Sharing is preserved by this operation; only if n_T is shared in H' is it that $n_{T\cup\{x\}}$ is shared in H''.

Case 3 – Materialisation. Here `y->sel` points to the abstract summary location $n_\emptyset$ that represents the target of $(n_Y, sel, n_\emptyset)$ as well as a set of other locations. By assigning one of these locations to x the location gets a label, therefore becomes a named abstract location and is materialised out of the summary location. $n_{\{x\}}$ then represents only the target of above heap edge while $n_\emptyset$ continues to represent all other abstract locations that remain clustered in the summary location.



[NNH99] approach the definition of the transfer function for the materialisation case by determining candidate shape graphs $(\mathsf{S}'',\mathsf{H}'',\mathsf{is}'')$ holding after the assignment ⟦`x = y->sel`⟧. First, the properties of above sequence of transfer functions are studied.

It is immediate that:

$$\begin{array}{rcl} kill_x((\mathsf{S},\mathsf{H},\mathsf{is})) & = & (\mathsf{S}',\mathsf{H}',\mathsf{is}') \\ kill_x((\mathsf{S}'',\mathsf{H}'',\mathsf{is}'')) & = & (\mathsf{S}''',\mathsf{H}''',\mathsf{is}''') \end{array}$$

as $kill_x$ represents the effect of removing the binding to x. The shape graphs possible at the point described by $(\mathsf{S}',\mathsf{H}',\mathsf{is}')$ should be the same as those possible at the point described by $(\mathsf{S}''',\mathsf{H}''',\mathsf{is}''')$. We therefore demand:

$$(\mathsf{S}''',\mathsf{H}''',\mathsf{is}''') \;=\; (\mathsf{S}',\mathsf{H}',\mathsf{is}')$$

It then follows that

$$kill_x((\mathsf{S},\mathsf{H},\mathsf{is})) \;=\; (\mathsf{S}',\mathsf{H}',\mathsf{is}') \;=\; kill_x((\mathsf{S}'',\mathsf{H}'',\mathsf{is}'')) \;=\; (\mathsf{S}''',\mathsf{H}''',\mathsf{is}''')$$

The operation that separates these states is $[\![\texttt{x = y->sel}]\!]$, which should only establish a binding to x. So, if the binding to x is left aside, the shape graphs before and after should be equal.

The transfer function is then given as the full set of compatible shape graphs that are equal to the input shape graph under the function $kill_x$:

$$\begin{array}{rcll} \phi_\ell^{\mathsf{NNH}}((\mathsf{S},\mathsf{H},\mathsf{is})) & = & \{(\mathsf{S}'',\mathsf{H}'',\mathsf{is}'') \mid & (\mathsf{S}'',\mathsf{H}'',\mathsf{is}'') \text{ is compatible} \wedge \\ & & & kill_x((\mathsf{S}'',\mathsf{H}'',\mathsf{is}'')) = kill_x((\mathsf{S},\mathsf{H},\mathsf{is})) \wedge \\ & & & (x, n_{\{x\}}) \in \mathsf{S}'' \;\wedge\; (n_Y, sel, n_{\{x\}}) \in \mathsf{H}''\} \end{array}$$

For the abstract state this means that we just add a variable edge $(x, n_{\{x\}})$ to obtain S''. For the sharing information it is established that:

$$\begin{array}{rcl} \mathsf{is}' \setminus \{n_\emptyset\} & = & \mathsf{is}'' \setminus \{n_\emptyset, n_{\{x\}}\} \\ n_\emptyset \in \mathsf{is}' & \text{iff} & n_\emptyset \in \mathsf{is}'' \vee n_{\{x\}} \in \mathsf{is}'' \end{array}$$

which means that the sharing information for all abstract locations except $n_\emptyset$ remains unchanged. If $n_\emptyset$ is unshared in $(\mathsf{S}',\mathsf{H}',\mathsf{is}')$ then both $n_\emptyset$ and the newly created $n_{\{x\}}$ are unshared as the materialisation cannot introduce sharing. On the other side, if $n_\emptyset$ was shared before the materialisation then that sharing can not go away but must give rise to $n_\emptyset$ or $n_{\{x\}}$ or both being shared.

When one of the locations clustered into $n_\emptyset$ is extracted and given a name then every edge destined at $n_\emptyset$ before the materialisation could have pointed to the now named $n_{\{x\}}$ location or to any other location still represented by $n_\emptyset$. In the first case, the edge now has to be redirected to the new location $n_{\{x\}}$ while in the latter case it must remain pointing to $n_\emptyset$. Because we cannot rule out either case we have to conservatively create graphs for both possibilities.

Similarly, edges that originated at $n_\emptyset$ before the materialisation could now originate at $n_\emptyset$ or at $n_{\{x\}}$. If the edge's target was shared, there even could be both edges, one starting at $n_\emptyset$ and one at $n_{\{x\}}$ in the same graph after materialisation. Above, this was inhibited by the invariants of compatible shape graphs as no selector may point to more than one location simultaneously in a compatible shape graph.

For each internal edge $(n_\emptyset, sel, n_\emptyset)$ that existed in the original shape graph there are four possible heap edges after materialisation: $(n_\emptyset, sel, n_\emptyset)$, $(n_\emptyset, sel, n_{\{x\}})$, $(n_{\{x\}}, sel, n_\emptyset)$, $(n_{\{x\}}, sel, n_{\{x\}})$.

Combinations of these edges are also possible if the sharing information permits it, if, for instance, both the new $n_{\{x\}}$ and any other location that is still summarized into $n_\emptyset$ had a backedge to themselves. Before materialisation there would have been only one backedge $(n_\emptyset, sel, n_\emptyset)$, but as soon as $n_{\{x\}}$ became distinguishable from the other anonymous locations, it became evident that there actually were more backedges.

3.1.5 Transfer Function ⟦`x->sel = null`⟧

Like ⟦`x = null`⟧ this transfer function reflects the assignment of non-pointer values including the null-pointer, only that ⟦`x->sel = null`⟧ does it for pointers that are stored in heap objects.

If there is no variable edge (x, n_X) in the abstract state then there also is no such heap object. This captures the case where the programmer wants to write into a field of an object that does not or no longer exist in the program being analysed. The effect on the shape graph is but the identity function.

If there is a variable edge (x, n_X) but no heap edge (n_X, sel, n_Y) then that pointer is absent already and need not be removed. In this case, too, the transfer function is the identity.

The interesting case is when there are both a variable edge (x, n_X) and a heap edge (n_X, sel, n_Y). Then the effect of the transfer function is to remove the heap edge (n_X, sel, n_Y):

$$\phi_\ell^{\mathsf{NNH}}((\mathsf{S},\mathsf{H},\mathsf{is})) \;=\; \{kill_{x.sel}((\mathsf{S},\mathsf{H},\mathsf{is}))\}$$

where the function $kill_{x.sel}((\mathsf{S},\mathsf{H},\mathsf{is})) = (\mathsf{S}',\mathsf{H}',\mathsf{is}')$ is given by:

$$\begin{array}{lcl}
\mathsf{S}' & = & \mathsf{S} \\
\mathsf{H}' & = & \{(n_V, sel', n_W) \mid (n_V, sel', n_W) \in \mathsf{H} \;\wedge\; \neg(X = V \;\wedge\; sel = sel')\} \\
\mathsf{is}' & = & \begin{cases}
\mathsf{is} \setminus n_U & \text{if} \quad n_U \in \mathsf{is} \;\wedge \\
 & \qquad\quad \#into(n_U, \mathsf{H}') \leq 1 \;\wedge \\
 & \qquad\quad \nexists sel' : (n_\emptyset, sel', n_U) \in \mathsf{H}' \\
\mathsf{is} & \text{otherwise}
\end{cases}
\end{array}$$

Whenever an heap edge is being removed, the explicit sharing information is updated, too. This was not necessary in the transfer function of $[\![$`x = null`$]\!]$ as the explicit sharing information only records heap-induced sharing, i.e. only heap edges count.

The function $\#into(n_U, \mathsf{H}')$ returns the number of heap edges that point to the abstract location n_U in the heap H'. If only one heap edge points to n_U then this location is no longer shared, unless the edge comes from $n_\emptyset$, in which case it possibly represents many unnamed heap locations pointing to n_U.

Note that a location n_U that became shared as a result of two named locations n_X and n_Y pointing to it, cannot lose its sharing when either of the two nodes was merged into $n_\emptyset$.

3.1.6 Transfer Function $[\![$`x->sel = y`$]\!]$

This transfer function assigns the value of y to the *sel* field in an object x.

As with the $[\![$`x->sel = null`$]\!]$ transfer function, the case where no variable edge (x, n_X) exists in the shape graph covers situations in which the program being analysed tries to access an object that does not or no longer exist. For the shape analysis the transfer function in this case is, again, the identity function. But a warning message could be issued when this case is detected.

When there exists an object at x in the shape graph, the effect of the transfer function $[\![$`x->sel = y`$]\!]$ is to establish a new binding for its field *sel* after the old binding, if any, has been removed.

If there is no $(y, n_{\{Y\}}) \in \mathsf{S}$ this means that y is not a pointer and the result of the transfer function is equal to that of $[\![$`x->sel = null`$]\!]$:

$$\phi_\ell^{\mathsf{NNH}}((\mathsf{S},\mathsf{H},\mathsf{is})) \;=\; \{kill_{x.sel}((\mathsf{S},\mathsf{H},\mathsf{is}))\}$$

For the case where both (x, n_X) and (y, n_Y) are in the abstract state the transfer function is:

$$\begin{aligned}
\phi_\ell^{\mathsf{NNH}}((\mathsf{S},\mathsf{H},\mathsf{is})) &= \{(\mathsf{S}'',\mathsf{H}'',\mathsf{is}'')\} \\
(\mathsf{S}',\mathsf{H}',\mathsf{is}') &= kill_{x.sel}((\mathsf{S},\mathsf{H},\mathsf{is})) \\
\mathsf{S}'' &= \mathsf{S}'(=\mathsf{S}) \\
\mathsf{H}'' &= \mathsf{H}' \cup \{(n_X, sel, n_Y)\} \\
\mathsf{is}'' &= \begin{cases} \mathsf{is}' \cup \{n_Y\} & \text{if } \#into(n_Y,\mathsf{H}') \geq 1 \\ \mathsf{is}' & \text{otherwise} \end{cases}
\end{aligned}$$

Through adding a new heap edge (n_X, sel, n_Y) it may be that location $n_{\{Y\}}$ becomes shared. The explicit sharing information is therefore updated accordingly.

3.2 SRW Intra-Procedural Transfer Functions

The transfer functions for the SRW shape analysis are described in [SRW98], pp. 21–31.

3.3 Transfer Function Wrappers

For simplicity in the elementary transfer functions it is demanded that all variables involved in assignments are distinct. Elementary transfer functions therefore do not have to cover "special cases" of assignments where a variable is assigned an expression that contains the same variable. This is the first aspect of normalizing assignments.

To achieve this, all transfer function calls are replaced by semantically equivalent sequences of elementary transfer functions that use temporary variables for intermediate results. This ensures that the nullification of the left-hand-side variable does not destroy the right-hand-side value of an assignment.

The composition of monotone functions $SGS^{\mathcal{C}} \to SGS^{\mathcal{C}}$ is monotone. All elementary transfer functions are monotone, so the compound transfer functions composed of the elementary transfer functions are also monotone.

For instance, the common statement $[\![$`x = x->sel`$]\!]$ is replaced by the semantically equivalent sequence:

$[\![$`tmp = x->sel`$]\!]$	save value in a temporary variable
$[\![$`x = null`$]\!]$	perform nullification
$[\![$`x = tmp `$]\!]$	then the assignment
$[\![$`tmp = null`$]\!]$	eliminate temporary variable again

Because the temporary variable is always a fresh one, no nullification is required before the assignment to it.

By going through this intermediate step of putting the right-hand-side value into a temporary variable, unwanted destruction of the value during nullification is avoided for programs that contain statements of the given form $[\![$`x = x->sel`$]\!]$. For the elementary transfer functions this intermediate step guarantees that the variables on either side of the assignment are distinct.

The remaining transfer functions also have their corresponding wrappers as shown in Fig. 3.1. Note that there is also a compound transfer function for $[\![$`x->sel = new`$]\!]^T$.

3.4 Normalization of Pointer Expressions

The second form of normalization is the decomposition of complex pointer expressions (the other was resolving special cases, cf. Sect. 3.3). As the elementary transfer functions only cover cases with one selector, more complex pointer expressions need to be translated into sequences of semantically equivalent statements consisting of only the elementary operations. This is done with the help of newly introduced temporary variables and statements.

For instance $[\![$`x->sel = y->sel`$]\!]^\ell$ is equivalent to the sequence:

$$[\![\texttt{tmp = y->sel}]\!]^{\ell_1}, [\![\texttt{x->sel = tmp}]\!]^{\ell_2}, [\![\texttt{tmp = null}]\!]^{\ell_3}$$

where `tmp` is a temporary variable that has no clashes with any other variable already in existence and ℓ_1, ℓ_2, ℓ_3 are fresh program labels. The transfer function for ℓ is then the composition of transfer functions $f_\ell = f_{\ell_3} \circ f_{\ell_2} \circ f_{\ell_1}$.

$[\![$`x = new`$]\!]^T \equiv$
  $[\![$`x = null`$]\!]$
  $[\![$`x = new`$]\!]$

$[\![$`x = null`$]\!]^T \equiv$
  $[\![$`x = null`$]\!]$

$[\![$`x = y`$]\!]^T \equiv$
  Identity, if `x` = `y`.
  Otherwise:
  $[\![$`tmp = y`$]\!]$
  $[\![$`x = null`$]\!]$
  $[\![$`x = tmp`$]\!]$
  $[\![$`tmp = null`$]\!]$

$[\![$`x = y->sel`$]\!]^T \equiv$
  $[\![$`tmp = y->sel`$]\!]$
  $[\![$`x = null`$]\!]$
  $[\![$`x = tmp`$]\!]$
  $[\![$`tmp = null`$]\!]$

$[\![$`x->sel = y`$]\!]^T \equiv$
  $[\![$`tmp = y`$]\!]$
  $[\![$`x->sel = null`$]\!]$
  $[\![$`x->sel = tmp`$]\!]$
  $[\![$`tmp = null`$]\!]$

$[\![$`x->sel = null`$]\!]^T \equiv$
  $[\![$`x->sel = null`$]\!]$

$[\![$`x->sel = new`$]\!]^T \equiv$
  $[\![$`x->sel = null`$]\!]$
  $[\![$`tmp = new`$]\!]$
  $[\![$`x->sel = tmp`$]\!]$
  $[\![$`tmp = null`$]\!]$

Figure 3.1: Transfer function wrappers $[\![\cdot]\!]^T$.

Example 7 Consider the assignment with nested pointer expressions shown in Fig. 3.2.

```
a->next->next->next = a->next->next->next;
```

Figure 3.2: C++ code with nested pointer expressions.

In order to perform shape analysis on this program fragment, the nested expressions first have to be decomposed into the elementary pointer operations for which transfer functions are specified:

$$\underbrace{a \to next} \to next \to next \;=\; \underbrace{a \to next} \to next \to next; \quad \Big| \; \texttt{lhs1 = a->next;}\ \texttt{rhs1 = a->next;}$$

$$\underbrace{lhs1 \;\to\; next} \to next \;=\; \underbrace{rhs1 \to next} \to next; \quad \Big| \; \texttt{lhs2 = lhs1->next;}\ \texttt{rhs2 = rhs1->next;}$$

$$lhs2 \;\to\; next \;=\; \underbrace{rhs2 \to next}; \quad \Big| \; \texttt{rhs3 = rhs2->next;}$$

$$lhs2 \;\to\; next \;=\; rhs3; \quad \Big| \; \texttt{lhs2 ->next = rhs3;}$$

■

After the assignment, the temporary variables are no longer needed and can be removed from the shape graph by nullification. The effect on the graph is then the same as if there had been a transfer function for arbitrarily nested pointer expressions.

However, it is worth noting that keeping the temporary variables in the graph even after the assignment has been performed may significantly improve the overall precision of the analysis as previously unnamed locations clustered into the summary node $n_\emptyset$ are now given a name (that of the temporary variables). These locations therefore remain materialised in the named nodes and are no longer lost in the summary location. The effects of retaining temporary variables on precision and analysis runtime are discussed in Sect. 6.

3.5 Inter-Procedural Transfer Functions

The SRW and NNH shape analyses are limited to the analysis of intra-procedural program fragments. To increase their precision and make them useful for whole program analysis we employ a naming scheme for variables that takes scope into account and model function calls with suitable assignments to temporary global variables for the passing of arguments and return values, thus turning the analyses into inter-procedural shape analyses.

All assignments to temporary variables are performed using the elementary transfer functions already described (cf. Sect. 3). The composition of monotone functions $SGS^{\mathcal{C}} \to SGS^{\mathcal{C}}$ (or $SG^{\mathcal{S}} \to SG^{\mathcal{S}}$ for SRW) is monotone. All elementary transfer functions are monotone, so the compound transfer functions composed of the elementary transfer functions are also monotone.

3.5.1 Naming Scheme

Variables in a program are assigned unique identifiers during the preparation of the ICFG. Two variables get the same identifier *iff* they are in fact the same variable, i.e. two variables with the same name, but in different scopes, have different identifiers [SBPP09, p. 16].

By coding the scope of variables into their identifier the various instances of variables with the same name can be kept apart. Only when reporting results, e.g. when alias sets are annotated in a program, are the variables converted back to their textual representation and the scope information is lost. When analysis results are to be used as input to other analyses, the full information is retained as analyses directly operate on these identifiers.

3.5.2 Function Calls

Neither of the two shape analyses have in their original texts been extended to account for function calls. Both are limited to intra-procedural program analysis.

The term *function* in the following is meant to also include procedures, member functions or methods, constructors, destructors, and overloaded operators.

Seen from a control flow point of view, a function is a section of code that is inserted at the function call site so that the control flow enters the function, executes it until its exit is reached, and then continues right after the point of the function call. At the begin of the function, its parameters are bound to the values that the function caller passed as arguments to the function. The value returned by a function must be made available to its caller.

Which ICFG nodes are connected to which other nodes in order to model a function call $[\![$`foo(A,B)`$]\!]$ is depicted in Fig. 3.3. `ArgumentAssignment` is an ICFG node that represents the assignment of function arguments to temporary variables. Next follows the `Call` node which is connected to the function's entry point via the call-edge and to the function return point via the local-edge. Analysis information passed along the call-edge is available in the function body while the information passed along the local-edge remains on the outside. After completing the function, the analysis results from inside the function and that passed along the local edge are joined at the `Return` node.

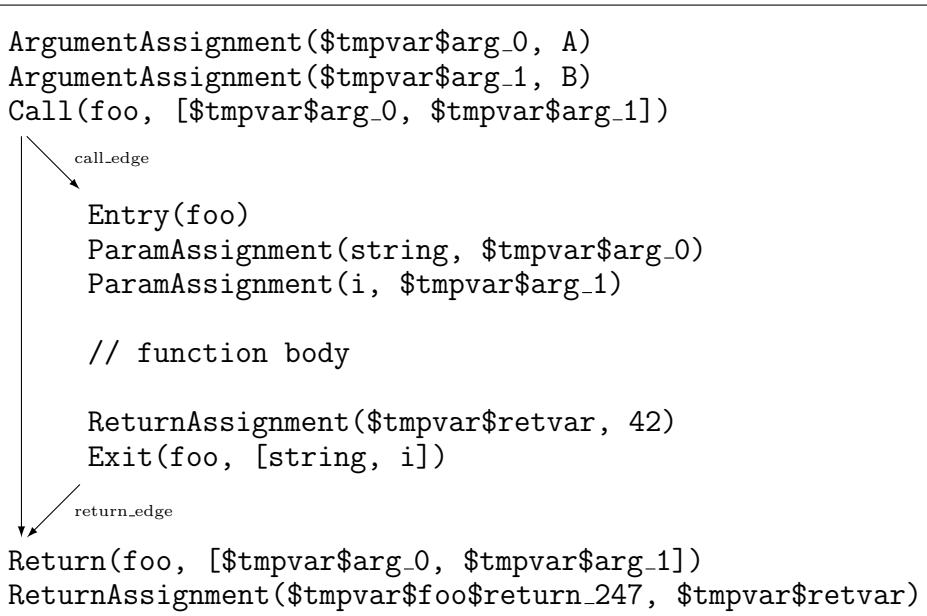


Figure 3.3: ICFG nodes involved in a function call.

Inside the function, the first nodes assign the temporary argument variables to the formal parameters of the function. After that, the passing in of argument values is complete and the nodes that make up the function's body are next to be visited.

At the end of a function its return value is assigned to the temporary variable `$tmpvar$retvar` which is used for all the return values of functions. Immediately after the `Return` node is another special assignment: `ReturnAssignment` puts the value of `$tmpvar$retvar` into another temporary variable unique to this invocation of the function. Every `return` statement in a function performs this assignment and then immediately jumps to the function's `Exit` node.

A function with n parameters can rely on the fact that the temporary variables `$tmpvar$arg_0` – `$tmpvar$arg_n` are filled with the arguments passed to the function at its call site. Once the temporary variables are read from, they are no longer needed and can be eliminated. Accordingly can the caller rely on the agreement that the return value can be read from `$tmpvar$retvar`. These conventions allow the passing of data from caller to callee and back.

Constructors, destructors, virtual methods, and more on implementation details for C++ specifics can be found in Sect. 5.5.2.

Chapter 4

Three-Valued Alias Computation from Shape Graphs

The SRW and NNH shape analysis algorithms produce for every program point $\ell \in \mathbf{Lab}_\star$ a conservative representation of the heap structure that can possibly arise during the actual execution of a program at ℓ. The following chapter explains how this conservative representation can be used to identify may- and must-aliases within that program.

Definition 7 (may-alias)**.** *Two expressions e_1 and e_2 are may-aliases at a program point ℓ if there exists at least one execution sequence leading to ℓ that produces a* concrete heap *in which both e_1 and e_2 refer to the same cell.*

Definition 8 (must-alias)**.** *Two expressions e_1 and e_2 are must-aliases at a program point ℓ if all execution sequences leading to ℓ produce a* concrete heap *in which both e_1 and e_2 refer to the same cell.*

Definition 9 (not aliased, independent)**.** *Two expressions e_1 and e_2 are not aliased at a program point ℓ if there exists no execution sequence leading to ℓ that produces a* concrete heap *in which e_1 and e_2 refer to the same cell ℓ.*

Using above definitions we can define for every program point ℓ a set of must-alias pairs $\mathbf{Must}_\ell$, may-alias pairs $\mathbf{May}_\ell$ and a set of expression pairs that do not alias $\mathbf{Indep}_\ell$:

$$\begin{aligned}\mathbf{May}_\ell &= \{(e_i, e_j) \mid e_i, e_j \in \mathbf{Exp}_\star, \mathrm{Al}(SGS_\ell, e_i, e_j) = \tfrac{1}{2}\}\\ \mathbf{Must}_\ell &= \{(e_i, e_j) \mid e_i, e_j \in \mathbf{Exp}_\star, \mathrm{Al}(SGS_\ell, e_i, e_j) = 1\}\\ \mathbf{Indep}_\ell &= \{(e_i, e_j) \mid e_i, e_j \in \mathbf{Exp}_\star, \mathrm{Al}(SGS_\ell, e_i, e_j) = 0\}\end{aligned}$$

From Definitions 7–9 follow these relations between the alias sets:

$$\begin{aligned}\mathbf{Must}_\ell &\subseteq \mathbf{May}_\ell\\ \mathbf{May}_\ell \cap \mathbf{Indep}_\ell &= \emptyset\\ \mathbf{May}_\ell \cup \mathbf{Indep}_\ell &= \mathbf{Exp}_\star \times \mathbf{Exp}_\star\end{aligned}$$

4.1 Extraction of the Alias Information

For every point $\ell \in \mathbf{Lab}_\star$ in the program exists a heap representation that is the result of a shape analysis. For the purpose of this discussion, it is assumed that the representation has been transformed into sets of compatible shape graphs SGS_ℓ as used by the NNH analysis. Section 2.7 shows the transformation of SRW summary shape graphs into NNH compatible shape graph sets. In practice, however, it is more efficient to avoid this conversion (cf. Sect. 6.2.1).

Consider the pointer expressions e^v and e^w:

$$\begin{aligned}e^v &\equiv v \to sel_1^v \to sel_2^v \to \ldots \to sel_N^v\\ e^w &\equiv w \to sel_1^w \to sel_2^w \to \ldots \to sel_M^w\end{aligned}$$

and their corresponding access paths p^v and p^w in a shape graph:

$$\begin{aligned}p^v &\equiv (v, n_0^v), (n_0^v, sel_1^v, n_1^v), (n_1^v, sel_2^v, n_2^v), \ldots, (n_{N-1}^v, sel_N^v, n_N^v)\\ p^w &\equiv (w, n_0^w), (n_0^w, sel_1^w, n_1^w), (n_1^w, sel_2^w, n_2^w), \ldots, (n_{N-1}^w, sel_N^w, n_N^w)\end{aligned}$$

Given two expressions e^v and e^w Algorithm 1 solves the aliasing problem for a given shape graph set SGS_ℓ, i.e. it answers the question whether every execution path leading to ℓ results in a heap where e^v and e^w refer to the same concrete location and thus are aliases. Clearly we can only say that two expressions must alias if they must alias in every $SG_i \in SGS_\ell$. If they

do not alias in any SG_i it is certain they are not aliases. In any other case it must be said that e^v and e^w possibly alias. The latter also includes those cases where it cannot be decided reliably whether two expressions refer to the same concrete location, as the shape graphs $SG_i \in SGS_\ell$ that are the result of the shape analysis are only finite approximations of the concrete store (cf. Sect. 2.6). If the corresponding access paths end at the summary location $n_\emptyset$ it is unknown, due to the summary location abstraction $n_\emptyset$, whether the expressions refer to the same concrete location or not. This uncertainty is reflected by a result of $\frac{1}{2}$ returned by the alias test.

Algorithm 1 Al(SGS_ℓ, e^v, e^w): Alias Extraction from a Shape Graph Set

1: $SGS_\ell \Leftarrow$ set of shape graphs at program point ℓ
2: $e^v \Leftarrow$ an expression
3: $e^w \Leftarrow$ an expression
4: **for all** $SG_i \in SGS_\ell$ **do**
5: $\quad al_i = \texttt{ExpressionAliasing}(SG_i, e^v, e^w)$
6: **end for**
7: **return** $al_1 \oplus al_2 \oplus \ldots \oplus al_n$

The $\oplus$ operator is used to combine the results of the individual alias tests for each shape graph $SG_i \in SGS_\ell$ into a single result. Whenever the aliasing is not certain for one shape graph, the result has to be may-alias ($\frac{1}{2}$) for the whole shape graph set at that program point. Note that also in the case where e^v and e^w are must-aliases in one shape graph and not aliased in another, the result for the whole shape graph set has to be may-alias.

The three-valued operator $\oplus$ is therefore defined as follows:

$$\oplus(x, y) = \begin{cases} x & \text{iff } x = y \\ \frac{1}{2} & \text{otherwise} \end{cases}$$

$\oplus(x,y)$	0	$\frac{1}{2}$	1
0	0	$\frac{1}{2}$	$\frac{1}{2}$
$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$
1	$\frac{1}{2}$	$\frac{1}{2}$	1

Whether two expressions e^v and e^w are aliases in the context of a single shape graph $SG_i \in SGS_\ell$ is determined by the `ExpressionAliasing` test. A naive implementation of this test is given in Algorithm 2, which closely resembles the algorithm presented by [RSW02].

$$same_{naive}(n_N^v, n_M^w) = \begin{cases} \frac{1}{2} & \text{iff } n_N^v = n_\emptyset \wedge n_M^w = n_\emptyset \\ & \textit{summary location } n_\emptyset \\ 1 & \text{iff } n_N^v = n_M^w \\ & \textit{the same named location} \\ 0 & \text{otherwise} \\ & \textit{different locations} \end{cases}$$

Algorithm 2 `ExpressionAliasing(`SG_i`,` e^v`,` e^w`)`

1: $(S, H, is) \Leftarrow SG_i$ `/*` a single shape graph `*/`
2: $v \Leftarrow$ variable at the head of expression e^v
3: $w \Leftarrow$ variable at the head of expression e^w
4: $Sel^v \Leftarrow$ set of selectors in expression e^v
5: $Sel^w \Leftarrow$ set of selectors in expression e^w
6: **if** $\exists\ sel^v_{i,1\leq i\leq N} \in Sel^v, sel^w_{j,1\leq j\leq M} \in Sel^w, n_0^v, \ldots, n_N^v, n_0^w, \ldots, n_M^w$:
$(v, n_0^v) \in S \wedge \bigwedge_{i=1}^{N} (n_{i-1}^v, sel_i^v, n_i^v) \in H$
$(w, n_0^w) \in S \wedge \bigwedge_{i=1}^{M} (n_{i-1}^w, sel_i^w, n_i^w) \in H$ **then**
7: **return** $same_{naive}(n_N^v, n_M^w)$
8: **else** `/*` e^v or e^w does not refer to a location `*/`
9: **return** 0
10: **end if**

Algorithm 2 reports that two expressions e^v, e^w are must-aliases in SG_i if their corresponding access paths p^v, p^w can be found in the shape graph SG_i and both paths have the same *named* node $n_N^v = n_M^w \neq n_\emptyset$ as their final vertex; that is, if the two expressions refer to the same abstract node that represents a single concrete node.
If the paths are not part of the shape graph (Alg. 2, line 8) or their final vertices are different, the expressions are not aliases.
And finally, if both access paths end at the abstract summary location $n_\emptyset$ the expressions e^v and e^w are may-aliases. Note that this is done irrespective of the summary location being shared. Consider the two cases of (a) a shared summary location and (b) an unshared summary location to see that the sharing information has no influence on the result:

(a) $n_\emptyset \in is$: A shared summary location means that two (or more) of the locations summarized by $n_\emptyset$ may be identical. As it cannot be recovered

which locations that are, the conservative may-alias answer ($\frac{1}{2}$) must be given for every pair of paths that end at the shared summary location.

(b) $n_\emptyset \notin is$: If the summary location is unshared, all locations summarized by $n_\emptyset$ are distinct. But still it must not be concluded that two expressions with paths that both end at the summary location never refer to the same concrete location!

See the shape graph in Fig. 4.1. Here the summary location is unshared, so all locations summarized in $n_\emptyset$ are distinct. Although the expressions $e^v =$ `x->next->next`, $e^w =$ `y->next->next` both have paths that end at the unshared summary location, they nevertheless refer to the same concrete location. This follows from Invariant 3 (cf. Sect. 2.6.2) of compatible shape graphs: source and selector of a named location uniquely determine the target, i.e. the `next` field of {`p`} always points to the same cell, no matter through what pointer expression {`p`} was reached. So, as both `x->next` and `y->next` end at the same *named* node {`p`}, any further vertices reached via a common set of selectors for e^v and e^w must be the same concrete cells for e^v and e^w, even if they are summarized in an unshared summary location $n_\emptyset$.

The common named node $n_i^v = n_j^w \neq n_\emptyset$ and the common sequence of selectors $sel_{i+1}^v = sel_{j+1}^w \wedge \ldots \wedge sel_N^v = sel_M^w$ will be called the *common tail*. Paths that end at the summary location and do not have a common tail always refer to distinct concrete locations. Paths that end at the summary location but share a common tail, however, refer to the very same concrete location, even if it is represented by the unshared summary location $n_\emptyset$.

Therefore, without taking common selector tails into account, the conservative answer has to be may-alias ($\frac{1}{2}$), even in the case of an unshared summary node $n_\emptyset$.

Both, Algorithm 1 and Algorithm 2 can introduce imprecision in the form of a may-alias. While the may-alias in Algorithm 1 is a result of the different shape graphs in SGS_ℓ that are the product of the execution paths leading to program point ℓ, the imprecision in the second case (Alg. 2) stems from the abstraction of the summary location $n_\emptyset$.

An improved version of the *same*(...) test that takes common tails of access paths p^v, p^w into account eliminates some of the imprecision introduced by the abstraction of the summary location $n_\emptyset$.

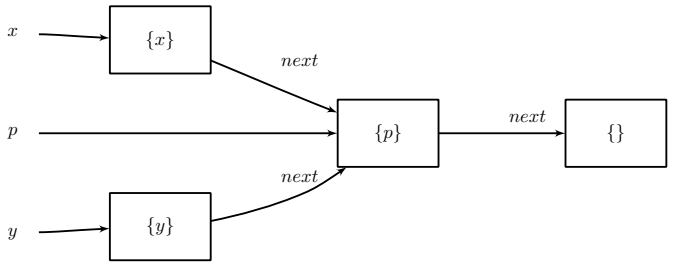


Figure 4.1: A shape graph featuring a common tail.

4.2 Improved Test with Common Tails

The precision of the alias test can be improved if a sequence of selectors at the end of both access paths, called the common tail, is considered instead of only the final vertex. We replace the call to $same_{naive}(n_N^v, n_M^w)$ in Alg. 2 with the more elaborate test $same_{ct}(n_N^v, n_M^w, sel_{1...N}^v, sel_{1...M}^w)$ that takes common tails into account:

$$same_{ct}(n_N^v, n_M^w, ...) = \begin{cases} \text{iff } n_N^v = n_\emptyset \wedge n_M^w = n_\emptyset : \\ \quad 1 \quad \text{iff } \exists i, 1 \leq i \leq N, \\ \qquad \exists j, 1 \leq j \leq M : \\ \qquad n_i^v = n_j^w \neq n_\emptyset \wedge \\ \qquad \bigwedge_{i<i' \leq N, j<j' \leq M} (sel_{i'}^v = sel_{j'}^w) \\ \quad 0 \quad \text{iff } n_\emptyset \notin is \\ \quad \frac{1}{2} \quad \text{otherwise} \\ 1 \quad \text{iff } n_N^v = n_M^w \\ \quad \textit{the same named location} \\ 0 \quad \text{otherwise} \\ \quad \textit{different locations} \end{cases}$$

If two paths p^v and p^w cross at a named location $n_i^v = n_j^w \neq n_\emptyset$ in the shape graph and from there on have the same sequence of selectors $sel_{i'}^v = sel_{j'}^w \wedge \ldots \wedge sel_N^v = sel_M^w$, then both paths not only have the same final vertex but also do the corresponding expressions e^v and e^w refer to the same *concrete*

location, even if the final vertex of the paths is the summary location $n_\emptyset$. This is true irrespective of the summary location's sharing.

Note, however, that a crossing of paths $n_i^v = n_j^w \neq n_\emptyset$ without identical sequences of remaining selectors, i.e. when there is no common tail, allows no such conclusions about the final vertices sel_N^v and sel_M^w of the paths e^v, e^w.

The improvement in $same_{ct}(\ldots)$ over $same_{naive}(\ldots)$ consist of the logic in the case where both paths end at the summary location $n_\emptyset$: if the summary node is reached via a common tail starting at $n_i^v = n_j^w \neq n_\emptyset$ it is certain that the corresponding expressions refer to the same concrete location. This follows from Invariant 3 (source and selector of a named location uniquely determine the target) and the fact that every named abstract location represents only one concrete location. If the same sequence of selectors $sel_{i+1}^v = sel_{j+1}^w \wedge \ldots \wedge sel_N^v = sel_M^w$ is followed starting at the same concrete location $n_i^v = n_j^w$, then the final node also has to be the same *concrete location* $n_N^v = n_M^w$ on both paths.

If there is no such common tail of selectors, the two paths still could meet in the final node of the path, the summary location $n_\emptyset$. Here we again have to distinguish between (a) a shared summary location and (b) an unshared summary location:

(a) $n_\emptyset \in is$: A shared summary location means that two (or more) of the edges pointing to $n_\emptyset$ actually point to identical concrete locations. It cannot, however, be recovered which concrete locations are the shared ones. So the conservative answer ($\frac{1}{2}$) must be given for every pair of paths that end at the shared summary location as even a single edge pointing to $n_\emptyset$ may actually represent two edges pointing to the same concrete location.

(b) $n_\emptyset \notin is$: If the summary location is unshared, all locations summarized by $n_\emptyset$ are distinct. The two paths therefore cannot end at the same concrete location without coming from a common tail. The improved algorithm therefore reports that the two expressions are not aliased.

In cases where a common tail is found the improved alias test $same_{ct}(\ldots)$ identifies e^v and e^w as must-aliases. When there is no common tail, the improved alias test $same_{ct}(\ldots)$ can discern between may aliases and no aliases based on the sharing information. Whereas the naive test $same_{naive}(\ldots)$ can only give may-alias ($\frac{1}{2}$) as conservative answer for every pair of expres-

sions with paths ending at $n_\emptyset$, as it cannot distinguish between presence and absence of a common tail.

The improvement of $same_{ct}(\ldots)$ over the naive alias test $same_{naive}(\ldots)$ is illustrated in Figs. 4.2 and 4.3 showing the results of the algorithms for cases with and without common tails and a shared summary location $n_\emptyset$. The alias test aware of common tails $same_{ct}(\ldots)$ can discern between no alias, may-alias, and must-alias cases, whereas the naive test $same_{naive}(\ldots)$ can only give may-alias ($\frac{1}{2}$) as conservative answer for every pair of expressions with paths ending at $n_\emptyset$. When the paths do not end at the summary location (must aliases in named locations or no aliases due to different locations at the end of the paths) the improved algorithm does not yield better results than the naive alias test.

	unshared $n_\emptyset$	shared $n_\emptyset$
common tail	$\frac{1}{2}$	$\frac{1}{2}$
no common tail	$\frac{1}{2}$	$\frac{1}{2}$

Figure 4.2: Cases in naive algorithm when $n_N^v = n_M^w = n_\emptyset$.

	unshared $n_\emptyset$	shared $n_\emptyset$
common tail	1	1
no common tail	0	$\frac{1}{2}$

Figure 4.3: Cases in improved algorithm when $n_N^v = n_M^w = n_\emptyset$.

Note that common tails can only begin outside the summary location, as only at named nodes it is certain that both expressions refer to the same intermediate cell $n_i^v = n_j^w$. If {p} of Fig. 4.1 was summarized in $n_\emptyset$, for example by assigning a nil-pointer to `p`, there would be no common tail and the information that `x->next` and `y->next` refer to the same concrete cell would be lost.

4.3 Algorithm

An overview of our algorithm to compute alias sets for a given program is given in Fig. 4.4. First, a shape analysis is used to produce the conservative approximation of the heap structure that can possible arise during

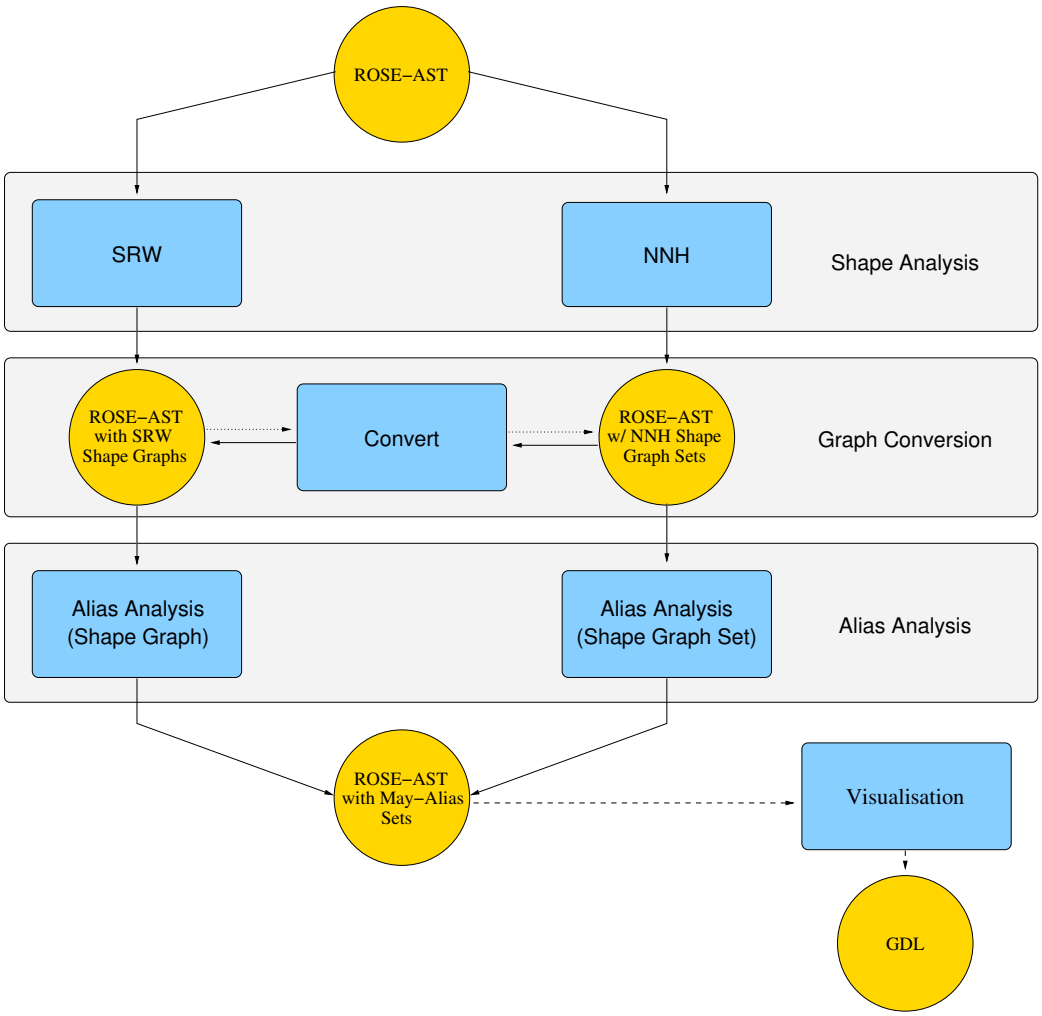


Figure 4.4: Overview of alias analysis algorithm.

actual execution of the program. Depending on shape analysis choice, this approximation either has the form of static shape graphs (SRW), or sets of compatible shape graphs (NNH) that are associated with each program point.

The second pass is to compute alias pairs from these heap abstractions. This can be done directly on nondeterministic SRW static shape graphs – which is fast but imprecise for shape analysis results obtained by the NNH shape analysis. It can also be performed on sets of compatible shape graphs which takes one test per shape graph of the set instead of only a single test per program point, but the more elaborate test pays off as it retains the increased

precision offered by the NNH shape analysis. While possible in theory, it makes no sense to have a nondeterministic SRW static shape graph with already reduced precision converted into a set of compatible shape graphs to perform the exhaustive alias test (cf. Sect. 6.2.1). Fig. 4.4 therefore shows the SRW-to-NNH conversion path as dotted line.

Both alias analyses work on the same principle: at every program point all expressions found in the program, *Exp*, are intersected with expressions also found in the shape representation at that program point. Other expressions may safely be discarded for this program point as they do not represent pointer values or are `null`, and therefore cannot constitute an alias at this program point.

These expressions are then combined pairwise, the order within the pair being unimportant as the alias relation is symmetric. Elements of this expression pair set are used as candidates for the alias tests as described above (cf. Sect. 4.1). This way, all pairs of aliased expressions that are actually *used* in program $\mathbf{S}_\star$ are identified and added as attributes to the ROSE-AST.

Our method does not, however, identify all *possible* expression pairs that are aliases in a specific shape graph at a program point. The answer set to this question is not relevant for practical purposes, generally larger than the set of aliased expressions actually used in the program, and in the presence of backedges in the shape graph, of infinite size.

Example 8 Given the shape graph in Fig. 4.5 we could identify these expression pairs as aliases:

```
(x, x)                  (y, y)
(x->n, x->n)            (y->n, y->n)
(x->n->n, x->n->n)      (y->n->n, y->n->n)
          ⋮                       ⋮

(x, y->n)               (y, x->n)
(x->n, y->n->n)         (y->n, x->n->n)
(x->n->n, y->n->n->n)   (y->n->n, x->n->n->n)
          ⋮                       ⋮

         ...                     ...
```

Each of these "classes" of aliased pairs has an infinite number of elements. Also note that the four classes shown could be extended with "`->n->n`" on

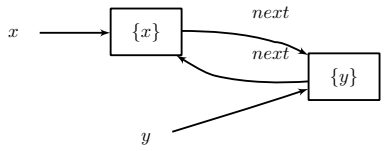


Figure 4.5: A shape graph with an infinite number of aliases.

the right hand side to form an unlimited number of further classes: `(x, x->n->n)`, `(x, y->n->n->n)`, `(x, x->n->n->n->n)`, etc. ad infinitum. ■

The alias sets in the ROSE-AST can be used as information for other analyses, or they can be added to the program as source code annotations. For an example of a program with annotated alias sets see Fig. 4.6. Here the implementation automatically adds may-alias pairs as comments to the original program. Note that in the case of the `while` statement, there are two predecessors: the statement that actually precedes the `while` statement and the very last statement inside the loop, hence the (`t`, `x->next`) alias pair on entry to the `while` statement.

It is also possible to have the program visualized as ICFG with shape graphs associated with every program. The visualization is described in Sect. 5.3 which also shows a sample as rendered by aiSee.

```
class List *reverseList(class List *x) {
  // pre may-aliases :
  class List *y;
  // post may-aliases :

  // pre may-aliases :
  class List *t;
  // post may-aliases :

  // pre may-aliases :
  y = ((0));
  // post may-aliases :

  // pre may-aliases : (t,y -> next),
  while(x != ((0))) {
    // pre may-aliases : (t,y -> next),
    t = y;
    // post may-aliases : (y,t),

    // pre may-aliases : (y,t),
    y = x;
    // post may-aliases : (y,x), (y -> next,(x -> next)),

    // pre may-aliases : (y,x), (y -> next,(x -> next)),
    x = (x -> List::next);
    // post may-aliases : (x,y -> next),

    // pre may-aliases : (x,y -> next),
    y -> List::next = t;
    // post may-aliases : (t,y -> next),
  }
  // post may-aliases : (t,y -> next),

  // pre may-aliases : (t,y -> next),
  t = ((0));
  // post may-aliases :

  // pre may-aliases :
  return y;
  // post may-aliases :
}
```

Figure 4.6: ReverseList function with automatically annotated may-alias sets (analysis parameters: srw-graphset-nokeep-tail, cf. Sect. 6.3), edited to fit the page.

Chapter 5

Implementation

The shape analyses presented were implemented as data flow analyzers for C++ programs using SATIrE[1] This chapter briefly introduces the SATIrE framework before discussing the implementation and limitations of the shape analyses.

5.1 SATIrE

SATIrE (*Static Analysis Tool Integration Engine*) is a framework for combining various tools for static analysis of computer programs. Its aim is to support a wide range of source-level analyses and transformations (including annotations and instrumentation) for C and C++ programs.

SATIrE is being developed at Vienna University of Technology[2] and University of Applied Sciences Technikum Wien[3]. Development of SATIrE has been funded within the ARTIST2 Network of Excellence on Embedded Systems Design[4] and the ALL-TIMES project[5]. [Sch07] describes SATIrE in detail.

Fig. 5.1 shows the architecture of the SATIrE program analysis system. Circles represent the program being analysed as source code or intermediate representation, boxes are analysis- or transformation components. A program in the input language L is translated to a high-level intermediate representation (HL-IR) that acts as the basis for all further analysis and transformation

[1] SATIrE 0.8.5, available at `http://www.complang.tuwien.ac.at/satire/`
[2] `http://www.tuwien.ac.at/`
[3] `http://www.technikum-wien.at/`
[4] `http://www.artist-embedded.org/`
[5] `http://www.all-times.org/`

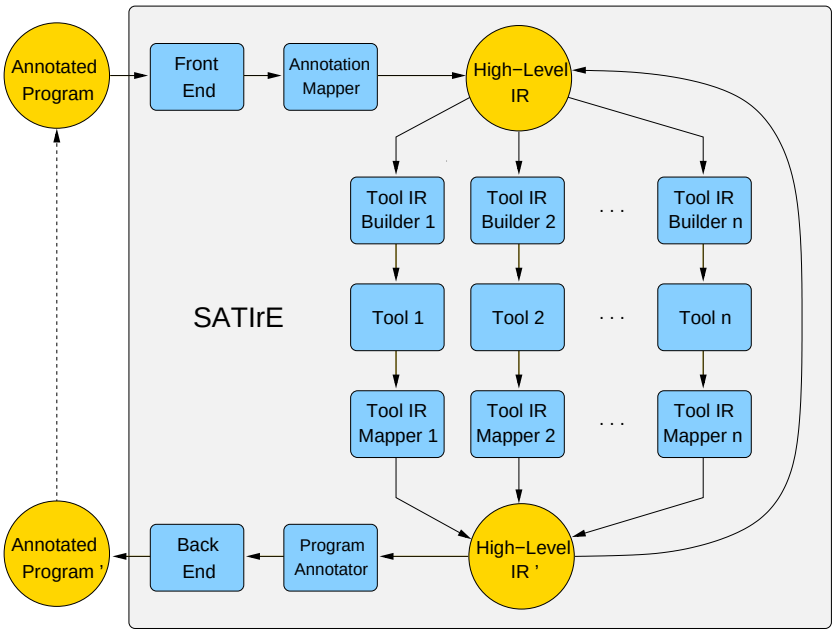


Figure 5.1: Abstract architecture of the SATIrE system. [Sch07, reproduced with permission].

steps. Annotations found in the input program are translated by the annotation mapper to annotations of the same HL-IR.

The strands connecting HL-IR and HL-IR' each represent one of the analysis toolchains integrated into SATIrE. Each chain consists of a preparatory step where the HL-IR is transformed to an intermediate representation specific to the analysis or transformation tool of that chain. After the analysis or transformation is complete, the intermediate representation mapper translates the results back to the high-level intermediate representation. The backedge to the HL-IR on the top shows that the analyses and transformations integrated into SATIrE can be applied iteratively.

After all analysis/transformation steps are performed, the HL-IR' is translated back into a program of the language L. Annotations in HL-IR' are transformed into annotations in the output program to make analysis results persistent. The dotted line shows that annotated programs can again serve as input programs, which is an essential aspect of SATIrE.

At the time of this writing, three major software products are integrated in SATIrE:

- relevant parts of the ROSE source-to-source infrastructure for C++ programs to handle the C/C++ parsing and unparsing,
- the Program Analyzer Generator (PAG) which generates data flow analyzers from high-level specifications, and
- Termite, a tool that can translate programs to and from a term representation suitable for a Prolog interpreter, so that program transformations and analyses can be written in Prolog.

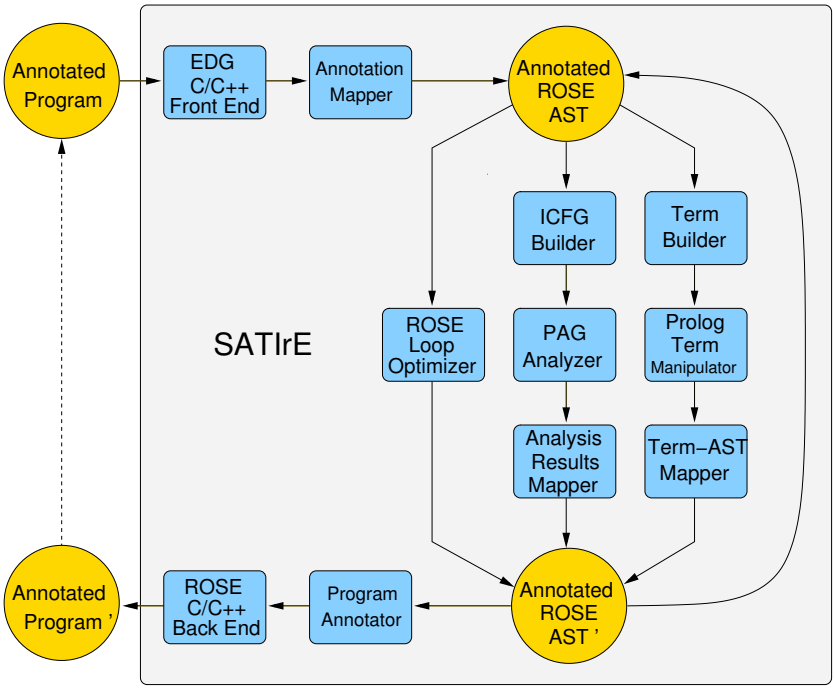


Figure 5.2: Architecture of the SATIrE system with integrated modules. [Sch07, reproduced with permission].

Fig. 5.2 shows where these modules integrated in SATIrE fit into the architecture: ROSE is used as frontend and back end. It uses the Edison Design Group (EDG)[6] C and C++ frontend to parse C/C++ files into an AST. This AST is then decorated with annotations found in the input program and turned into the object-oriented ROSE-AST which, besides holding the annotations, retains line and column information for later unparsing and provides an attribute mechanism to add user-defined information to nodes of the AST.

[6] `http://www.edg.com/`

Work on using Clang[7] as an alternative C frontend is in progress. The representation built by this frontend will still use the ROSE-AST classes.

The ROSE-AST implements the high-level intermediate representation HL-IR central to all SATIrE toolchains. Fig. 5.2 also shows the three toolchains that are now part of the standard SATIrE distribution:

- the ROSE Loop Optimizer is an example for an analysis and transformation that directly operates on the ROSE-AST.
- The second toolchain uses PAG to turn an data flow analysis specification into a data flow analyzer. To do so, an ICFG needs to be built from the ROSE-AST for PAG to work on (ICFG Builder). Once the PAG-generated analyzers are completed, the Analysis Results Mapper writes the results as analysis-specific attributes back into the ROSE-AST.
- In the third toolchain the information of the ROSE-AST is translated to Prolog term syntax. Then an analysis or program transformation written in Prolog may alter the term representation that is then transformed back into ROSE-AST representation.

Once the desired analyses or transformations have been performed, the ROSE backend can unparse the ROSE-AST to C++ source code resembling the original input program. The main differences between the input program and the unparsed program lie in formatting and explicitly inserted type conversions, and preprocessor macros are expanded. Annotations found in the ROSE-AST can be unparsed as pragmas, assertions, or comments. The generated source code can then be passed to a backend compiler that generates machine code, or it could again serve as input program for another SATIrE analysis with some analysis results already made persistent as annotations.

5.1.1 ROSE

The ROSE[8] source-to-source compiler infrastructure provides its users with an intermediate representation of input programs as object-oriented ASTs. Developers can perform simple queries or complex analyses and transformations on these ASTs and have them unparsed into programs of the source language. The source-to-source approach is especially useful to communicate program flaws to the programmer, to warn about bugs and possible optimizations.

[7] `http://clang.llvm.org/`
[8] ROSE 0.9.3a

ROSE covers C/C++, Fortran and some machine languages, but it is not possible to translate between languages, as they each use different intermediate representations. For SATIrE, only the C/C++ parts of ROSE are relevant at the moment.

ROSE has been under development at the Lawrence Livermore National Laboratory (LLNL) since 2001 and is now available under a BSD-style license from the project's website at `http://www.rosecompiler.org/`. Details on ROSE can be found in [SQ03].

5.1.2 PAG

The Program Analyzer Generator (PAG[9]) allows the generation of program analyzers from an analysis specification in a high-level functional input language called FULA.

As of June 1st, 2007, PAG is no longer available as a stand-alone product but used by AbsInt for its other program analysis products.[10] There is, however, still the PAG/WWW[11] interface where static program analysis with PAG can be experienced for the WHILE programming language [NNH99].

Essentially, the analysis specification consists of type declarations, the transfer functions, an analysis direction, the initial analysis value, and the combination operator that defines how results of two or more ICFG nodes are to be combined if there exists a common successor node.

The other components required for a data flow analyzer, like domain functionality, ICFG traversal, or the fixed point algorithm, are provided by PAG. PAG is described in detail in [Mar98].

5.1.3 Termite

The Prolog term integration allows SATIrE users to specify program transformations in Prolog [Int95]. The transformed term representation is then translated back to HL-IR by the intermediate representation mapper.

This approach has been successfully adopted within the COSTA[12] project for performing Worst-Case Execution Time Analysis for a given C program. A detailed description can be found in [Pra07].

[9] PAG 9.0.0.92248

[10] `http://www.absint.com/pag/`

[11] `http://www.program-analysis.com/`

[12] `http://costa.tuwien.ac.at/`

5.2 Shape Analysis Implementation

The implementation of the SRW and NNH shape analyses is divided into four layers, from top to bottom:

1. Transfer Functions (ICFG matching)
2. Normalization of nested pointer expressions
3. Transfer function wrappers that resolve special cases
4. SRW/NNH transfer functions

Only the lowest layer is analysis-specific. The other parts are (apart from the type of the analysis information carrier) equal for both shape analyses and can thus be shared.

As PAGs analysis specification was, for a long time, limited to only a single type definition file and a single file that contained the code, this sharing could conveniently only be exploited by adding a preprocessor step that generates the analysis specification from separate files containing above parts. During this preprocessing the datatype of the carrier is also inserted in the generic definitions of the common code to address PAGs lack of polymorphism.

5.2.1 Transfer Functions

The transfer functions perform the matching on the ICFG. ICFG nodes indicate the type of program statement, edge labels provide information on the type of flow: `normal_edge` for regular flow, `jump_edge` for unconditional jumps, `true_edge` and `false_edge` for branches including loops; `call_edge`, `return_edge`, and `local_edge` for function calls, returns, and for connecting call and return site in the caller, respectively.

For each edge exiting an ICFG node relevant to shape analysis, the appropriate functions provided by the lower layers are called.

Example 9 Fig. 5.3 contains a section of the transfer functions matching ICFG node/edge pairs. It shows how both an assignment statement and the argument assignment before a function call use the same layer 2 function `assign` that can decompose arbitrarily nested pointer expressions.

It can also be seen how different analysis information is passed from the same ICFG node over different edges: the analysis information received from

a node's inputs, `@`, is passed into a function via the `call_edge`, while on the `local_edge` that connects call site with return site, only the least element `bot` is passed.

```
// Assignment Statement
ExprStatement(AssignOp(lhs,rhs)), normal_edge:
  assign(lhs,rhs,@);

// before procedure call: $arg = expr
ArgumentAssignment(lhs,rhs), normal_edge:
  assign(lhs,rhs,@);

// call_edge: from Call to Entry
FunctionCall(funcname,vars), call_edge:
  @;

// local_edge: from Call to Return
FunctionCall(funcname,vars), local_edge:
  bot;
```

Figure 5.3: Section of ICFG-matching transfer functions (Layer 1).

■

For more information on the semantics of the inter-procedural ICFG nodes see Sect. 3.5.

5.2.2 Normalization Code

The method of normalizing arbitrarily nested pointer expressions by introducing temporary variables was already described in Sect. 3.4. The function `assign` implements this method.

Pointer expressions on the left and possibly on the right hand side of every assignment are parsed to build a list of transfer function wrapper calls that have, when actually called, the desired effect on the carrier. At the same time, function calls that again remove the temporary variables after use are appended to this operation list.

In a second pass, the list is traversed calling the transfer function wrapper functions to modify the analysis information carrier.

Explicitly creating a list of transfer functions to be called helps working around the missing support for polymorphism in PAG: the complex construction of the list is thereby made independent of the carrier type and may therefore be shared for different carrier types. Analyses only have to traverse the list calling the respective transfer functions that operate on the carrier type.

As a benefit, it also helps balancing the introduction and elimination of temporary variables.

5.2.3 Transfer Function Wrappers

This is the last layer before the actual transfer functions of the analyses are invoked. The transfer function wrappers pick up on normalization where the previous layer resolving arbitrarily nested pointer expression stopped: the special cases with pointer expressions that have only one level of indirection. This layer ensures that the variables on either side of an assignment are distinct (cf. Sect. 3.3). It thus bridges the gap to the elementary transfer functions of the shape analyses as described in Sect. 3.2 (SRW) and Sect. 3.1 (NNH).

5.2.4 SRW/NNH Transfer Functions

This layer implements the six elementary transfer functions (cf. Sect. 3) of the shape analyses. As all the special cases of nested pointer expressions are already taken care of by the upper layers, the code closely resembles the analysis specifications found in [SRW98] and [NNH99].

5.3 Visualization

PAG offers a convenient way to generate a visualization of the program under analysis. It automatically emits a description of the ICFG in the graph description language (GDL), the format used by AbsInt's graph viewer aiSee.[13]

The visualization of the carrier itself can be controlled by providing an implementation of the function `dfi_write_` that will be called by PAG when `DFI_WRITE` is defined during compilation. Otherwise PAG uses its own im-

[13] http://www.aisee.com/

plementation of that function which results in the carrier being inserted as text into the ICFG.

Fig. 5.4 shows a shape graph as rendered by aiSee. Only the visualization of the carrier, that is what is inside the box next to the edge labels, can be controlled by the developer, PAG provides the rest.

5.4 Alias Information

The alias information for a program is obtained in a postprocessing pass after the shape analysis has computed its heap approximation for every program point ℓ of the program $\mathbf{S}_\star$. Actually, the analysis information is computed for the points immediately before and immediately after ℓ, approximating the state of the heap before and after the execution of the statement at position ℓ.

The `AliasPairsAnnotator` class implements the AST traversal that adds the identified alias pairs as attributes to the ROSE-AST. The actual alias testing functions are implemented in PAG's convenient FULA language also used for the analysis specification.

5.5 Shape Analyses and C++

The NNH analysis was developed for an extended version of the WHILE-language (cf. [NNH99]) only. SRW is formulated with no concrete syntax in mind but rather as analysis applicable to all programs manipulating objects in the heap through pointers.

Our implementation of both shape analyses has been extended to inter-procedural program analysis (cf. Sect. 3.5) of programs written in a subset of the C++ programming language [Int03] with the detection of heap based aliases in mind.

When an analysis encounters a language construct that does not lie within its language subset, it aborts the analysis and reports an error because language constructs unknown to the analysis could alter the semantics of the program being analysed in a way that prevents the analysis from computing a conservative approximation. Merely ignoring unknown constructs would make analysis results unreliable.

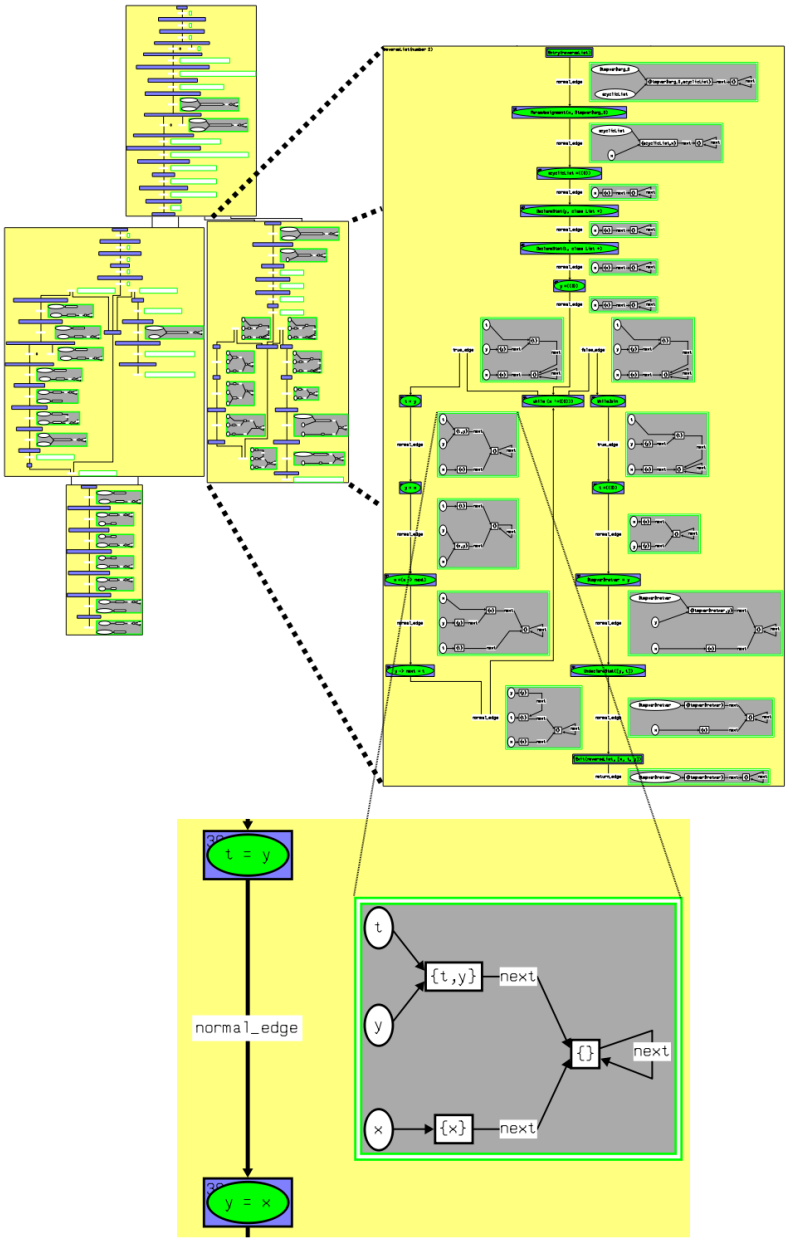


Figure 5.4: Visualization of the reverselist C++ program with shape graphs in aiSee. The annotated ICFG is shown zoomed to program-, function-, and statement-level.

This chapter outlines the subset of C++ that is currently supported by our implementation of the shape analyses. The structure of this chapter loosely

follows that of the C++ Standard [Int03], with references to the relevant clauses of the standard included in the reference, e.g. [Int03, 1 intro].

5.5.1 Types

C++ programs are constructs that manipulate objects. The basics of objects have already been covered in Sect. 2.2 where the definitions were required for the discussion of the shape analysis carrier.

Objects are described by their type, which is either a fundamental type or a compound type. The fundamental types consist of integral (integer) types: `bool`, `char`, `wchar_t`, signed integer types (`signed char`, `short int`, `int`, `long int`), unsigned integer types (`unsigned char`, `unsigned short int`, `unsigned int`, `unsigned long int`), and the `void` type, which has an empty set of values [Int03, 3.9.1 basic.fundamental].

Compound types can be constructed as arrays of a given type, classes, unions, enumerations, functions, pointers, references, and pointers to non-static class members (pointers to static class members are treated as ordinary pointers to objects or functions) [Int03, 3.9.2 basic.compound]. While the distinction between pointer type and pointer-to-member type [Int03, 8.3.3 dcl.mptr] is relevant to the implementation of a C++ compiler, it is irrelevant to a shape analysis as the differences lie in syntax and memory layout of the types only, conceptually they just are pointers.

We now turn to the individual types in detail.

Arrays

An array is an object that contains a contiguously allocated non-empty set of n sub-objects [Int03, 8.3.4 dcl.array]. The sub-objects are numbered from 0 to $n - 1$ and can be accessed through the array subscript operator `[]`. Despite constant propagation and array-range analysis [YH04] the subscript used to access an element of an array is often unknown at compile time.

When this is the case, the current implementation aborts the analysis.

To cover array accesses in a safe manner, a shape analysis could ignore the subscripts altogether, treating the whole array as a single location. Assignments to sub-objects, though, would not be allowed to perform strong updates as it were not certain which of the existing elements had to be replaced with the new value.

Still, it must be guaranteed that the subscript remains within the bounds of the array – otherwise, assignments to the array would change arbitrary memory locations, potentially destroying the shape of the heap.

Classes

A class definition introduces a new type. Classes may contain data members and methods [Int03, 9.2 class.mem]. A structure is a class defined with `struct` [Int03, 9 class]. Its member variables and methods are public by default.

Member access to classes or structs is mapped to the use of selectors in the original shape analysis specifications. Both, the dot-deref operator combination `(*ptr).sel` and the operator `ptr->sel` are understood by the analysis implementations. Member access for classes that are created on the stack is not supported by the analyses.

Unions

A union is a class defined with `union`. Its member variables and methods are public by default and it holds only one member at a time [Int03, 9.5 class.union].

Unions are not part of the language subset covered by our implementation. Because the focus of this thesis lies on aliases created in the heap, the complexity introduced by a conservative treatment of unions seems disproportionate. This decision shall be illustrated with two examples:

Example 10 Fig. 5.5 shows a C++ program where a union overlaps two classes with the same memory layout `ListA` and `ListB`. One member of the union, `a`, is assigned to point to a newly created object in the heap. In the next statement the other member of the union, `b`, is used to deallocate this object again.

While this code is perfectly valid C++, the shape analyses have their difficulties. The problem lies in the abstraction used by the shape analyses: the allocation statement creates a heap node labelled with one variable id and then tries to remove that same heap node using a different variable id. The information that both `u.a` and `u.b` refer to the same location is not available without running an alias analysis upfront.

If the data members of the union were not of the same memory layout, the code would still be valid C++ but the effect of writing to one member and reading from another would be even more complex.

```
class ListA {
public:
  ListA(int i):next(0) {val=i;}
  ListA* next;
  int val;
};

class ListB {
public:
  ListB(int i):next(0) {val=i;}
  ListB* next;
  int val;
};

union U {
  ListA *a;
  ListB *b;
};

int main(int argc, char **argv) {
  U u;
  u.a = new ListA(42);
  delete u.b;

  return 0;
}
```

Figure 5.5: A C++ program with a union.

■

Example 11 Anonymous unions are used do declare variables that share the same address but otherwise can be used like ordinary variables. Members of an anonymous union are created in the scope in which the anonymous union is declared [Int03, 9.5 class.union]. A sample program is shown in Fig. 5.6.

As the use of `p` and `i` deliberately looks like normal variables, the information that they alias must, again, be retrieved by another analysis upfront.

```
class List {
public:
  List():next(0) {}
  List* next;
};

int main(int argc, char **argv) {
  union {
    List *p;
    int i;
  };
  p = new List();
  // ...
  i = 7;
  // ...
}
```

Figure 5.6: A C++ program with an anonymous union.

■

Enumerations and Fundamental Types

An enumeration is a type with named constants. Enumeration types are implemented by an underlying type that fits all enumeration values. It is implementation defined which type is used as underlying type [Int03, 7.2 dcl.enum].

Fundamental types do not require special treatment by the shape analysis since they cannot be the source of heap edges and the shape of the heap is everything the shape analyses are interested in. Objects of fundamental type are only relevant when allocated in the heap and referred to by a pointer.

Pointers

Pointer values are at the center of interest to shape analyses. A pointer holds as its value the address of an object [Int03, 3.9.2 basic.compound, 8.3.1

dcl.ptr]. Pointers to objects in the heap account for the edges found in shape graphs while pointers to objects allocated on the stack are not considered by the shape analyses.

Pointer arithmetic [Int03, 5.7 expr.add] is not supported by our implementation.

Arbitrarily nested pointer expressions consisting of only the dot, the dereference and the combined dot-deref operator `->` are transformed into a series of elementary transfer functions using temporary variables. The method used is illustrated in Sect. 3.4.

Neither analysis features a transfer function that covers the address-of operator (`&`) found in C and C++. The address-of operator is therefore not part of the language subset supported by the analyses.

The absence of the address-of operator guarantees that all pointers refer to objects in the heap as arbitrary pointers can only be created by the address-of operator or by functions that explicitly operate on memory. These functions, however, have unknown bodies, so the analysis must not rely on its information when encountering such a function.

Without the address-of operator there are no pointers to pointers and call-by-value parameter passing cannot introduce aliases.

References

A *reference* can be thought of an additional name for an object. There are no references to references, pointers to references or arrays of references, as a reference is not an object. It is unspecified whether or not a reference requires storage [Int03, 8.3.2 dcl.ref].

References are used for call-by-reference function parameters. Reference function parameters directly operate on the argument variables, not on copies. Changes made to a reference parameter inside a function thus directly affect the values of the outside arguments.

References can also be used as return value of functions, turning the function call into an lvalue [Int03, 5.2.2 expr.call].

In general, references only introduce another name for an existing variable. All of the above uses introduce aliasing but not through pointers in the heap. These aliases can therefore not be detected by shape analysis based alias analyses but rather have to be collected separately. This separate reference

based alias analysis is not part of our implementation, references are thus not included in the language subset.

5.5.2 Functions

The method that allows the conservative treatment of function calls has already been described in Sect. 3.5. C++ specifics are discussed here.

Member functions (methods), constructors, and destructors are treated as normal function calls by the ICFG generator with the exception that they receive the implicit parameter `this` via the temporary variable `$tmpvar$this`. The ICFG generator represents overloaded functions, overloaded operators, virtual functions, overloaded virtual functions, virtual destructors, and default arguments in a safe manner. Support for copy constructors is currently under development but not of concern to the shape analyses themselves.

Functions whose implementation is unknown to the analysis – i.e. when they are called via function pointer or because the definition is not included in the program under analysis – are represented in the ICFG by an `ExternalCall` node. As the code of these functions is unknown it cannot be told what it directly or indirectly (through the call of other functions) modifies that is relevant to the analysis. Analyses that encounter such a function call therefore must invalidate their analysis information to ensure a conservative result. Library functions that explicitly operate on memory, i.e. `memcpy`, `memmove`, `alloc` and variations, etc., also fall into this category.

How the analyses deal with call-by-reference or call-by-value using the address of an object is treated in Sect. 5.5.1 under References and Pointers, respectively.

5.5.3 Control Flow and Templates

Statements that influence the control flow of the program under analysis, i.e. `if/else`, `switch`, `while`, `for`, `do/until`, `break`, `continue`, `return`, and `goto` are interpreted by the ICFG builder that adds the control flow edges accordingly. The analysis itself does not interfere with these ICFG nodes.

The same is true for exceptions. However, they are not yet covered properly in the ICFG construction, so the subset of C++ covered by our implementation does not yet support exceptions.

Templates are, like preprocessor directives, already resolved at the frontend level.

Chapter 6

Evaluation

This chapter presents an evaluation of the performance of the analysis variations on certain example input programs. The variations of analyses are introduced and compared to each other in terms of cost and precision.

The size of may-alias sets is used as the primary measure of precision for shape analysis results. A shape analysis could say that every expression is aliased with every other expression, which would be safe but useless. The smaller the set of *possibly* aliased expressions is, the more precise the result. Small sets of may-aliases stem from shape graphs that contain few excess edges. The number of edges, however, is not a sufficiently good measure for the comparison of SRW and NNH shape analyses as they use different shape representations. Converting either representation to the other and then counting the number of edges does not correctly reflect the precision of the computed approximation.

When the NNH analysis is able to identify a must-alias, this information is used as secondary measure of precision. Here, the larger set is the better result.

6.1 Shape Analysis Variations

The high level of abstraction innate to SRW static shape graphs allows for very compact representation but it is also the major source of imprecision in the SRW analysis.

6.1.1 Crosstalk

For all execution paths only a single static shape graph is used in SRW which leads to the combination of shape graph parts that would be kept separate in NNH using shape graph sets. This effect will be called "crosstalk" and shall be illustrated with an example:

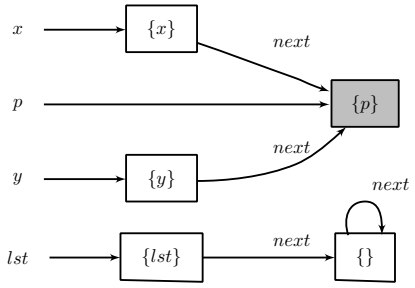


Figure 6.1: SRW static shape graph with shared named heap node.

Example 12 Fig. 6.1 shows a simple static shape graph where x and y are pointers that indirectly refer to the same heap location $n_{\{p\}}$. Because $n_{\{p\}}$ is target of two heap edges it is marked as shared (gray). Independent of this first structure there is an acyclic list pointed to by lst. That the list is free of cycles is deduced from the summary location being unshared. If no cell in the list is pointed to by two (or more) cells then there cannot possibly be any cycles. Also, it is trivial to see, that this acyclic list cannot be reached from either x or y.

A single assignment can alter this picture drastically. Compare the shape graph in Fig. 6.2 that is the product of the assignment `p = null;` applied to above SRW shape graph.

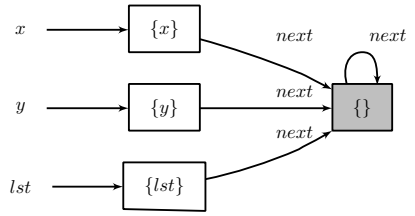


Figure 6.2: SRW static shape graph with shared summary node.

The previously named heap location $n_{\{p\}}$ is now summarized into $n_{\emptyset}$. As x and y still indirectly refer to the same heap location there has to be at

least one shared cell inside the summary location. SRW does not, however, distinguish between a single shared location in $n_\emptyset$ and all locations clustered into the summary location being the same location.

Much of the information in the first graph is now lost:

- It is no longer certain that x and y indirectly point to the same location. All we can deduce is that x, y and lst point to abstract locations that have no names. Due to the sharing information at least two edges going into $n_\emptyset$ actually refer to the same location, but that these two edges are `x->next` and `y->next` is lost. Neither is it preserved that the list cannot be reached from x or y.
- The targets of `x->next` and `y->next` are no longer just cells but could as well be lists of arbitrary length.
- The list pointed to by lst could now as well have cycles.

This is just one example of the imprecision introduced by letting nodes slip into the summary location. There could be even worse cases, for instance with $n_{\{p\}}$ being the head of another list with selectors that use other names than the selectors of x and y. Then the two lists would be indistinguishable from a list with mixed selectors, or a linked hash table structure, or even a tree.

The NNH analysis does significantly better in this case. Above example will be referred to as the `crosstalk.C` testcase (cf. Sect. 6.3). ■

6.1.2 There is no Must-Alias Information in SRW

Consider the SRW static shape graphs shown in Figs. 6.3 and 6.4. The first shape graph is computed in the true-branch of an `If`, the other shape graph comes from the false-branch. As we cannot at compile time decide the result of the condition, the analysis has to conservatively cover both branches. The resulting SRW shape graph after joining both branches is shown in Fig. 6.5.

While the connected heap structure consisting of $n_{\{b\}}$, $n_{\{x\}}$, and $n_{\{y\}}$ can be found on both branches, only two of the remaining three nodes $n_{\{a\}}$, $n_{\{v\}}$, and $n_{\{w\}}$ were connected on either branch. It is only after the joining of the branches that $n_{\{a\}}$, $n_{\{v\}}$, and $n_{\{w\}}$ become connected in the SRW shape graph.

The implication of this is that there is no must-alias information available in SRW static shape graphs.

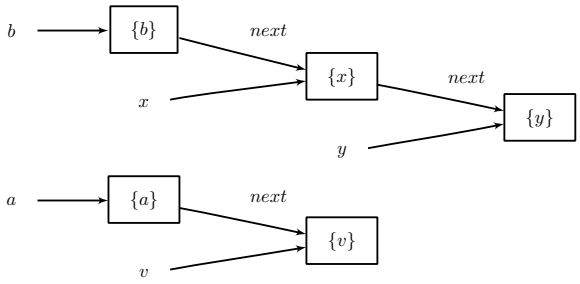


Figure 6.3: SRW static shape graph at the end of a true-branch.

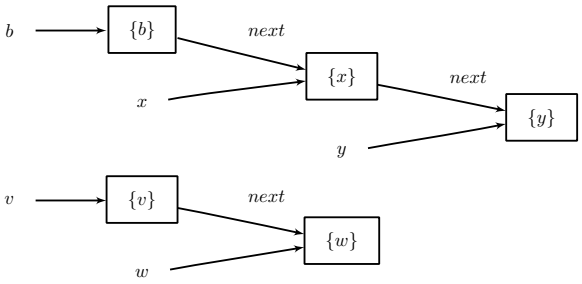


Figure 6.4: SRW static shape graph at the end of a false-branch.

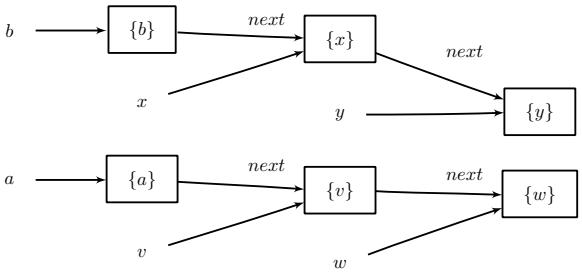


Figure 6.5: SRW static shape graph at program point ℓ, immediately after the branches are joined.

Assuming there is must-alias information contained in SRW static shape graphs we would deduce that the expressions `b->next->next` and `y` are must-aliases at program point ℓ which we take to be right after the join. This would be based on the observation that node $n_{\{y\}}$ is reachable via `b->next->next` on all execution paths leading to ℓ, hence `y` and `b->next->next` must alias at that point. The problem is that this information cannot be based on an interpretion of the SRW static shape graph, because the previously uncon-

nected nodes $n_{\{a\}}$ and $n_{\{w\}}$ are represented in the same way as the connected nodes $n_{\{b\}}$ and $n_{\{y\}}$. If we were to report a must-alias for the expression pair (`b->next->next`,`y`), we would also have to identify (`a->next->next`,`w`) as must-alias, which would be incorrect.

The safe answer for both cases thus is that the nodes represented as connected only constitute may-aliases. Every edge of an SRW static shape graph at program point ℓ could be absent on at least one execution path leading to ℓ. The SRW static shape graph thus represents a set of deterministic shape graphs of which at least one does not contain that edge absent on at least one execution path. As the SRW static shape graph, however, does not distinguish between "certain" edges and edges that may be missing, all edges need to be assumed to be absent on at least one execution path leading to ℓ. Therefore SRW static shape graphs do not contain must-alias information.

Note that for this reason, the extraction of deterministic graphs from a SRW static shape graph includes almost as many graphs as are in the powerset of the static shape graph (less those graphs that are incompatible). For details on the conversion see Sect. 2.7.

The NNH shape analysis would again do much better: If the two shape graphs shown in Figs. 6.3 and 6.4 were the elements in the set of compatible shape graphs at the end of the branches, the analysis information carrier at the join node would simply be the union of both shape graph sets. The graphs themselves remain unmodified and the alias test can identify (`b->next->next`,`y`) as must-alias, because the expressions refer to the same abstract location in all shape graphs, while (`a->next->next`,`w`) is not even a may-alias. Both results come without the uncertainty of a may-alias and thus are more precise, while the results obtained by SRW are still on the safe side.

There is, however, some kind of limited must-alias information in SRW static shape graphs associated with state-edges and common tails: Consider the SRW static shape graph in Fig. 6.6. In this shape graph it is certain that `x` and `y` always point to the same location or they both are null, but neither `x` nor `y` can point to a location without the other pointing to the very same location. When the abstract location $n_{\{x,y\}}$ is the starting point for a common tail (cf. Sect. 4.2), all locations passed along the common tail are also guaranteed to be identical for the corresponding elements of the expressions starting at `x` and `y` – as long as no pointer is null, in which case it would be null for both `x` and `y`. This kind of must-alias information is though limited as it can not be deduced from the SRW static shape graph whether pointers are null or pointing to locations.

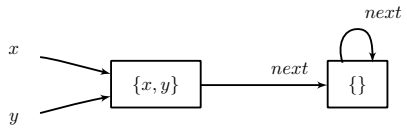


Figure 6.6: SRW static shape graph with some kind of must-alias information.

6.1.3 SRW Does Not Always Perform Strong Update

While [SRW98] claim that their algorithm always performs a strong update for statements of the form ⟦`x->sel = y`⟧, even when `x` does not point to a unique shape node, this does not, however, hold for all transfer functions.

Example 13 Consider the sequence of static shape graphs in Figs. 6.7–6.10. They show the abstract heap before and after the execution of these statements: `tmp = x->next`, `tmp->next = target`, and `tmp = null`.

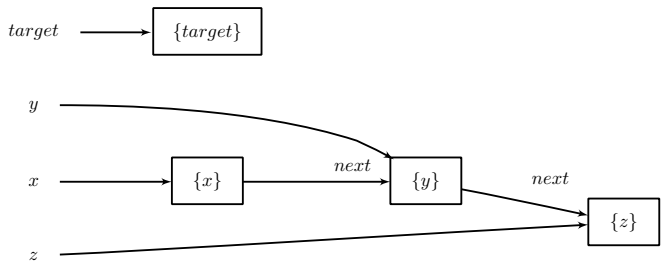


Figure 6.7: SRW static shape graph.

After the execution of the last statement, the shape graph has two edges going out of node $n_{\{y\}}$: one points to $n_{\{target\}}$ and the other one still points to $n_{\{z\}}$, the same node that $n_{\{y\}}$ pointed to at the beginning. The three statements shown above are what `x->next->next = target` would be transformed into, hence for ⟦`x->next->next = target`⟧ there is no strong update.

This is not a mistake but follows from the absence of certainty in SRW static shape graphs (cf. Sect. 6.1.2): If `x` or `x->next` are null-pointers, `tmp = x->next` would have no effect on the shape graph in Fig. 6.7 and $n_{\{y\}}$ would not have `tmp` as further variable-edge pointing to it, so the node's name remains $n_{\{y\}}$. If, on the other side, $(n_{\{x\}}, next, n_{\{y\}})$ is present in the static shape graph, i.e. `x->next` points to $n_{\{y\}}$, then `tmp = x->next` has the effect of renaming $n_{\{y\}}$ into $n_{\{tmp,y\}}$ because now the variable `tmp` also directly

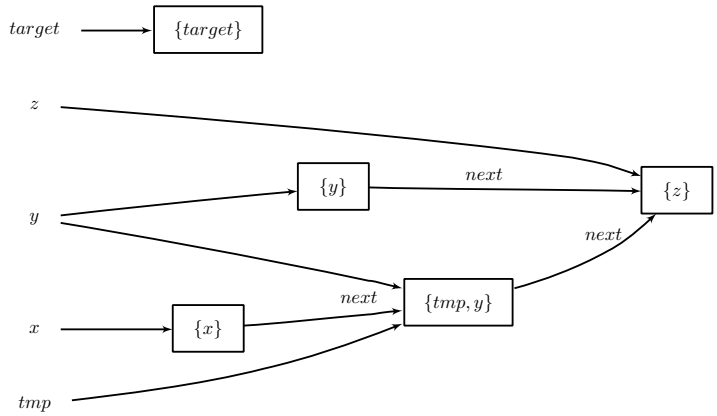


Figure 6.8: SRW static shape graph after ⟦`tmp = x->next`⟧.

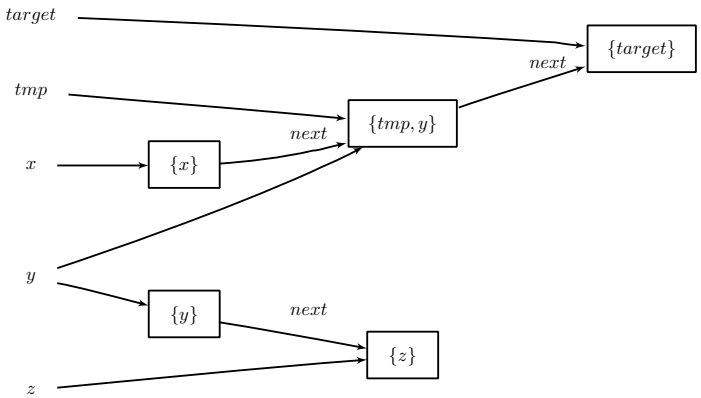


Figure 6.9: SRW static shape graph after ⟦`tmp->next = target`⟧.

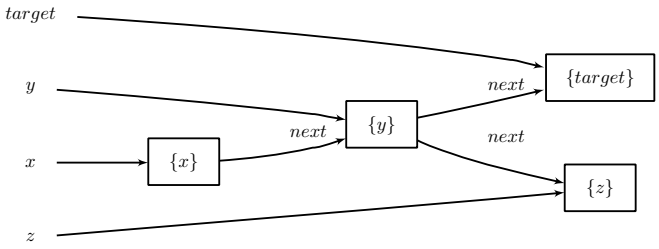


Figure 6.10: SRW static shape graph after ⟦`tmp = null`⟧.

points to that node. To conservatively cover both cases, the resulting SRW static shape graph has to be as shown in Fig. 6.8.

For the next statement, `tmp->next = target`, the algorithm is again able to perform a strong update, but this does not modify the edges going out from $n_{\{y\}}$. When the temporary variable `tmp` is then removed in the final step, the nodes $n_{\{tmp,y\}}$ and $n_{\{y\}}$ collapse. The resulting graph has both the new heap edge $(n_{\{y\}}, next, n_{\{target\}})$ and the old edge $(n_{\{y\}}, next, n_{\{z\}})$, so there is no strong update for $[\![$`x->next->next = target`$]\!]$. ■

Programs that contain above sequence of statements cause the SRW shape analysis algorithm to perform a weak update only. While [SRW98] is often referred to as the first shape analysis that is *always* able to perform strong updates, the paper only claims this for the $[\![$`x->sel = y`$]\!]$ transfer function [SRW98, pp. 45f]. The place where the SRW shape analysis loses precision by not being able to perform a strong update is but, as shown above, the transfer function $[\![$`x = y->sel`$]\!]$.

6.1.4 Temporary Variables

Temporary variables are required in the normalization process (cf. Sects. 3.3 and 3.4) but they also have a positive effect on the precision of the analyses as they prevent nodes from being summarized into $n_{\emptyset}$.

Usually temporary variables are removed after they are read from once as they are then no longer needed for the transfer functions. But we can also keep them in the shape graph as support nodes, providing additional names for locations.

This feature is still experimental, however. A strategy to decide when to delete temporary variables needs to be devised as keeping all temporary variables indefinitely prevents the analysis from terminating as it never reaches a fixpoint.

6.2 Alias Analysis Variations

The computation of aliases consumes a large fraction of the overall analysis runtime.

For a program that contains m pointer expressions there exist $\frac{m*(m+1)}{2}$ pairwise combinations of pointer expressions under commutativity that could alias each other. Note that compound pointer expressions like `x->next->next` count as the expression and all sub-expressions, i.e. there are three elementary pointer expressions in the example: `x`, `x->next`, and `x->next->next`.

In order to check whether such a pair of pointer expressions constitutes a may-alias (or must-alias) pair, the alias test has to be performed on every shape graph in the NNH set of compatible shape graphs. The SRW static shape graph may be converted into the same set representation to perform the test on each deterministic shape graph *contained* in the static shape graph for a given program point.

When the SRW shape graph has a size of k edges and contains l shared nodes, the number of contained deterministic shape graphs n_{dg} is (at most):

$$n_{dg} = 2^k * 2^l$$

Note that an exhaustive test in all deterministic shape graphs is only required in the worst case. When searching for a may-alias, the test is abandoned as soon as the first constituent shape graph yields $\frac{1}{2}$ (may-alias) as its result. As it is sufficient for the expressions e_i and e_j to be may-aliases if they are may-aliases in a *single* shape graph, the testing of the remaining shape graphs can be aborted at this point.

When searching for a must-alias, 0 (no alias) and $\frac{1}{2}$ (may-alias) are the results for an early exit, as e_i and e_j can only be must-aliases if they are so on every execution path leading to this program point.

This procedure has to be performed (at most) four times for each statement for which aliasing information is required: for the may-alias and must-alias information immediately before and after the statement. Note that, as $\mathbf{Must}_\ell \subseteq \mathbf{May}_\ell$ holds, the must-alias test need only be performed if the may-alias test was successful (cf. Sect. 4).

For a program with n statements, an average graph size of k edges, l shared nodes, and m pointer expressions this amounts to n_{tests}, the number of times to check whether two expressions describe a walk to the same end node in a shape graph:

$$n_{tests} = n * 2^k * 2^l * \frac{m * (m+1)}{2} * 4$$

For a small test program with $n = 40$ statements, $m = 19$ pointer expressions, and an average graph size of $k = 30$ edges with $l = 1$ shared node this yields:

$$\begin{aligned} n_{tests} &= 40 * 2^{30} * 2^1 * \tfrac{19*(19+1)}{2} * 4 \\ &= 40 * 2^{30} * 2 * 190 * 4 \\ &\approx 6.5 * 10^{13} \end{aligned}$$

The cost of the alias computation predominantly depends on the number of deterministic shape graphs represented by the static shape graph. The worst case is bounded by $O(2^{k+l})$. Fortunately we can further reduce the number of tests without losing precision, at least in the case of analysis results obtained from SRW.

6.2.1 Alias Computation on Static Shape Graphs

Instead of extracting the set of compatible shape graphs from an SRW static shape graph to resolve indeterminism and the possible absence of edges, the alias computation can also be performed directly on the SRW graph representation, saving the exponential cost of above operation.

The aliasing test still has to be performed for every pointer expression pair at every statement of interest, but only on a single shape graph:

$$n'_{tests} = n * \frac{m * (m+1)}{2} * 4$$

The complexity of the alias computation on a single static shape graph is thus bounded by the number of pointer expressions: $O(m^2)$.

Note that the alias test needs to be adapted when performed on an indeterministic shape graph whose edges cannot be depended upon. It is still based on the same observation: if two expressions each describe a walk in the shape graph that has the same terminal end, then the two expressions are aliases in that graph. But as the edges may not be depended upon the test must not yield must-aliases.

When this simplified test is used for a NNH shape graph set converted to an SRW static shape graph, the information required to identify must-aliases is lost during the conversion and no must-aliases can be identified. Not identifying any must-aliases is a conservative answer. For the computation of may-aliases, however, the simplified test performed directly on the SRW static shape graph is *no less precise* than working on the set of compatible shape graphs!

Assume all contained shape graphs g_i are computed from the static shape graph G to perform the may-alias test. Two expressions are identified as may-aliases if they possibly alias in a single g_i. It is therefore sufficient to use the least precise g_i for the test, as all other g_j will only contain subsets of the may-aliases found in g_i. The least precise shape graph contained in G is the one where all edges that could possibly be absent are actually

present. This $g_i = G$ so the test can be directly performed on G without losing precision. Due to the indeterminism still contained in g_i, the alias test occasionally needs to backtrack when two walks are described by the same expression.

Even NNH shape analysis results could (inexpensively) be converted to a SRW static shape graph to profit from the lower computational complexity of the simplified test. Note that the conversion process, however, will make the results less precise than what could have been found testing on the set of compatible shape graphs. The difference lies in the shape graphs *not contained* in the results obtained from the NNH analysis. This information is lost when joining independent graphs in a single static shape graph (cf. "crosstalk" in Sect. 6.1.1).

6.3 Measurements

In summary, the precision and cost of the heap alias analysis can be adjusted in either the underlying shape analysis or the extraction of the aliases themselves:

SRW shape analysis (`srw`) The SRW algorithm [SRW98] is used as underlying shape analysis to compute the heap representation (cf. Sect. 1.2.1).

NNH shape analysis (`nnh`) The NNH algorithm [NNH99] is used as underlying shape analysis to compute the heap representation (cf. Sect. 1.2.2).

Retaining temporary variables (`keep`) Do not delete temporary variables immediately to prevent heap nodes from slipping into the summary node (cf. Sect. 6.1.4).

Deleting temporary variables (`nokeep`) Temporary variables are removed from the shape graph after they are read from the first time in order to keep the shape graph small.

Aliases from shape graph set (`graphset`) The alias test is performed on every member of the shape graph set. Note that this increases precision only if the shape graph set was obtained from the NNH analysis (cf. Sect. 6.2.1).

Aliases from static shape graph (`nographset`) The alias test is performed directly on the static shape graph for considerably reduced computation cost (cf. Sect. 6.2.1).

Alias test with "common tail" (`tail`) If two expressions describe shape graph walks that end at the summary node, the additional "common tail" test could be used to increase precision (cf. Sect. 4.2).

Alias test with no "common tail" (`notail`) The "common tail" extension is not used.

Context-sensitive analysis (callstring length ≥ 1) Call strings are used to keep the information of different call sites separate (cf. Sect. 1.1.4).

Context insensitive analysis (no callstring) The analysis is performed without context information (cf. Sect. 1.1.4).

With these orthogonal choices there are in total 16 different implementations of the shape analysis based alias analyses to be compared in order to measure the individual and combined effects of the optimizations on analysis runtime and precision. Each of these analyses can be run as context-insensitive or context-sensitive analysis, depending on the chosen length of the callstring.

The measurements were taken on a machine equipped with an 8-core Intel Xeon CPU (E5450) with 3.00 GHz clock frequency. PAG was given 1024 megabytes of storage to reduce garbage collection time (`--pag-memsize-mb=1024 --pag-memsize-grow=30`) but never used more than 40 MB, except in the RecSplice case where `nnh-nokeep` used 62, and `nnh-keep` variants used even 300 megabytes. Compared with PAGs default memory settings (`--pag-memsize-mb=5 --pag-memsize-grow=30`) this reduced analysis runtime in all cases. Notable was the RecSplice case, which is the most memory-intense case. There the analysis runtime was almost cut in half by allocating enough memory upfront to avoid garbage collection interruptions. The time required for the alias extraction was not affected by this.

Table 6.1 shows the number of may-alias pairs identified by the parametrized alias analyses for our benchmark. Smaller numbers constitute more precise results. The variants of analyses are listed in the column headers. Note that `graphset` is only performed for the NNH analysis, as `graphset` cannot improve precision over `nographset` for SRW shape graphs (cf. Sect. 6.2.1).

The results shown in table 6.1 were obtained with context-sensitivity implemented by callstrings of length 3. As expected, results obtained without context-sensitivity were less precise. Interestingly, however, the context-insensitive analyses also took drastically *more* time to complete. As all objects are created by constructor calls which are handled like function calls, the analysis information at every allocation site is merged with that at every other allocation site when no context information is used. As a result, shape

Testcase	nnh-graphset-keep-tail	nnh-graphset-keep-notail	nnh-graphset-nokeep-tail	nnh-graphset-nokeep-notail	nnh-nographset-keep-tail	nnh-nographset-keep-notail	nnh-nographset-nokeep-tail	nnh-nographset-nokeep-notail	srw-nographset-keep-tail	srw-nographset-keep-notail	srw-nographset-nokeep-tail	srw-nographset-nokeep-notail
Analysis Parameters												
1: artificialListCreation	48	48	72	85	48	48	85	85	48	48	85	85
2: artificialSumWithDS	18	18	33	43	18	18	43	43	18	18	43	43
3: artificialSumWithDS_unrolled	7	7	52	97	7	7	97	97	7	7	97	97
4: srw98create	7	7	7	7	7	7	7	7	7	7	7	7
5: srw98insert	44	44	44	44	44	44	44	44	44	44	44	44
6: srw98reverse	44	47	44	47	47	47	47	47	47	47	47	47
7: independent-lists	6	76	4	79	76	76	79	79	76	76	79	79
8: semi-independent-lists	43	146	41	162	167	167	170	170	167	167	170	170
9: append-if	20	20	20	20	28	28	28	28	28	28	28	28
10: common-tail	22	28	22	30	28	28	30	30	37	37	57	57
11: RecSplice	68	68	68	68	68	68	68	68	68	68	68	68
12: crosstalk	34	34	58	58	34	34	58	58	34	34	58	58
13: crosstalk_if	34	34	58	58	34	34	58	58	34	34	58	58
14: pathjoin	33	33	33	33	41	41	41	41	47	47	47	47

Table 6.1: Measurements: May-Aliases (callstring length 3).

graphs and shape graph sets end up being very large, causing the shape analyses and the alias analysis pass to take longer. The biggest speedup was achieved with nnh-graphset-keep-tail and the RecSplice testcase: in the context-insensitive setting, the NNH analysis did not finish within the 15 minute limit, while in the context-sensitive setting, shape- and alias analysis time was reduced to $5 + 57.7$ seconds.

The common tail extension (`tail`) works best with programs that manipulate heap-allocated data structures that are only indirectly reachable from the stack, i.e. linked data structures where only the entry element is pointed to by a variable, when these data structures do not share heap locations. Two or more linked lists without common elements are the model case for this extension, and it is the independent-lists and semi-independent-lists testcases where `tail` had the strongest impact. Measurements show that the common tail extension only makes sense in combination with the `nnh-graphset` variants. In all other combinations, the common tail extension was not able to improve may-alias results. The cost of this extension, on the other hand, can be neglected, as even in the 57.7 seconds RecSplice testcase the difference between `tail` and `notail` only amounts to fractions of a second that are below measuring precision, so that in some cases the `tail` extension even ran faster than `notail`.

Retaining temporary variables (`keep`) had the biggest impact on accuracy. Only in testcases where the common tail extension is strong did it not improve results. The common tail test works best when many heap locations are coalesced in the summary node. Contrary, `keep` prevents locations from being summarized by retaining temporary variables as additional names. It is therefore clear that these variants are best used mutually exclusive.

The NNH analysis with alias computation performed on individual graphs (nnh-graphset) is in most cases more precise, i.e. it finds smaller sets of may-alias pairs, than performing a single alias test on the combined static shape graph (nnh-nographset), or than using SRW as underlying shape analysis. The increased precision comes with the usual trade-off in analysis runtime.

Precision of the corresponding versions with nnh-nographset and srw-nographset is very similar. Only in two cases, common-tail and pathjoin, is the computationally more expensive NNH analysis despite the lazy alias test still more precise. When the precision obtained by the fast srw-nographset versions is not sufficient, one should consider the nnh-graphset family of analyses and skip the only slightly faster but significantly less precise nnh-nographset variants.

Testcase	nnh-graphset-keep-tail	nnh-graphset-keep-notail	nnh-graphset-nokeep-tail	nnh-graphset-nokeep-notail	nnh-nographset-keep-tail	nnh-nographset-keep-notail	nnh-nographset-nokeep-tail	nnh-nographset-nokeep-notail	srw-nographset-keep-tail	srw-nographset-keep-notail	srw-nographset-nokeep-tail	srw-nographset-nokeep-notail
Analysis Parameters												
1: artificialListCreation	1.7	1.7	6.0	5.9	0.2	0.2	0.9	0.9	0.0	0.0	0.0	0.0
2: artificialSumWithDS	1.0	1.0	3.2	3.2	0.1	0.1	0.3	0.3	0.0	0.0	0.0	0.0
3: artificialSumWithDS_unrolled	0.6	0.6	2.6	2.5	0.1	0.1	0.2	0.2	0.1	0.0	0.0	0.1
4: srw98create	0.1	0.1	0.1	0.1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
5: srw98insert	3.6	3.6	3.2	3.2	0.3	0.3	0.2	0.2	0.1	0.1	0.1	0.1
6: srw98reverse	5.9	5.9	5.7	5.7	0.6	0.6	0.4	0.4	0.1	0.1	0.1	0.1
7: independent-lists	0.6	0.5	0.5	0.5	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
8: semi-independent-lists	1.1	1.0	0.9	0.8	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
9: append-if	0.5	0.5	0.5	0.5	0.0	0.1	0.1	0.1	0.1	0.0	0.0	0.0
10: common-tail	0.9	0.9	0.9	0.9	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
11: RecSplice	57.7	57.6	15.5	15.5	20.7	20.8	2.3	2.4	0.1	0.1	0.1	0.1
12: crosstalk	0.5	0.5	0.5	0.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
13: crosstalk_if	0.5	0.5	0.4	0.4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
14: pathjoin	0.7	0.7	0.7	0.7	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1

Table 6.2: Measurements: Alias Computation Time (callstring length 3).

Chapter 7

Related Work

A vast amount of work has been published on pointer analysis [Hin01]. Our work compares two of the most precise shape analysis algorithms known [SRW98; NNH99], for their relative quality. We do this by comparing alias pairs extracted from the shape graphs and improved the algorithm that is used for the extraction of may-aliases [RSW02] by taking common suffixes of access paths into account. Thus, we give an overview of related work in the fields of pointer alias- and shape analysis.

[EGH94] 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 most locations don't have names and recursive data structures give rise to a theoretically unbound number of locations. It has been shown [Lan92; Ram94] that the may-alias and must-alias problems are undecidable (not recursive) for programs with dynamic storage and recursive data structures. The must-alias problem is not even recursively enumerable (uncomputable) 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.

Flow- and context-sensitivity have the strongest impact on the precision of pointer analyses. Flow-sensitive analyses respect the control-flow of programs and compute separate solutions for every point in the program while flow-insensitive analyses compute a single solution per function, or entire program. Context-sensitivity determines whether an analysis models function

call semantics so that the calling contexts of procedures are independent of each other, or, in a flow-insensitive analysis, each procedure is analysed only once for all its calling contexts combined. Flow- and context-sensitivity are orthogonal properties. The most scalable type of pointer analysis are flow- and context-insensitive analyses. Adding either flow- or context-sensitivity can increase precision, although obviously at additional cost.

7.1 Flow-Insensitive Context-Insensitive

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 [HP01; HP00].

[Ste96] describes a points-to analysis that is almost time-linear but substantially more precise than the address-taken analysis [HP00]. Steensgaard's algorithm uses a type system to describe the store: the type of a variable represents a set of locations and includes a type which in turn represents a set of locations possibly pointed to by the variable. Types thus correspond to nodes in a shape graph, shape graph edges to other nodes correspond to type components. 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.

[And94] extracts subset constraints from a program that reflect 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.

[SH97] compare the address-taken, [Ste96], and [And94] algorithms in combination with subsequent analyses that rely on alias analysis' results. They found that using a more precise pointer analysis (Andersen's) not only leads in general to "transitively" more precise results, i.e. results of the subsequent analysis, but also causes the client analysis to run faster. For small pro-

grams, Andersen's algorithm produced the most accurate results and led to the fastest overall analysis run-times.

Recent work either tries to make unification-based (Steensgaard) analyses more precise or inclusion-based (Andersen) analyses more efficient.

[Das00] studies real-world C programs finding multi-level pointers to be less frequent than single-level pointers: the most frequent use of pointers (in C) is to pass addresses of objects or updateable values as arguments to procedures. Using this observation the precision of a unification-based algorithm is increased to achieve results close to flow-sensitive, context-insensitive algorithms.

[HL07] considerably improve the efficiency of the state of the art in inclusion-based analyses [HT01; PKH04; BLQ+03] 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.

7.2 Flow- or Context-Sensitive

Results in [HP01, HP00] 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 [Hin01] and [FFA00] context-sensitivity brings little or no improvements to Andersen-style analyses, but can be beneficial for simpler equality-based analyses.

More recent research [LH06; LH08] studies the impact of context-sensitivity for Java programs and concludes that it improves precision, but to what extent depends considerably on the client analysis: they list small improvements for call graph construction and major improvements in cast safety analysis. They also note that object-sensitive techniques, techniques that take the receiver object of a method call as relevant context criterion [MRR02; Mil03; MRR05], are the most efficient and precise mechanisms for context-sensitivity in object-oriented languages.

The algorithm described in [LA03, LLA07] combines context-sensitivity by cloning and an explicit heap model with an efficient flow-insensitive unification-based algorithm called data structure analysis (DSA). Heap cloning distinguishes between heap objects that were created at the same allocation site via using the call paths that lead to the allocation site. Each function's heap is represented using a "data structure graph". The main difference between

data structure graphs and the heap representation employed by the shape analyses used in our work is, besides that shape analysis algorithms compute their heap representation for every statement, that the edges in data structure graphs model may-point-to relationships but are only allowed to point to a single target. If the analysis discovers two (or more) nodes that may be pointed to by the same pointer, the nodes are merged. This unification prevents the graph from growing exponentially and is thus essential for its efficiency. Data structure graphs also contain "call nodes" that indicate function calls. This allows the analysis to extract all relevant information from the program in a single pass so that subsequent analysis phases performing the cloning and merging only rely on the data structure graphs. Finally, the shape analyses in our work perform source-level analysis while DSA is performed at link time.

7.3 Flow-Sensitive Context-Sensitive

Analyses in this class are the most precise and computationally expensive. Concentrating on precision, this class also focuses on a more detailed representation of the heap, ultimately leading to what is now called shape analysis.

Work on shape analysis started with [Rey68, JM79]. They statically analysed the structure of the run time heap in order to deduce data set definitions and help with memory allocation, respectively. Both analyses are flow-sensitive but do not treat functions.

[JM79] was 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 [NNH99], but the mechanism to make the shape graphs finite differs: [JM79] limit paths in their shape graphs to a fixed length k (k-limiting), while [NNH99] uses the naming scheme also found in [SRW96, SRW98] 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 one 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 [CWZ90] do not use k-limiting, but summarize shape graph nodes in different summary nodes according to

their allocation site instead. They follow [JM82] in that they also use a single shape graph instead of sets of shape graphs. Their algorithm is able to perform strong updates under certain conditions.

[SRW96, SRW98] was the first shape analysis that was able to always perform strong updates for the elementary transfer functions. There exist, however, uses when strong update cannot be performed, as we have shown in Sect. 6.1.3. [SRW96] deliberately drop the information about the allocation site and represent shape graph nodes only by the set of variables directly pointing to the locations. Being able to always perform strong updates makes the algorithm most accurate, but the 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), which drastically reduces the size of programs for which the algorithm is able to provide results in reasonable time.

[NNH99] based their analysis on [SRW96] but use sets of shape graphs 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.

[Deu94] uses access paths to abstract the structure of the heap, but instead of k-limiting the paths, regular expressions are used to make them finite. It is not clear whether [Deu94] or [SRW96] produce more precise results.

[SRW99, SRW02] presents a parametric framework for shape analyses based on three-valued logic [NNS01]. Instantiations of this framework use three-valued logical structures instead of shape graphs to carry the analysis information. Using this framework, the authors claim, 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."[SRW02, p. 278]

Previous work on shape analysis could be implemented as instantiation of the three-valued logic analysis (TVLA) framework, but some of these algorithms are more efficient than instantiations of the framework would be [SRW02, p. 279].

7.4 Other related work

Recent work [BW00; PSW08] uses shape analysis algorithms for the visualization of programs.

Related to pointer alias analysis is *conflict detection* or dependence analysis. A conflict occurs between two (or more) statements that access the same location with at least one statement modifying it. Aliasing occurs at a single program point while conflicts generally occur between two or more program points. Conflicting statements are an obvious problem in parallelization. The execution order of conflicting statements is essential and must also be preserved by program transformations.

Extending an alias analysis to conflict detection is a question of labelling the locations so that access paths in different statements can be compared.

As both, alias analysis and conflict detection, try to answer the question "which expression also accesses this location?" – alias analysis for different expressions at the same program point, conflict detection for expressions at program points – there is some overlap. [LH88] describe a graph-based alias analysis to address the conflict detection problem for heap objects.

Also related is the work on link-time code analysis and optimization. While most pointer alias analyses, including our work, are formulated in terms of high-level language features, there are also a number of systems that analyse executable code [CGLR97; Fer95; Goo97; RVL+97; SW94; SW93]. [DMW98] present an alias analysis that can be used to obtain aliasing information from executable code.

Chapter 8

Perspectives and Conclusions

The previous chapters discussed the SRW and NNH shape analyses and their application to computing heap-based aliases. This chapter describes the directions for ongoing work to improve the analyses for speed or precision. Finally, a summary of the most important concepts and results concludes this thesis.

8.1 Speed Improvements

Improving on the runtime cost of the alias analyses can address both the cost of the underlying shape analyses, as well as the extraction of aliases:

Reduce Number of Shape Nodes [SRW98], pp. 38–41 suggest the experimentation with different widening policies to reduce the number of shape graphs to trade efficiency for accuracy.

Caching of Alias Tests A large number of tests has to be performed to find the answer to the aliasing question for two expressions and a given shape graph. In our benchmark, the shape graphs contained in NNH shape graph sets were often the same for various points in the program. For shape graphs contained in SRW static shape graphs the recurrences were even more frequent. This suggests that a cache of alias test results for a given expression pair and a shape graph is worthwhile.

If two program points are connected by a statement whose effect on the shape graph is the identity function, there is no need to recompute the alias sets as they, too, will not have changed.

Focus on Relevant Expressions The idea of demand-driven analysis could be applied to focus the alias computation on a smaller set of relevant pointer expressions to improve analysis runtime.

8.2 Precision and Coverage Improvements

The following extensions are suited to further increase the precision of the shape analyses. The language subset (cf. Sect. 5.5) for which the analysis is able to produce relevant results, not only the most conservative ones, could also be extended in various directions:

Explicit nil in SRW If edges in an SRW static shape graph that are deleted on all execution paths were marked as deleted explicitly, e.g. by letting them point to a node called `$nil`, all remaining edges could be depended upon (cf. Sect. 6.1.2), thus increasing the precision of the static shape graph approximation to the point where even must-aliases could be extracted. Selectors that both point to a "real node" and the special nil-node would have the same meaning as the possibly absent edges have in current SRW static shape graphs.

More Summary Nodes The summary node is the main source of imprecision. During materialisation all properties of the summary location are also taken to the newly created location, which introduces imprecision in order to be safe. Especially with a shared summary location the imprecision can be substantial. Separating the summary location into a shared and an unshared summary location would remedy this crosstalk of the sharing. The benefits of keeping separate summary locations for different types need further investigation.

Retaining Temporary Variables Keeping temporary variables to prevent heap nodes from slipping into the summary node is currently an experimental extension to the analysis that already increases the precision. A strategy for deleting temporary variables at the right moment to ensure termination of the analysis needs to be devised.

Arrays Ignoring subscripts of arrays and treating the whole array as a single location on which only weak updates are performed would allow to cover the use of arrays with increased accuracy.

Address-Of Operator Support for the C++ address-of operator (`&`) would greatly increase the coverage of the analyses as the language subset for

which they are applicable would then also include pointer operations on the stack (`y = &x;`), or even the general case of taking addresses of any pointer expression (`y = &(x->next)`).

Aliases on the Stack A points-to analysis ([And94; Ste96]) that provides detailed information about the stack and only limited information about the heap could be combined with a shape analysis to form a hybrid, merging the benefits of both analysis types.

Pointer Arithmetic Programs that contain pointer arithmetic are currently rejected by the analysis. Pointer arithmetic could at least be covered in a conservative way: pointers whose address was computed are may-aliases with every other pointer.

Unions If the aliases introduced by unions (cf. Sect. 5.5) and especially local unions were resolved during assignment of variable ids, the shape analyses could produce correct results in the presence of unions. Different names that refer to the same memory location inside a union need to be assigned the same variable id.

References References introduce additional names for existing objects. Aliases like these need to be considered during assignment of variable ids to extend the shape analyses so that references are supported.

Garbage Collection Shape graph nodes that end up being unreachable could be removed from the shape graph to reduce its size. This can only affect the summary node because all other nodes are by definition pointed to by at least one variable. Eliminating a self-referential summary node that is otherwise unreachable also increases the precision of the shape graph as future nodes being clustered into the summary node are freed of crosstalk with the unreachable structure.

This is guaranteed to be safe as unreachable heap nodes cannot be manipulated by the program and they cannot be re-transformed into manipulatable nodes except through the use of pointer arithmetic. Being unable to manipulate these nodes, the program under analysis cannot contain any statements that introduce aliasing with other pointer expressions.

8.3 Conclusions

This work presented theory and practical implementation of two shape analyses [SRW98; NNH99] and exploited their analysis results to compute alias relations for heap-allocated data structures.

Shape analysis is a powerful program analysis technique that discovers properties of linked data structures in the heap. A wide range of questions about the heap can be answered using shape analysis results. This work concentrated on what was the original motivation for shape analysis: to provide accurate information about aliases in the heap.

A prototype alias analysis based on shape analysis results has been constructed for a subset of C++. The analysis builds on the SATIrE program analysis framework that connects program analysis and transformation tools. The PAG system was used to implement the data flow analyses that compute the heap approximation required for the alias analysis, the ROSE framework provided the facilities to annotate the source code of input programs with alias analysis results.

The SRW and NNH shape analyses were extended to inter-procedural operation in the presence of functions, methods, constructors/destructors and virtual methods. The analyses were adapted to be used for a subset of the C++ programming language. The transfer functions were specifically extended to be applicable to arbitrarily nested pointer expressions.

A novel alias test has been proposed that provides better results when expressions describe paths in dynamic data structures which have no shared elements (cf. Sect. 4.2).

It has been shown that the SRW shape analysis *cannot* be used to compute must-alias sets due to the limitation of the SRW static shape graph representation (cf. Sect. 6.1.2). Contrary to what is often said about [SRW98], the SRW shape analysis does *not always* perform a strong update, which was demonstrated in the evaluation (cf. Sect. 6.1.3).

The prototype alias analysis was tested on C++ programs that employ heap operations. Measurements showed that while the shape graphs are hard to compare directly, the derived may-alias sets can be used to make statements about the precision of the underlying shape analyses. Recommendations for the combination of analysis parameters that affect accuracy and analysis runtime were given. The experimental results made a strong point against context-insensitivity in the presence of constructor calls, as this causes the analysis to run significantly longer while producing worse results.

Bibliography

[ALSU07] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 2nd edition, 2007.
Referenced on page(s) 19.

[And94] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen, May 1994.
Referenced on page(s) 14, 96, 103.

[BLQ+03] Marc Berndl, Ondrej Lhoták, Feng Qian, Laurie Hendren, and Navindra Umanee. Points-to analysis using BDDs. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 103–114, New York, NY, USA, 2003. ACM.
Referenced on page(s) 97.

[BW00] Beatrix Braune and Reinhard Wilhelm. Focusing in algorithm explanation. *IEEE Transactions on Visualization and Computer Graphics*, 6(1):1–7, 2000.
Referenced on page(s) 100.

[CGLR97] Robert Cohn, David Goodwin, P. Geoffrey Lowney, and Norman Rubin. Spike: An optimizer for Alpha/NT executables. In *Proceedings of the USENIX Windows NT Workshop*, pages 17–24, 1997.
Referenced on page(s) 100.

[CWZ90] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *PLDI '90: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 296–310, New York, NY, USA, 1990. ACM.
Referenced on page(s) 98.

[Das00] Manuvir Das. Unification-based pointer analysis with directional assignments. *SIGPLAN Notices*, 35(5):35–46, 2000.
Referenced on page(s) 97.

[Deu94] Alain 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*, pages 230–241, New York, NY, USA, 1994. ACM.
Referenced on page(s) 99.

[DMW98] Saumya Debray, Robert Muth, and Matthew Weippert. Alias analysis of executable code. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 12–24, 1998.
Referenced on page(s) 100.

[EGH94] Maryam Emami, Rakesh Ghiya, and Laurie 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*, pages 242–256, 1994.
Referenced on page(s) 95.

[Fer95] Mary F. Fernandez. Simple and effective link-time optimization of modula-3 programs. In *PLDI '95: Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, pages 103–115, 1995.
Referenced on page(s) 100.

[FFA00] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. Polymorphic versus monomorphic flow-insensitive points-to analysis for C. In *SAS '00: Proceedings of the 7th International Symposium on Static Analysis*, pages 175–198, London, UK, 2000. Springer-Verlag.
Referenced on page(s) 97.

[Goo97] David W. Goodwin. Interprocedural dataflow analysis in an executable optimizer. In *PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, pages 122–133, 1997.
Referenced on page(s) 100.

[Hin01] Michael 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*, pages 54–61, New York, NY, USA, 2001. ACM.
Referenced on page(s) 13, 95, 97.

[HL07] Ben Hardekopf and Calvin Lin. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. *SIGPLAN Notices*, 42(6):290–299, 2007.
Referenced on page(s) 97.

[HP00] Michael Hind and Anthony Pioli. Which pointer analysis should i use? In *ISSTA '00: Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, pages 113–123, New York, NY, USA, 2000. ACM.
Referenced on page(s) 96, 97.

[HP01] Michael Hind and Anthony Pioli. Evaluating the effectiveness of pointer alias analyses. *Science of Computer Programming*, 39(1):31–55, 2001.
Referenced on page(s) 96, 97.

[HT01] Nevin Heintze and Olivier Tardieu. Ultra-fast aliasing analysis using CLA: a million lines of C code in a second. *SIGPLAN Notices*, 36(5):254–263, 2001.
Referenced on page(s) 97.

[Int95] International Organization for Standardization. *ISO/IEC 13211-1:1995: Programming languages – Prolog – Part 1: General core.* International Organization for Standardization, Geneva, Switzerland, 1995.
Referenced on page(s) 68.

[Int03] International Organization for Standardization. *ISO/IEC 14882:2003: Programming languages – C++.* International Organization for Standardization, Geneva, Switzerland, 2003.
Referenced on page(s) 18, 19, 20, 72, 74, 75, 76, 77, 78.

[JM79] Neil D. Jones and Steven 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*, pages 244–256, New York, NY, USA, 1979.

ACM.
Referenced on page(s) 98.

[JM82] Neil D. Jones and Steven 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*, pages 66–74, New York, NY, USA, 1982. ACM.
Referenced on page(s) 99.

[KKS98] Jens Knoop, Dirk Koschützki, and Bernhard Steffen. Basic-block graphs: Living dinosaurs? In *CC '98: Proceedings of the 7th International Conference on Compiler Construction*, pages 65–79, London, UK, 1998. Springer-Verlag.
Referenced on page(s) 18.

[LA03] Chris Lattner and Vikram Adve. Data structure analysis: An efficient context-sensitive heap analysis. Technical report, University of Illinois at Urbana-Champaign, 2003.
Referenced on page(s) 97.

[Lan92] William Landi. Undecidability of static analysis. *Letters on Programming Languages and Systems (LOPLAS)*, 1(4):323–337, 1992.
Referenced on page(s) 13, 95.

[LH88] J. R. Larus and P. N. Hilfinger. Detecting conflicts between structure accesses. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 24–31, New York, NY, USA, 1988. ACM.
Referenced on page(s) 100.

[LH06] Ondrej Lhoták and Laurie Hendren. Context-sensitive points-to analysis: Is it worth it? In *Compiler Construction*, volume 3923 of *Lecture Notes in Computer Science (LNCS)*, pages 47–64. Springer, 2006.
Referenced on page(s) 97.

[LH08] Ondřej Lhoták and Laurie Hendren. Evaluating the benefits of context-sensitive points-to analysis using a BDD-based implementation. *ACM Transactions on Software Engineering and Methodologies (TOSEM)*, 18(1):1–53, 2008.
Referenced on page(s) 97.

[LLA07] Chris Lattner, Andrew Lenharth, and Vikram Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. *SIGPLAN Notices*, 42(6):278–289, 2007.
Referenced on page(s) 97.

[Mar98] Florian Martin. PAG – an efficient program analyzer generator. *International Journal on Software Tools for Technology Transfer*, 2(1):46–67, 1998.
Referenced on page(s) 68.

[Mil03] Ana Lubenova Milanova. *Precise and practical flow analysis of object-oriented software*. PhD thesis, Rutgers University, New Brunswick, NJ, USA, 2003.
Referenced on page(s) 97.

[MRR02] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for java. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 1–11, New York, NY, USA, 2002. ACM.
Referenced on page(s) 97.

[MRR05] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to analysis for java. *ACM Transactions on Software Engineering Methodology (TOSEM)*, 14(1):1–41, 2005.
Referenced on page(s) 97.

[NNH99] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*, chapter Shape Analysis, pages 102–129. Springer, 1999.
Referenced on page(s) 4, 16, 17, 21, 22, 38, 41, 42, 68, 71, 72, 90, 95, 98, 99, 104.

[NNS01] Flemming Nielson, Hanne Riis Nielson, and Mooly Sagiv. Kleene's logic with equality. *Information Processing Letters*, 80(3):131–137, 2001.
Referenced on page(s) 99.

[PKH04] David J. Pearce, Paul H. J. Kelly, and Chris 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*, pages 37–42, New York, NY,

USA, 2004. ACM.
Referenced on page(s) 97.

[Pra07] Adrian Prantl. Source-to-source transformations for WCET analysis: The CoSTA approach. In *24. Workshop der GI-Fachgruppe "Programmiersprachen und Rechenkonzepte"*, pages 51–60. Department of Computer Science, Christian-Albrechts-Universität zu Kiel, 2007.
Referenced on page(s) 68.

[PSW08] Sascha A. Parduhn, Raimund Seidel, and Reinhard Wilhelm. Algorithm visualization using concrete and abstract shape graphs. In *SoftVis '08: Proceedings of the 4th ACM symposium on Software visualization*, pages 33–36, New York, NY, USA, 2008. ACM.
Referenced on page(s) 100.

[Ram94] G. Ramalingam. The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(5):1467–1471, 1994.
Referenced on page(s) 13, 95.

[Rey68] John C. Reynolds. Automatic computation of data set definitions. In *IFIP Congress (1)*, pages 456–461, 1968.
Referenced on page(s) 98.

[RSW02] Thomas W. Reps, Mooly Sagiv, and Reinhard Wilhelm. Shape analysis and applications. In *The Compiler Design Handbook: Optimizations and Machine Code Generation*, pages 175–218. CRC Press, 2002.
Referenced on page(s) 4, 9, 11, 54, 95.

[RVL+97] Ted Romer, Geoff Voelker, Dennis Lee, Alec Wolman, Wayne Wong, Hank Levy, Brian Bershad, and Brad Chen. Instrumentation and optimization of Win32/Intel executables using Etch. In *Proceedings of the USENIX Windows NT Workshop*, pages 1–7, 1997.
Referenced on page(s) 100.

[SBPP09] Markus Schordan, Gergö Bárány, Adrian Prantl, and Viktor Pavlu. *SATIrE Manual*. Part of the SATIrE documentation, June 2009.
Referenced on page(s) 50.

[Sch07] Markus Schordan. Combining tools and languages for static analysis and optimization of high-level abstractions. In *24. Workshop der GI-Fachgruppe "Programmiersprachen und Rechenkonzepte"*, pages 72–81. Department of Computer Science, Christian-Albrechts-Universität zu Kiel, 2007.
Referenced on page(s) 64, 65, 66, 114.

[SH97] Marc Shapiro and Susan Horwitz. The effects of the precision of pointer analysis. In *Proceedings of the 4th International Symposium on Static Analysis*, pages 16–34. Springer-Verlag, 1997.
Referenced on page(s) 96.

[SQ03] Markus Schordan and Dan Quinlan. A source-to-source architecture for user-defined optimizations. In *Joint Modular Languages Conference*, 2003.
Referenced on page(s) 68.

[SRW96] Mooly Sagiv, Thomas W. Reps, and Reinhard 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*, pages 16–31, New York, NY, USA, 1996. ACM.
Referenced on page(s) 16, 98, 99.

[SRW98] Mooly Sagiv, Thomas W. Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 20(1):1–50, January 1998.
Referenced on page(s) 4, 16, 21, 22, 26, 28, 38, 46, 71, 85, 87, 90, 95, 98, 99, 101, 104.

[SRW99] Mooly Sagiv, Thomas W. Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 105–118, New York, NY, USA, 1999. ACM.
Referenced on page(s) 99.

[SRW02] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(3):217–298, May 2002.
Referenced on page(s) 99.

[Ste96] Bjarne Steensgaard. Points-to analysis in almost linear time. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 32–41, New York, NY, USA, 1996. ACM.
Referenced on page(s) 96, 103.

[SW93] Amitabh Srivastava and David W Wall. A practical system for intermodule code optimization at link-time. *Journal of Programming Languages*, 1(1):1–18, March 1993.
Referenced on page(s) 100.

[SW94] Amitabh Srivastava and David W. Wall. Link-time optimization of address calculation on a 64-bit architecture. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 49–60, 1994.
Referenced on page(s) 100.

[YH04] Suan Hsi Yong and Susan Horwitz. Pointer-range analysis. In *SAS*, volume 3148 of *Lecture Notes in Computer Science (LNCS)*, pages 133–148. Springer, 2004.
Referenced on page(s) 74.

List of Tables

6.1 Measurements: May-Aliases (callstring length 3). 92
6.2 Measurements: Alias Computation Time (callstring length 3). 94

List of Figures

1 A short example C++ program with pointers. 3
2 The ReverseList C++ function. 5
3 Concrete heap during execution of ReverseList: before (after line 6, iteration 1), during (after line 5, iteration 3), and after reversal (after line 11). 6
4 Abstract heap represented by SRW static shape graphs as computed by the SRW analysis and illustrated using our visualization for aiSee: before, during, and after reversal. 6
5 Abstract heap represented by NNH sets of compatible shape graphs as computed by the NNH analysis and illustrated using our visualization for aiSee: before, during, and after reversal. . 7
6 Part of function ReverseList. May-alias information is computed by our alias analysis and added to the program's source code as annotations. 8

2.1 A pointer-to-list object and a list object are created by definitions. 19
2.2 A list object is created using a new-expression. 19
2.3 A list as concrete shape graph. 23
2.4 Stores in concrete and abstract semantics. 24
2.5 A list as abstract shape graph. 25
2.6 Taxonomy of Shape Graphs. 28

2.7 Shape graphs used in SRW and NNH. 29
2.8 An SRW static shape graph $SG^{\mathcal{S}}$. 33

3.1 Transfer function wrappers $[\![\cdot]\!]^T$. 48
3.2 C++ code with nested pointer expressions. 48
3.3 ICFG nodes involved in a function call. 51

4.1 A shape graph featuring a common tail. 57
4.2 Cases in naive algorithm when $n_N^v = n_M^w = n_\emptyset$. 59
4.3 Cases in improved algorithm when $n_N^v = n_M^w = n_\emptyset$. 59
4.4 Overview of alias analysis algorithm. 60
4.5 A shape graph with an infinite number of aliases. 62
4.6 ReverseList function with automatically annotated may-alias sets (analysis parameters: srw-graphset-nokeep-tail, cf. Sect. 6.3), edited to fit the page. 63

5.1 Abstract architecture of the SATIrE system. [Sch07, reproduced with permission]. 65
5.2 Architecture of the SATIrE system with integrated modules. [Sch07, reproduced with permission]. 66
5.3 Section of ICFG-matching transfer functions (Layer 1). 70
5.4 Visualization of the reverselist C++ program with shape graphs in aiSee. The annotated ICFG is shown zoomed to program-, function-, and statement-level. 73
5.5 A C++ program with a union. 76
5.6 A C++ program with an anonymous union. 77

6.1 SRW static shape graph with shared named heap node. 81
6.2 SRW static shape graph with shared summary node. 81
6.3 SRW static shape graph at the end of a true-branch. 83
6.4 SRW static shape graph at the end of a false-branch. 83
6.5 SRW static shape graph at program point ℓ, immediately after the branches are joined. 83
6.6 SRW static shape graph with some kind of must-alias information. 85
6.7 SRW static shape graph. 85
6.8 SRW static shape graph after $[\![$`tmp = x->next`$]\!]$. 86
6.9 SRW static shape graph after $[\![$`tmp->next = target`$]\!]$. 86
6.10 SRW static shape graph after $[\![$`tmp = null`$]\!]$. 86

Index

⊗, 30
⊕, three-valued combination operator, 54
k-limiting, 98
& operator, 78
`class`, 75
`enum`, 77
`struct`, 75
`this`, 79
`union`, 75

abstract heap, H, 23
abstract location, 23
abstract state, S, 23
address, 20
address-of operator, 78
address-taken, 96
aiSee, 71
alias
  may, 52
  must, 52
  no, 52
alias analysis, 13
alias extraction algorithm
  improved, 57
  naive, 54
AliasPairsAnnotator, 72
analysis
  alias, 13
  backward, 14
  context-insensitive, 14 95
  context-sensitive, 14 95
  dynamic, 13
  flow-insensitive, 14 95
  flow-sensitive, 14 95
  forward, 14
  inter-procedural, 14
  intra-procedural, 14
  may, 15
  must, 15
  NNH, 16
  pointer, 13
  pointer alias, 13
  points-to, 13
  shape, 13
  SRW, 16
  static, 13
analysis information, 17
analysis parameters, 90
Andersen, 96
anonymous union, 76
applications of shape analysis, 11
array, 19 74
automatic storage duration, 18

backward analysis, 14

Clang, 67
common tail, 56
compatible shape graph, $\mathcal{CSG}$, 22
concrete location, 20
concrete memory representation, 20
constructor, 79
context-insensitive, 14 95
context-sensitive, 14 95
control flow graph (CFG), 17
conversion
  NNH to SRW, 29

SRW to NNH, 29
crosstalk, 81

data structure analysis (DSA), 97
default arguments, 79
dependence analysis, 100
destructor, 79
DFI_WRITE, 72
directed flow graph, 17
dynamic analysis, 13
dynamic memory allocation, 19
dynamic storage duration, 19

EDG C/C++ frontend, 66
enumeration, 77
expressions in $\mathbf{S}_\star$, 17

flow graph, 17
flow-insensitive, 14 95
flow-sensitive, 14 95
forward analysis, 14

GDL, 71

heap, 19
heap edges, 21

inter-procedural, 14
inter-procedural control flow graph, 18
inter-procedural transfer functions, 49
intra-procedural, 14
intra-procedural transfer functions, 38
invariants, 26

library functions, 79

materialisation, 42
may analysis, 15
may-alias, 52
memory abstraction, 23
methods, 79
must analysis, 15
must-alias, 52

nil, 20
NNH, 16
no alias, 52
normalization
nested pointer expressions, 47
special cases (tassign), 46
null check, 11

object, 18
overloaded method, 79
overloaded operator, 79

PAG, 68
parametric analyses, 90
path in a flow graph, 17
pointer, 19
pointer alias analysis, 13
pointer analysis, 13
points-to analysis, 13
predecessor in a flow graph, 17
program (analysis input), 17

reference, 19 78
ReverseList Example, 4
ROSE, 67

SATIrE, 64
shape analysis, 13
shape graph, 21
abstract, 28
abstract, $\mathcal{ASG}$, 24
compatible, $\mathcal{CSG}$, 27
compatible, $\mathcal{CSG}$, 28
concept, $\mathcal{SG}$, 21 28
concrete, 22 28
conversion, 29
deterministic, $\mathcal{DSG}$, 21 22 28
invariants, 26
static, $\mathcal{SSG}$, 22 26 28
taxonomy, 28

shape node labelling, 23
sharing information, is, 25
skip, 18
SRW, 16
stack, 18
stack edges, 21
statement labels in $\mathbf{S}_\star$, 17
static analysis, 13
static memory allocation, 18
static shape graph, 26
static storage duration, 18
Steensgaard, 96
store, 20
strong nullification, 15
strong update, 15
successor in a flow graph, 17
summary location, 23

Termite, 68
three-valued combination operator $\oplus$, 54
transfer function
  $[\![$`x = new T`$]\!]$, 40
  $[\![$`x = null`$]\!]$, 39
  $[\![$`x = y->sel`$]\!]$, 41
  $[\![$`x = y`$]\!]$, 40
  $[\![$`x->sel = null`$]\!]$, 44
  $[\![$`x->sel = y`$]\!]$, 45
  materialisation, 42
transfer function wrappers, 46
transfer functions
  inter-procedural, 49
  intra-procedural, 38
TVLA, 99

union, anonymous, 76

variables in $\mathbf{S}_\star$, 17
virtual functions, 79
visualization, 71

weak update, 15
worst case
  NNH, 27
  SRW, 26