CONTEXT-SENSITIVE POINTER ANALYSIS BASED ON PROCEDURAL SUMMARIES

BY
HONG-SEOK KIM
B.S., Seoul National University, 1998

THESIS
Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2004

Urbana, Illinois
CONTEXT-SENSITIVE POINTER ANALYSIS BASED ON PROCEDURAL SUMMARIES

Hong-Seok Kim, Ph.D.
Department of Computer Science
University of Illinois at Urbana-Champaign, 2004
Wen-mei W. Hwu, Advisor

This dissertation addresses the scalability problem of context-sensitive pointer analysis based on Andersen’s inclusion-based approach [1]. The key observation is that procedures without any observable effects on callers, roughly termed procedural side effects, are never involved in the generation of spurious interprocedural dataflow. Therefore, by carefully migrating procedural side effects from callees to callers, we can transform a program into a side-effect free form yet with the same overall pointer information.

In such a case, the scalability of the overall algorithm is determined by the size of procedural summaries propagated during the bottom-up phase. To reduce the size of procedural summaries, we have developed a number of simplification algorithms that produce an observably equivalent yet smaller procedural summaries.

Overall, the accuracy of the proposed context-sensitive analysis is equivalent to that achieved by exhaustively inlining all procedure calls with the exception for recursive calls. On the other hand, thanks to the simplification algorithms, the scalability of the proposed context-sensitive analysis is comparable to that achieved by context-insensitive analysis.
ACKNOWLEDGMENTS

It must be a bless from God to have all these great people around me throughout my graduate studies. Without their supports, this dissertation would have been impossible. First of all, I wish to thank my advisor, Professor Wen-mei Hwu, for all the things that he taught me both technically and non-technically. I also thank my committee members, Professor Vikram Adve, Professor David Padua, and Professor Grigore Rosu, for their suggestions that greatly improved the quality of this research. I also owe many thanks to all my colleagues in the IMPACT research group who were always willing to provide any help required. I owe particular thanks to Erik Nystrom who developed the magnificent infrastructure that enabled this research and Chien-Wei Li who made my office life as enjoyable as it could be.

I give my earnest thanks to my parents, Kook-Hoo Kim and Hye-Kyung Shin, and little sister, Eun-Jung Kim, for their endless supports. I also thank my daughter Suhyoon Kim for all the delight she has brought into my life. Finally, I would like to express the most sincere gratitude to my lovely wife, Jisun Park, who has been the greatest companion ever since we met. I dedicate this dissertation to her.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Contribution</td>
<td>4</td>
</tr>
<tr>
<td>1.2 Organization</td>
<td>6</td>
</tr>
<tr>
<td>2 BACKGROUND</td>
<td>8</td>
</tr>
<tr>
<td>2.1 Flow Sensitivity</td>
<td>9</td>
</tr>
<tr>
<td>2.2 Flow-Insensitive Approaches</td>
<td>11</td>
</tr>
<tr>
<td>2.3 Context Sensitivity</td>
<td>16</td>
</tr>
<tr>
<td>3 Simple Model Language</td>
<td>24</td>
</tr>
<tr>
<td>3.1 Intraprocedural Setting</td>
<td>24</td>
</tr>
<tr>
<td>3.2 Interprocedural Setting</td>
<td>25</td>
</tr>
<tr>
<td>4 CONTEXT-SENSITIVE POINTER ANALYSIS</td>
<td>29</td>
</tr>
<tr>
<td>4.1 Previous Approaches</td>
<td>29</td>
</tr>
<tr>
<td>4.2 Two-Phase Computation</td>
<td>32</td>
</tr>
<tr>
<td>5 OVERVIEW</td>
<td>38</td>
</tr>
<tr>
<td>5.1 Source of Spurious Dataflow</td>
<td>39</td>
</tr>
<tr>
<td>5.2 Procedures without Side Effects</td>
<td>40</td>
</tr>
<tr>
<td>5.3 Migration of Side Effects</td>
<td>44</td>
</tr>
<tr>
<td>5.4 Handling Escaping Objects</td>
<td>48</td>
</tr>
<tr>
<td>5.5 Simplification Opportunities</td>
<td>51</td>
</tr>
<tr>
<td>5.5.1 Elimination of Local-Only Dataflow</td>
<td>54</td>
</tr>
<tr>
<td>5.5.2 Collapsing Copy Assignments</td>
<td>55</td>
</tr>
<tr>
<td>5.5.3 Redundancy Removal</td>
<td>56</td>
</tr>
<tr>
<td>5.6 Overall Algorithm</td>
<td>57</td>
</tr>
<tr>
<td>6 DISCOVERING CRITICAL ASSIGNMENTS</td>
<td>59</td>
</tr>
<tr>
<td>6.1 Symbolic Calling Contexts</td>
<td>59</td>
</tr>
<tr>
<td>6.1.1 Holding Property</td>
<td>60</td>
</tr>
<tr>
<td>6.1.2 Escaping Property</td>
<td>62</td>
</tr>
<tr>
<td>6.2 Details of Criticality Detection Algorithm</td>
<td>66</td>
</tr>
</tbody>
</table>
7 BACK-TRACING ALGORITHM .............................................. 69
  7.1 Coping with Aliases ................................................. 71
  7.2 Handling Escaping Objects ........................................... 74
  7.3 Details of Back-Tracing Algorithm ................................ 75
    7.3.1 Initiation ...................................................... 75
    7.3.2 Address Assignments ......................................... 76
    7.3.3 Copy Assignments ............................................. 77
    7.3.4 Load Assignments ............................................. 78
    7.3.5 Store assignments ............................................ 80
8 REDUNDANCY REMOVAL ALGORITHM .................................... 84
  8.1 Automata-Theoretic Approach ....................................... 85
  8.2 Handling Escaping Variables ....................................... 88
  8.3 Partitioning Algorithm ............................................ 92
    8.3.1 Hopcroft’s Partitioning Algorithm for DFA ................... 92
    8.3.2 AWZ’s Partitioning Algorithm for SSA Graphs ............... 93
    8.3.3 Partitioning Algorithm for Procedural Summaries .......... 94
9 EXPERIMENTAL RESULTS .................................................. 101
  9.1 Efficiency .......................................................... 101
    9.1.1 Field-Insensitive Setting .................................... 101
    9.1.2 Field-Sensitive Setting ..................................... 104
  9.2 Accuracy ............................................................ 108
    9.2.1 Points-to Set Size ........................................... 109
    9.2.2 Accesses per Object ......................................... 111
10 CONCLUSION ............................................................ 115
REFERENCES ............................................................... 117
AUTHOR’S BIOGRAPHY ..................................................... 121
CHAPTER 1

INTRODUCTION

In programming tools developed for languages with pointers, such as C, C++, and Java, accurate information about pointer usage is in high demand. Without such information, conservative assumption regarding pointer access must be made, which can greatly affect the effectiveness of such tools. Pointer analysis is an algorithm that statically determines all possible pointer usage during program execution. Since such analysis is undecidable in general [2, 3, 4], termination is guaranteed by computing approximated information instead.

A large collection of approximation approaches have been proposed with a significant variation in accuracy and scalability. It is generally accepted that flow sensitivity has relatively low impact on the accuracy of pointer analysis in spite of its high overhead [5, 6]. For this reason, most recent pointer analyses use flow-insensitive approaches.

Many different forms of flow-insensitive pointer analysis are available in the literature [1, 7, 8, 9]. It is generally accepted that unification-based pointer analysis [9] could significantly degrade accuracy [5, 6, 10, 11, 12]. Some of the lost accuracy can be recovered at the cost of adding a relatively small overhead to the unification-based pointer analysis [7, 8]. However, at the same time there have been significant improvements in the scalability of inclusion-based approaches [13, 14, 15, 16].
Context sensitivity is becoming increasingly more important because modern programming practices exasperate the problem of spurious dataflow across procedural boundaries. In many cases, context-sensitive analysis can achieve a higher degree of accuracy than its context-insensitive counterpart by avoiding unrealizable interprocedural dataflow. In pointer analysis, the impact of context sensitivity is increased further when analyzing programs that use an extensive amount of heap-allocated data structures.

There have been many approaches to context-sensitive pointer analysis in the literature. Early analyses \cite{17, 18, 19, 20, 21} mimic dynamic execution, repeating call-return sequences until the analysis reaches a global fixed point. In these analyses, a procedure must be re-analyzed whenever there is a change in its callers or callees, leading to a serious scalability problem.

On the other hand, much recent work \cite{22, 23, 11, 24, 25} formulates pointer analysis as a two-phase computation. In the first phase of the two-phase computation, called the bottom-up phase, a flow function is constructed for each procedure by propagating procedure summaries from callees to callers. Then, the second phase, called the top-down phase, computes the actual pointer information by propagating calling contexts from callers to callees.

Effectively, context sensitivity exists in two different venues.

- During the bottom-up phase, it is important to separate the effects of distinct procedure calls to the same procedure. The separation of the effects is usually realized by providing distinct copies of procedural summaries for distinct procedure
calls. In such a case, to maintain scalability, it is important to limit the growth of
the summary as the analysis progresses upwards in the call graph.

- During the top-down phase, it is important to separate the effects of the calling
contexts from different call sites. The separation of the effects is usually realized
by re-analyzing callees with different calling contexts. In such a case, the size of
calling contexts and the number of different calling contexts encountered during
the top-down phase greatly affects scalability.

In unification-based context-sensitive pointer analysis, in which objects pointed to by
the same object are merged for the efficiency reason, the size of procedural summaries and
calling contexts is kept under control by unification itself. Therefore, scalability rarely
becomes an issue and no additional techniques to enhance scalability are needed [25, 26,
24]. Moreover, since unification leaves many parts of calling contexts indistinguishable,
one can also expect that the number of different calling contexts encountered during the
top-down phase could be small enough so that procedures can be re-analyzed for each
different calling context without causing serious performance degradation. For instance,
the top-down phase in the unification-based context-sensitive escape analysis proposed
by Ruf [27] is capable of re-analyzing each procedure under different calling contexts
without encountering serious performance degradation.

The situation is dramatically different when context-sensitive pointer analysis is based
on Andersen’s inclusion-based approach. The dataflow modeling of the inclusion-based
approach does not provide a way to reduce the information size. Therefore, without ex-
explicit management with simplification or approximation, the size of procedural summaries and calling contexts grows explosively.

1.1 Contribution

In this dissertation, we address the scalability problem of context-sensitive pointer analysis based on Andersen's inclusion-based approach [1]. The key observation is that procedures without any observable effects on callers, roughly termed procedural side effects, are never involved in the generation of spurious interprocedural dataflow. Therefore, by carefully migrating procedural side effects from callees to callers, we can transform a program into a form that lacks procedural side effects yet maintaining the same overall pointer information.

In such a case, the scalability of the overall algorithm is determined by the size of procedural summaries propagated during the bottom-up phase. To reduce the size of procedural summaries, we develop a number of simplification algorithms that produce an observably equivalent yet smaller procedural summaries.

Overall, the accuracy of the proposed context-sensitive analysis is equivalent to that achieved by exhaustively inlining all procedure calls with the exception for recursive calls. On the other hand, thanks to the simplification algorithms, the scalability of the proposed context-sensitive analysis is comparable to that achieved by context-insensitive analysis.
**Scalable Bottom-Up Context Sensitivity.** To reduce the size of procedural summaries, we develop a number of simplification algorithms. In particular, we exploit the following simplification opportunities.

- Much dataflow made in callees has no effect on the dataflow made in callers. Therefore, we can eliminate such dataflow from procedural summaries.

- Even within the dataflow visible to callers, there exists much indirection through a sequence of copy assignments. By collapsing such copy assignments, the size of procedural summaries can be further reduced.

- In many cases, the effects of two calls to the same procedure are indistinguishable from the perspective of the flow modeling based on Andersen’s approach. In such cases, by eliminating redundant dataflow made through different calls, the size of procedural summaries can be further reduced.

**Scalable Top-Down Context Sensitivity.** By migrating procedural side effects from callees to callers, procedural side effects of callees are now generated by the local copies of procedural summaries in callers. From the perspective of callees, the local computation that depends on the original procedural side effects can now receive their impact from calling contexts. Therefore, the procedural side effects from callees are no longer necessary.

On the other hand, the dataflow generating procedural side effects is also the dataflow generating spurious interprocedural dataflow. Therefore, such dataflow is problematic
and becomes redundant after the bottom-up phase. This observation leads us to the reformulation of two-phase computation as follows.

- In addition to the propagation of procedural summaries from callees to callers, the bottom-up phase transforms callees into a form that lacks procedural side effects. Effectively, procedural side effects are migrated from callees to callers.

- After the bottom-up phase, the overall pointer information remain unchanged, while context sensitivity becomes inherent in the program. Therefore, a single run of context-insensitive analysis is sufficient as the top-down phase that generates accurate pointer information without experiencing any interprocedural dataflow.

Thanks to the simplification algorithms used in the bottom-up phase, the size of the program after the bottom-up phase is comparable (in most cases within 10% increase in the number of variables) to that of the original program. Therefore, the scalability of the top-down phase is no longer of a concern.

1.2 Organization

The remainder of this dissertation is organized as follows. Chapter 2, explores various approaches to pointer analysis and discuss their effectiveness in terms of accuracy and scalability.

Chapter 3 presents a simple model language that captures the essence of Andersen’s inclusion-based dataflow modeling with a small number of derivation rules and procedure-call semantics with inlining and variable renaming.
Chapter 4 presents the previous approaches to context-sensitive pointer analysis. We introduce two-phase computation and discuss the limitations of conventional approaches with an observation that sheds light on how to overcome such limitations.

Chapter 5 generalizes the observation made in Chapter 4 into the reformulation of two-phase computation and presents a brief overview of the proposed techniques along with the overall structure.

Chapters 6, 7, and 8 present the details of the proposed techniques. Chapter 6 presents an algorithm that discovers how procedural side effects are generated within procedural summaries. Chapter 7 presents an algorithm that performs garbage collection and copy collapsing simultaneously. Chapter 8 presents an algorithm to identify redundant variables.

Chapter 9 presents the experimental results that demonstrate the effectiveness of the proposed techniques in terms of accuracy and scalability. Chapter 10 concludes the dissertation.
CHAPTER 2
BACKGROUND

In this Chapter, we explore various approaches to pointer analysis and discuss their effectiveness in terms of accuracy and scalability. Among many factors affecting accuracy and scalability of pointer analysis, we focus on flow and context sensitivity.

- We assume that when designing a completely automatic whole-program pointer analysis, flow-insensitive approaches are more attractive than its flow-sensitive counterparts. This argument is supported by the experimental results of previous work [5, 6, 10].

- In literature, various flow-insensitive approaches have been proposed including Andersen’s inclusion-based approach [1], Steensgaard’s unification-based approach [9], and many other approaches that lie between them [8, 7]. We assume that the blind application of unification could lead to a considerable amount of accuracy degradation especially when programs are analyzed context-insensitively. This assumption is supported by the experimental results of previous work [11, 12] and by our own experience.

- There have been much debate how much impact context sensitivity has on accuracy. So far, the experimental results have been mixed. While context sensitivity is
proven to significantly improve the quality of unification-based pointer analysis [25, 11, 26], it did not lead to a noticeable improvement for flow-sensitive or inclusion-based analyses [20, 11, 28].

- Despite these negative previous results, we show that context sensitivity is critical even for inclusion-based analysis especially for the large programs using an extensive amount of heap-allocated data structures. This argument is supported by the experimental results in Chapter 9.

## 2.1 Flow Sensitivity

Flow-sensitive analysis attempts to faithfully reflect the effects of executing statements in the order specified by control-flow graphs. The benefit is that, when there is more than one write to the same memory location, flow-sensitive analysis can potentially determine which write is the outstanding one at the specific control-flow point. This is typically referred to as *strong updates*. The cost is that flow-sensitive analysis needs to maintain separate dataflow information for each control-flow point and explicitly propagate it through control-flow graphs.

In interprocedural pointer analysis, the cost involved in flow sensitivity can easily overwhelm the benefit of flow sensitivity. Empirical studies suggest that at least for context-insensitive analysis, flow sensitivity does not make much difference in terms of the quality of the output pointer information [5, 6, 10].
Interprocedural dataflow typically involves pointer manipulation across procedural boundaries. Therefore, in interprocedural pointer analysis, the points-to graphs within one procedure tend to contain variables from many other procedures. Considering that the size of points-to graphs grows quadratic to the number of variables, maintaining distinct points-to graphs for each control-flow point interprocedurally has an apparent scalability problem.

On the other hand, even flow-sensitive analysis fails to perform strong updates in many cases. Consider a pointer analysis which maintains the dataflow information in the form of points-to graphs. In such a case, to guarantee termination, heap objects are typically abstrated into a finite number of abstract heap objects (for instance one abstract heap object per allocation site). In such a case, the strong updates on abstract heap objects can be unsafe since it remains unsure which concrete heap object is updated by the write. For this reason, many flow-sensitive analyses, such as [17, 21, 22], never perform strong updates for heap objects. Whenever there is a write into an heap object, its effect is accumulated rather than overrides the previous writes into the object. This is typically referred to as weak updates. Considering that the majority of pointer access is on heap objects, it is less likely that flow sensitivity in those analyses turns into a significant improvement on accuracy.

For this reason, most recent work on efficient pointer analysis employs a flow-insensitive approach. By being insensitive to control-flow structures, it effectively ignores the execution order imposed by control-flow graphs. In the following section, we discuss various approaches to flow-insensitive pointer analysis.
1:     struct Table { Node* node; }
2:     struct Node { Info* info; }
3:     main {
4:         t1 := new Table();
5:         t2 := new Table();
6:     }
7:     n1 := new Node();
8:     n2 := new Node();
9:     t1->node := n1;
10:    t2->node := n2;
11:    if(c) {
12:        t := t1;
13:    } else {
14:        t := t2;
15:    }
16:    n := t1->node;
17: }

**Figure 2.1** Example illustrating the difference between Steensgaard’s, Das’, and Andersen’s approaches to flow-insensitive pointer analysis.

### 2.2 Flow-Insensitive Approaches

Many different forms of flow-insensitive pointer analysis have been proposed in literature. Steensgaard’s unification-based approach [9] scales well to large programs (it has almost linear-time complexity) but produces relatively imprecise results. On the other hand, Andersen’s subtyping-based approach [1] produces much better results, but can be substantially slower than Steensgaard’s unification-based approach (it has cubic worst-
Figure 2.2 Application of various flow(insensitive algorithms on the example in Figure 2.1. (a) points-to graph from Andersen’s algorithm [1] (b) points-to graph from Steensgaard’s algorithm [9] (c) points-to graph from Das’ algorithm [7].

For the comparison between the context-insensitive implementations of Steensgaard’s and Andersen’s approaches, we refer readers to [12, 11].

There are also algorithms that lie between Steensgaard’s and Andersen’s algorithms. In an attempt to balance accuracy and scalability, Shapiro and Horwitz proposed a randomized multiple-outdegree algorithm that bridges the gap between Steensgaard’s
and Andersen's algorithm by $k$-limiting the outdegree of the nodes in points-to graphs [8]. Das also proposed a one-level flow algorithm that uses a restricted form of subtyping to avoid unification of variables at the top level of pointer chains in points-to graphs while using unification elsewhere [7].

The typical usage of local variables as a means to scan through data structures greatly affect the accuracy of Steensgaard's and Das' algorithm. In typical programs, while heap objects form the back bone of the data structures manipulated during the program execution, local variables are used for traversing various data structures formed with heap objects. In such cases, the unification performed by Steensgaard’s and Das' algorithms would merge distinct data structures into a single data structure as local variables traverse those data structures. Of course, the situation is better for Das’ algorithm since unification is avoided for top-level pointer variables. However, many data structures have multiple levels and even Das’ algorithm is unable to avoid the unification at the levels other than the top level. We will elaborate on this issue with an example shortly. Based on these observations, we choose Andersen’s subtyping-based approach as a basis of our context-sensitive algorithm.

There has been noticeable improvement in the efficiency of the context-insensitive Andersen’s algorithm. Many simplification algorithms have been proposed to speed up the solution of subtyping constraints [13, 14, 15]. The context-insensitive algorithm proposed in [16] is reported to analyze million lines of code within seconds. However, the scalability of the context-sensitive Andersen's algorithm still remains as a challenge.
In the following, using the simple example in Figure 2.1, we give a comparison of different flow-insensitive algorithms. The example in Figure 2.1 demonstrates a typical pointer usage in C, C++, and Java programs except that all dataflow is made within a single procedure instead of occurring across procedural boundaries.

In this example, the procedure main() allocates two disjoint tables each assigned into t1 and t2. The variable t is used to scan the data structures formed by Table and Node objects. To analyze this example accurately, it is important to recognize that the program maintains two disjoint tables.

In the points-to graphs shown in Figure 2.2, nodes represent memory locations such as variables and heap objects while directed edges represent points-to relation between memory locations. For instance, in the points-to graph 2.2(a), the edge from the node t1 to the node T1 is a consequence of the allocation assignment t1 := new Table() at the line 5 in Figure 2.1. To represent heap objects, we introduce pseudo variables such as T1 and N1.

Figure 2.2(a) shows the final points-to graph obtained from Andersen’s algorithm. In order to accommodate directionality, it allows unlimited outdegree in points-to graphs. Therefore, the fact that the variable t points to both T1 and T2 does not force n to point to N2.

Figure 2.2(b) shows the final points-to graph obtained from Steensgaard’s algorithm. The key difference between the points-to graphs (a) and (b) is that each node in the points-to graph (b) has at most one outbound edge. It is because Steensgaard’s algorithm merges the nodes pointed to by the same node into a single node. This merging turns
main() {
    c := foo₁(&a);
    d := foo₂(&b);
}

foo(q) {
    return q;
}

**Figure 2.3** Code fragment illustrating how a spurious dataflow is formed through a call-return mismatch.

into a great advantage in bounding analysis time. The points-to graph is always linear to the number of variables and no fixed-point computation is required. Unfortunately, it comes at the cost of accuracy. In particular, the unification leads to the spurious pointer information for n. (The variable n points to both N1 and N2.)

Figure 2.2(c) shows the final points-to graph obtained from the one-level flow algorithm by Das [7]. In the one-level flow algorithm, subtyping is used for top-level pointers while unification is used elsewhere. For instance, in Figure 2.2(c), the variable t is a top-level pointer, since it is not pointed to by any variable. Since subtyping is applied for the node t (an arbitrary degree of outbound edges is allowed), nodes T₁ and T₂ are left distinct. On the other hand, since unification is applied for the node T₁ and T₂ (they need to points to the same set of nodes), nodes N₁ and N₂ are merged into a single node. Consequently, the node n is forced to point to both N₁ and N₂ due to unification.
2.3 Context Sensitivity

The other factor that greatly affects the accuracy and scalability of pointer analysis is context sensitivity. In flow-sensitive analysis, context sensitivity corresponds to correlation between call and return branches [29]. In particular, a procedure is supposed to return to the call site that made the last call to the procedure. Consider the example in Figure 2.3. When analyzing this code fragment context-insensitively, the address &a could be assigned into the variable c following a valid call-return path within call site foo_1() as depicted in the following derivation trees called traces. (The details of trace derivation will be discussed in Chapter 3.)

\[
\begin{array}{c}
[c := q]_1 [q := &a]_1 \\
c := &a
\end{array}
\]

On the other hand, following unrealizable call-return paths [19, 30] between call sites foo_1() and foo_2(), the address &a could be also assigned into the other variable d as follows.

\[
\begin{array}{c}
d := q]_2 [q := &a]_1 \\
d := &a
\end{array}
\]

In the flow-insensitive setting, it is also possible that a spurious dataflow is formed due to the interaction between two parameter assignments associated with different call sites. Consider the example in Figure 2.4. When analyzing this code fragment context-
main() {
    foo1(&e, &a);
    foo1(&f, &b);
}
foo(s, t) {
    u := *t;
    *s := t;
}

**Figure 2.4** Example illustrating the formation of spurious dataflow due to the interaction between two parameter assignments generated from different call sites.

insensitively, the following trace produces the spurious result e := &t.

\[
\begin{array}{c}
*s := t \quad [s := &e]_1 \\
 e := t \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad [t := &b]_2 \\
 e := &b
\end{array}
\]

The problem is as follows. From the call site `foo1()`, the address `&e` is passed into the procedure `foo()` through the parameter `s`. Therefore, within the call site `foo1()`, any address stored in the variable `t` would be assigned into the variable `e` at the store assignment `*s := t`. On the other hand, from the call site `foo2()`, the address `&b` is passed into the procedure `foo()` through the parameter `t`. However, since the address assignment `t := &b` is valid only within the call site `foo2()`, it should not interact with the dataflow `e := t`, which is valid only within the other call site `foo1()`.

It is also possible that two return-variable assignments associated with different call sites interact producing a spurious dataflow. Consider the example in Figure 2.5. When analyzing this code fragment context-insensitively, the following trace leads to the spu-
```c
main() {
    a := foo1();
    b := foo2();
    c := &x;
    *a := c;
    d := *b;
}
foo() {
    // Assume that it returns the address of object y.
    return new Object();
}
```

**Figure 2.5** Example illustrating the formation of spurious dataflow due to the interaction between two return-variable assignments associated with different call sites.

The problem is as follows. From the call site `foo2()`, the address of the heap object `y` is returned. By dereferencing this address, the store assignment `*a := c` assigns the address `&x` into the heap object `y`. On the other hand, from the call site `foo1()`, the address of the other instance of the heap object `z` is returned. By dereferencing this address, the load assignment `d := *b` forms the dataflow `d := y`. However, the dataflow `d := y` is not supposed to interact with the address assignment `y := &x` since the instance of `y` in `d := y` and the instance of `y` in `y := &x` does not match.
As demonstrated, in the flow-insensitive setting, it is no longer natural to regard context sensitivity as keeping correlation between call and return branches. The following is a more appropriate interpretation. In the actual execution of programs, a variable corresponds to a collection of instances, one for each call chain that leads to the home procedure of the variable. In such a case, a write into one instance must not affect a read out of the other instance. From this perspective, in context-insensitive analysis, spurious dataflow is triggered when a write into a variable in one context spuriously interacts with a read out of the variable in another context.

The importance of context sensitivity becomes more apparent when analyzing programs extensively relying on dynamically allocated data structures. In such cases, it is important that pointer analysis distinguishes the instances of heap objects created under different call chains. The example in Figure 2.5 already demonstrates a case in which the spurious dataflow is generated through the under-cloned heap object \( y \).

Figure 2.6 shows a more involved example capturing a typical usage of heap-allocated tables. To determine that variable \( \text{info} \) at Line 12 only points to the \( x \) object allocated at Line 5, pointer analysis must discover the following facts.

**Fact (1).** Line 41 is the only allocation site for \( \text{Table} \) objects. However, since each execution of the allocation site returns a fresh heap object, \( \text{table1} \) and \( \text{table2} \) point to distinct \( \text{Table} \) objects.

**Fact (2).** Line 36 is the only allocation site for \( \text{Node} \) objects. However, since each execution of the allocation site returns a fresh heap objects, the \( \text{Table} \) objects stored in variables \( \text{table1} \) and \( \text{table2} \) never point to the same \( \text{Node} \) object.
1:   main() {
2:    table1 := create1();
3:    table2 := create2();
4:  
5:    info1 := new X();
6:    info2 := new Y();
7:  
8:    add_node3(table1, info1);
9:    add_node4(table2, info2);
10:   
11:    node = get_node(table1);
12:    info = node->info;
13:   
14:    if(c) {
15:       main_table = table1;
16:    }
17:    else {
18:       main_table = table2;
19:    }
20:   }
21:  
22:   add_node(table, info) {
23:      if(c) {
24:         insert5(table, info);
25:      }
26:      else {
27:         insert6(table, info);
28:      }
29:   }
30: 
31:   get_node(table) {
32:      return table->node;
33:   }
34: 
35:   insert(table, info) {
36:      node := new Node(info);
37:      table->node = node;
38:   }
39: 
40:   create() {
41:      table := new Table();
42:      return table;
43:   }

Figure 2.6  Example demonstrating importance of heap-object specialization.
Figure 2.7(a) shows how context-insensitive analysis without heap object specialization captures the usage of heap objects within the example in Figure 2.6. In context-insensitive analysis, each allocation site is given a unique heap object. Effectively, all heap objects allocated at the same site are treated equally. In Figure 2.7(a), the node \( T \) corresponds to the allocation site at the line 41 and the node \( N \) corresponds to the allocation site at the line 36. Therefore, context-insensitive analysis fails to recognize both facts, thus, leads to the spurious conclusion that \( \text{info} \) points to both \( X \) and \( Y \).

One can augment context-insensitive analysis by detecting the pattern of \textit{object factories}, returning fresh heap objects, and specialize such procedures along different call chains. For instance, procedure \( \text{create()} \) has the pattern of object factories. Therefore, by specializing the object created in \( \text{create()} \) for two calls at Lines 2 and 3, even context-insensitive analysis can discover that the variables \( \text{table1} \) and \( \text{table2} \) point to different \texttt{Table} objects.

Figure 2.7(b) shows the effect of treating object factories in a special way. In this points-to graph, the nodes \( T_1 \) and \( T_2 \) correspond to those cloned heap objects. However, since procedure \( \text{insert()} \) does not return fresh heap objects, it would not be detected as an object factory. Therefore, even after specializing object factories, context-insensitive analysis would have only a single heap object \( N \) associated with the allocation site at Line 36. In such a case, both objects \( T_1 \) and \( T_2 \) end up pointing to the single object \( N \). In turn, it forces the object \( N \) to point to both \( X \) and \( Y \) objects in procedure \( \text{insert()} \). Effectively, \( \text{info} \) ends up pointing to both \( X \) and \( Y \) objects, too.

21
Figure 2.7 Effect of different strategies in specializing heap objects. (a) No heap-object specialization (b) specialization for object factories (c) exhaustive specialization along call chains (d) after redundant nodes in (c) are merged.

As demonstrated, an ad-hoc approach such as detecting object factories may end up increasing the analysis overhead without bringing any significant improvement in accuracy. To accurately analyze programs such as the one in Figure 2.6, pointer analysis must be truly context-sensitive in its nature. In many context-sensitive pointer analyses [23, 18, 24, 26, 27], heap-object specialization is performed along call chains. Even though
imperfect, heap-object specialization is useful in eliminating sources of spurious dataflow in many cases.

Figure 2.7(c) shows the points-to graph when heap-object specialization is performed along call chains in the example in Figure 2.6. In this points-to graph, the node T1 corresponds to the heap object allocated at the line 41 cloned along the call chain (1) and the node Τ35 to the heap object allocated at the line 36 cloned along the call chain (3·5). In this points-to graph, it is clear that tables T1 and T2 are disjoint. Therefore, it recognizes that the variable info only points to the x object.

While this approach provides a higher degree of accuracy, it has a clear implication on scalability since too many heap objects are created. In reality, exhaustive specialization along all distinct call chains creates a significant amount of redundant heap objects. Consider procedure calls insert5() and insert6() in the procedure addnode(). Except that they are made under different control-flow points, from the perspective of flow-insensitive and context-sensitive pointer analysis, the effects of these two calls are identical. Therefore, the differentiation of the heap objects created within the procedure calls insert5() and insert6() is unnecessary. Effectively, nodes Τ35 and 36 (and nodes Τ45 and 46) in Figure 2.7(c) behave identically. Figure 2.7(d) shows the points-to graph after the redundant nodes in the points-to graph in Figure 2.7(c) are removed.
CHAPTER 3

Simple Model Language

Based on this discussion made in Chapter 2, we decide to develop a context-sensitive pointer analysis based on Andersen’s inclusion-based approach. For accurate treatment of heap-allocated data structures, heap objects are cloned along call chains leading to their allocation sites.

To allow rigorous discussion without being overly complicated by less irrelevant details of realistic programming languages, we present a simple model language in Chapter 3. The simple model language captures the essence of Andersen’s inclusion-based dataflow modeling with a small number of derivation rules and procedure-call semantics with inlining and variable renaming.

3.1 Intraprocedural Setting

In a flow-insensitive setting, statements are assumed to be executed in an arbitrary order. Therefore, it is convenient to regard a procedure body simply as a set of statements. Since the main focus of this dissertation is on context sensitivity, for further simplification, we ignore all the offsets, such as array indices and field accesses. Note that, however, the actual implementation is sensitive to the fields within variables while array
indices are still ignored. In the model language, the following four kinds of statements are considered.

\[ u := \&v \mid u := v \mid *u := v \mid u := *v \]

The intuitive meaning of the address operator (denoted \&) and dereference operator (denoted *) follow those from the C language.

In a path-insensitive setting, it is convenient to view execution traces in a tree form. The rules in Figure 3.1 specify small-step derivation rules in our abstract setting following [16]. In these rules, \( T, T_1, \) and \( T_2 \) are metavariables ranging over traces. The root of trace \( T \) is called the conclusion of \( T \). When we want to emphasize that \( e_x := e_y \) is a conclusion of trace \( T \), we write \( T[e_x := e_y] \). From those rules, we can define the meaning of a set of statements \( A \) as follows.

\[ \llbracket A \rrbracket = \{ u := \&v \mid u := \&v \text{ derivable from } A \} \]

### 3.2 Interprocedural Setting

We extend the assignments with procedures. For simplicity, we ignore return statements in formal discussion, since the effect of return statements can be captured using extra parameters and store assignments. Then, a procedure declaration has the following form

\[ p(x, x', \cdots) \{ S, S', \cdots \} \]
\[
\frac{e := e'}{e := e' \in A} \quad (3.1)
\]

\[
\frac{x := y \quad y := \&z}{x := z} \quad (3.2)
\]

\[
\frac{x := \&y \quad y := \&z}{x := z} \quad (3.3)
\]

\[
\frac{\&x := z \quad x := \&y}{y := z} \quad (3.4)
\]

**Figure 3.1** Derivation rules for Andersen’s pointer analysis.

where \( p \) is the *procedure name*, \( x \) and \( x' \) are *formal parameters*, and \( \{S, S', \cdots\} \) is the body of the declared procedure. Statements \( S \) and \( S' \) include assignments and *procedure calls* in the following form.

\[
q_k(e, e', \cdots)
\]

where \( q \) is the named of the *called procedure*, the label \( i \) is the *call site identifier* given uniquely to each procedure call, and the *actual parameters* (denoted \( e, e' \)) are expressions

\[
x \mid \&x \mid \&x.
\]

A *program* is simply a set of procedure declarations. By default, we assume that it includes a declaration of procedure *main*, which does not have any caller. We also assume that all programs are well-formed in an intuitive sense. For instance, each called
procedure is declared in the program, number of actual and formal parameters match, etc. All the variables appearing in the definition of a procedure declaration, including formal parameters, are considered local to the procedures. At this moment, we ignore global variables.

The specialization of a call site consists of the specialized version of the body of its called procedure and a set of assignments capturing the effects of parameter passing at the call site. A small-step inlining is a triple \((A, c, A')\) where \(A\) is a set of assignments and \(c\) is a call site in \(A\). Then, the statement set \(A'\) is obtained by replacing \(c\) in \(A\) with its specialization. A big-step inlining is a sequence of small-step inlinings

\[(A_1, c_1, A_2), (A_2, c_2, A_3), \ldots, (A_{n-1}, c_{n-1}, A_n).\]

In such a case, we say that the big-step inlining begins with \(A_1\) and ends with \(A_n\). Let \(\Psi\) be a program. Since recursion is handled in a context-insensitive fashion, we can assume that the call graph of the program \(\Psi\) is acyclic. Therefore, it is clear that there exists a big-step inlining that begins with the body of main and ends with some \(A_n\) without any call site. In such a case, the corresponding ending statements are called the exhaustive expansion of the program \(\Psi\).

By intraprocedurally analyzing the exhaustive expansion of the program, we can compute all the pointer information completely and accurately. We assign a meaning to
main() {
    foo₁(&a, 1);
    foo₂(&b, 3);
}

foo(p,q) {
    bar₃(&r,q);
    bar₄(&s,5);
    *p := r;
}

bar(x,y) {
    z := y; *x := z;
}

(a)  

(b)

Figure 3.2 Example of the program written in the simple model language. (a) a program and (b) exhaustive expansion of the program.

the program \( \mathcal{P} \) as follows.

\[
[\mathcal{P}] = \{ u := \&v \mid \exists \alpha, \beta \text{ s.t. } u_\alpha := \&v_\beta \in [A_\beta]\}
\]

For instance, consider the small program in Figure 3.2(a). The set of assignments in Figure 3.2(b) corresponds to the exhaustive expansion of the program. From these assignments, we can derive the following four results: \( x_{13} := 1, x_{14} := 5, x_{23} := 3, \) and \( x_{24} := 5 \), each corresponding to four possible call chains from \textbf{main()} to \textbf{bar()}. Without call chain information, these address assignments correspond to the following assignments: \( x := 1, x := 3, \) and \( x := 5. \)
CHAPTER 4

CONTEXT-SENSITIVE POINTER ANALYSIS

This chapter presents the previous approaches to context-sensitive pointer analysis, discusses the limitation of the previous approaches, and an observation that sheds light on how to overcome such limitations.

4.1 Previous Approaches

Context-sensitive pointer analyses have been approached from several directions. In [17, 18, 19, 20, 21], procedures are re-analyzed until a global fixed point is reached. Among these, [21] is unique in the sense that the overhead of re-analysis is reduced by memoization. On the other hand, the analyses in [22, 23, 11, 24, 25] are based on procedure summaries formulated as two-phase computation. Among those, Foster et al. [11] is the closest to the pointer analysis proposed in this dissertation.

CFL reachability [30] provides an alternative approach to context sensitivity. One of its merits over other approaches is that recursion can be handled also context-sensitively. Fähndrich et al. [31] employed a variant of CFL reachability with the unification-based modeling proposed by Steensgaard [9]. Das et al. [28] took a similar approach using the dataflow modeling proposed by Das [7] yet with one-level context sensitivity.
Guyer and Lin [32] applied *counter-example directed refinement* to pointer analysis. Counter-example directed refinement is a technique popular in model checking [33]. In this approach, refinement is driven by feedback from a less accurate abstraction. When results computed by less accurate pointer analysis (both flow- and context-insensitive) turn out to be insufficient, the accuracy is refined by adding flow or context sensitivity. To add context sensitivity, a procedure call is specialized by renaming the callee’s body. After being specialized, all the procedures are re-analyzed until a global fixed point is reached. From this perspective, the techniques proposed in this dissertation can also greatly aid the efficiency of their analysis.

Many algorithms, for instance, [34, 11], employ simplification techniques in an effort to construct a *concise yet observably-equivalent* summary of the analysis information. Constraint simplification in general has been thoroughly studied. Simplification has been used in context-insensitive frameworks to improve efficiency [13, 15, 14]. There has also been significant interest in constraint simplification in the context of subtyping, for instance, [35, 36, 37]. For a more complete list of references and in-depth discussion, we refer readers to [36]. In reducing the size of procedural summaries, The simplification algorithm proposed in this dissertation heavily depends on *escape* information extensively discussed in the context of object-oriented languages [38, 39, 40, 41, 42, 43, 44].

In the functional language domain, there has been significant interest in polymorphic flow analysis. Recently work such as [45, 46] have reduced the complexity to $O(n^3)$, where $n$ is a size of a type-annotated program potentially exponentially larger than its
type-erased counterpart. One key concept is shared by both algorithms—polymorphism is achieved through constraint instantiation rather than inlining.

There are two factors which make such an approach applicable. First, they exploit type information to reason and model information flow in programs. This implies that the calling contexts of a function have a fixed, known shape that is solely determined by type information. Second, there is no aliasing. This implies that the shape of intraprocedural data-flow is also fixed and independent to calling contexts.

The situation in context-sensitive points-to analysis is quite different. First, the shapes of calling contexts are not known a priori because this knowledge requires the points-to information itself. Second, the shape of the intraprocedural data-flow depends on calling contexts through the effect of aliasing. In general, the data-flow created by context-sensitive points-to analysis has a less regular structure and thus is more difficult to exploit.

One interesting difference between our work and [45, 46] is how recursion and indirect calls are handled. We handle recursive procedures context-insensitively by merging them into a single procedure. However, [45, 46] handle recursive procedures in a context-sensitive way. We also handle each instance of indirectly invoked procedures context-sensitively whereas [45, 46] allows polymorphism per each instance of a procedure appearance.
main() {
    foo1(&a, &b, &a);
    foo2(&c, &d, &a);
}

foo(x, y, z) {
    *x := y;
    w := *z;
}

(a)  

{x := &a; y := &b; z := &a;
 x := &c; y := &d; z := &a;
 *x := y;
 w := *z;
}(b)

**Figure 4.1** Walking example: (a) code fragment and (b) context-insensitive equivalent without procedural boundaries.

## 4.2 Two-Phase Computation

This chapter illustrates the limitation of previous summary-based algorithms and demonstrates how the proposed approach overcomes such limitations. Throughout this chapter, the code fragment in Figure 4.1 is used as a running example. Within this example, we focus on the following aspects.

1. In the first call to `foo()`, denoted `foo1()` in the code fragment, variable `a` declared in caller `main()` is updated with `&b`. Since `z` points to `a`, variable `w` declared in callee `foo()` is updated with `&b`.

2. In the second call to `foo()`, denoted `foo2()` in the code fragment, variable `c` declared in caller `main()` is updated with `&d`. Since `z` also points to `a`, variable `w` is updated with `&b`.  

32
Figure 4.1 (b) shows a context-insensitive equivalent of the code fragment in Figure 4.1 without procedural boundaries. Note that parameter passing between `main()` and `foo()` is captured with a number of pseudo assignments between formal and actual parameters. When this example is analyzed context-insensitively, a spurious result `a := &d` can be derived from the following trace.

\[
\begin{align*}
\*x &:= y \quad \text{[}x := &a]_1 \\
a &:= y \quad \text{[}y := &d]_2 \\
a &:= &d
\end{align*}
\] (4.1)

The trace above is flow-insensitive in the sense that ordering between assignments is ignored. In the trace, the assignments above the lines correspond to the conditions in which the assignment below the lines are derived. More rigorous discussion of flow-insensitive abstract execution will be discussed in the following chapter.

The trace above is spurious in the sense that two parameter assignments from different call sites, namely `x := &a` from the first call site and `y := &d` from the second call site, interact and lead to the spurious result `a := &d`. Therefore, to obtain accurate pointer information from this example, context sensitivity is critical.

In a typical summary-based algorithm, the goal of the bottom-up phase is to construct a concise flow function for each procedure. Given a calling context as an input, this function should return the final pointer information of the procedure, incorporating all the effects of its callees.

All flow functions are computed in a single bottom-up sweep of the call graph, propagating procedural summaries from callees to callers. The procedural summary is similar
main() { 
    x_1 := &a; y_1 := &b; z_1 := &a;
    *x_1 := y_1;
    foo_1(&a, &b, &a);
}

foo(x, y, z) {
    *x := y;
    w := *z;
}

(a)

Figure 4.2 Walking example after the bottom-up phase: (a) conventional approach and (b) proposed approach that removes the critical assignment *x := y after bottom-up propagation of procedural summaries.

to the flow function except that, instead of the final pointer information for the whole procedure, it returns only the change in pointer information visible to the callers, called
procedural side effects.

In Figure 4.1(a), the only assignment in callee foo() that causes procedural side effects is *x := y. Therefore, the summary of foo() consists of the single assignment
*x := y. In the bottom-up phase, as shown in Figure 4.2(a), a specialized version of *x := y is copied into each call site of foo() (for instance, *x_1 := y_1 for call site foo_1()) along with a number of assignments for parameter passing.

The goal of the following phase, usually called the top-down phase, is to compute the actual pointer information. Since the flow functions of each procedure is available,
it can be computed in a single top-down sweep of the call graph. The top-down phase begins by analyzing caller \texttt{main}. As shown in Figure 4.2(a), the inclusion of \texttt{foo}'s summary makes the effects of two procedure calls \texttt{foo} and \texttt{foo2} exist locally. At this moment, only the valid results \texttt{a := &b} and \texttt{c := &d} are derived. As expected, the spurious dataflow caused by interaction between two call sites is effectively eliminated. To analyze callee \texttt{foo}, the calling contexts of \texttt{foo} must be formed and propagated from \texttt{main} to \texttt{foo}. The calling context for the first and second call sites are as follows.

\[
\{a := \&b; c := \&d; x := \&a; y := \&b; z := \&a;\} \implies \texttt{foo1}
\]

\[
\{a := \&b; c := \&d; x := \&c; y := \&d; z := \&a;\} \implies \texttt{foo2}
\]

To achieve scalability, before analyzing callee \texttt{foo}, the calling contexts from multiple call sites are typically merged into a single calling context. Otherwise, the same procedure needs to be analyzed multiple times for different calling contexts. The following set of assignments corresponds to the single calling context in which the calling contexts for \texttt{foo1} and \texttt{foo2} are merged.

\[
\{a := \&b; c := \&d; x := \&a; y := \&b; z := \&a; x := \&c; y := \&d;\}
\]
It is apparent that, when this calling context is analyzed with the body of callee \texttt{foo()}, the spurious result \texttt{a := &d} is derived from the same flow-insensitive trace (4.1). In fact, the situation has not been changed at all from the perspective of callee \texttt{foo()} compared to the situation before the bottom-up phase. Therefore, all spurious results derived by the context-insensitive analysis would be reproduced when analyzing \texttt{foo()} in the top-down phase.

In practice, even with its negative impact on scalability, explicit copying of calling contexts during the propagation of pointer information from callers to callees only screens some of the contamination. When the pointer information of \texttt{a} is needed, pointer analysis clients will look up the output of \texttt{main()}’s flow function, which does not contain the spurious result \texttt{a := &d}. However, without true top-down context sensitivity, pointer analysis clients must look up the output of \texttt{foo()}’s flow function for \texttt{w} and will obtain the spurious result \texttt{w := &d}.

\textbf{Observation.} The spurious derivation of \texttt{a := &d} during the top-down phase can be avoided by removing \texttt{*x := y in foo()} following the propagation of \texttt{foo()}’s summary. After the bottom-up phase, the original assignment \texttt{*x := y in foo()} is irrelevant from the perspective of \texttt{main}, since the local copies of \texttt{foo()}’s summary already provide all necessary effects. Moreover, since the consequence of \texttt{*x := y} (the updates of variables \texttt{a} and \texttt{c}) are naturally available to callee \texttt{foo} as a part of the calling context, the original assignment \texttt{*x := y} is no longer necessary from the perspective of callee \texttt{foo()}. There-
fore, the removal of \( *x := y \) prevents spurious dataflow while leaving overall pointer information unchanged.

This observation is reflected in Figure 4.2(b) in which the problematic assignment \( *x := y \) is removed from callee \( \text{foo()} \). After its removal, the top-down phase derives only valid results. Moreover, since caller \( \text{main()} \) is truly unaffected by \( \text{foo()} \) after this change, the pointer information of \( \text{main()} \) does not need to be set aside through explicit copying. Effectively, a single run of the context-insensitive analysis is sufficient as the top-down phase, producing accurate pointer information. The rest of this dissertation is a generalization of this observation.
CHAPTER 5

OVERVIEW

In the previous chapter, we demonstrated the limitations of the conventional summary-based algorithms and made an observation how those limitations can be overcome. In this chapter, we develop the observation into the reformulation of the summary-based algorithm.

This chapter is organized as follows. In Section 5.1, we examine how spurious results are derived when programs are analyzed context-insensitively. In particular, we observe the close relation between derivation of spurious results and procedural side effects. In Section 5.3, we reformulate the two-phase computation so that procedural side effects are removed from callees after procedural summaries are propagated into callers. In the reformulated approach, since the bottom-up phase leaves the program free of procedural side effects, a single run of context-insensitive analysis is sufficient for the top-down phase, producing accurate pointer information without experiencing spurious dataflow due to the lack of context information. In Section 5.4, we discuss how escaping objects must be handled to maintain overall pointer information unchanged after procedural side effects are removed from callees. In Section 5.5, we briefly go over the key simplification opportunities available during the bottom-up phase. In Section 5.6, we present the overall organization of the proposed context-sensitive pointer analysis.
5.1 Source of Spurious Dataflow

The rationale behind the proposed algorithm is best demonstrated by examining how procedural side effects contribute to derivation of spurious dataflow when programs are analyzed context-insensitively. Reconsider the code fragment in Figure 5.1, repeated from Figure 4.1. By analyzing this example context-insensitively, we can derive the following traces.

\[
\begin{align*}
&a := y \\
&y := &b \\
&a := &b
\end{align*}
\]

Note that these two traces represent valid execution of the code fragment in Figure 5.1. The left trace represents the execution of the first call `foo1()` while the right trace represents the execution of the second call `foo2()`. However, intermixing between these traces results in spurious traces shown below.

\[
\begin{align*}
&a := y \\
&y := &d \\
&a := &d
\end{align*}
\]

\[
\begin{align*}
&c := y \\
&y := &b \\
&a := &c
\end{align*}
\]

The key problem in the left trace stems from the fact that `a := y` is valid only within the first call site while `y := &d` is valid only within the second call site. The right trace also has a similar problem.

It turns out that no such spurious interaction can occur if a program completely lacks procedural side effects. Even though a procedure may be called in many different call sites, without procedural side effects, there is no way in which the contexts from different
main() {
    foo1(&a, &b, &a);
    foo2(&c, &d, &a);
}

foo(x, y, z) {
    *x := y;
    w := *z;
}

Figure 5.1  Example repeated from Figure 4.1(a).

call sites can interact. Therefore, for side-effect free programs, full context sensitivity is
inherent even when they are analyzed context-insensitively. A formal proof is presented
in the following section.

5.2 Procedures without Side Effects

A call graph is a directed multi-graph \((V, E)\), where \(V\) is a set of procedures and \(E\)
a set of call sites. Since the proposed analysis merges all the recursive procedures into a
single procedure, we can assume that the call graph is acyclic. A call chain refers to a
path in the call graph.

Given an acyclic call graph, a partial ordering relation \(\sqsubseteq: \text{Proc} \times \text{Proc} \rightarrow \text{Bool}\)
can be defined as follows: \(p \sqsubseteq q\) if and only if \(p\) is a descendant of \(q\) in the call graph.
We can override this ordering to work with expressions as follows: \(e_u \sqsubseteq e'_u\) if and only if
\(h(u) \sqsubseteq h(v)\), where \(h(u)\) denotes the procedure declaring the variable \(u\). The symbol \(e_u\)
denotes an arbitrary expression that contains variable $u$. The symbol $h(u)$ denotes the declaring procedure of variable $u$.

Intuitively speaking, a procedure has a side effect if it modifies one of the local variables of its proper ancestor in the call graph. In context-insensitive analysis, such a situation can be detected by examining derivations as follows.

**Definition 5.2.1** In context-insensitive analysis, if a derivation of $e_u := e'_u$ contains a variable declared by a proper descendant of $h(u)$, then $e_u := e'_u$ is said to be a *side effect*.

Then, a program is said to be *free* of side effects if no side effect is derived by context-insensitive analysis. As stated in the following theorem, for a side-effect free program, context-insensitive analysis is as accurate as context-sensitive analysis. In the following, we write $\Psi \vdash_{ci} e_u := e'_u$ if the assignment $e_u := e'_u$ can be derived context-insensitively from the program $\Psi$. On the other hand, we write $\Psi \vdash_{cs} e_u := e'_u$ if the assignment $e_u := e'_u$ can be derived context-sensitively from the program $\Psi$.

**Theorem 5.2.1** Let us consider a side-effect free program $\Psi$. Then, $\Psi \vdash_{ci} e_u := e'_u$ iff $\Psi \vdash_{cs} e_{u,\alpha} := e'_{v,\beta}$ for some call chains $\alpha$ and $\beta$.

The if part is relatively straightforward. Given a trace $T$ built by the context-sensitive analysis, let $T'$ be the trace identical to $T$ except that subscripts are removed from all variables. Then, $T'$ is a trace built by the context-insensitive analysis.

The only-if part is proved as follows. In the following definition, the *param-set* of a derivation $T$ refers to the set of all the parameter assignments used in $T$.  

41
**Definition 5.2.2** Let $\mathcal{T}$ be a derivation that results in $e_u := e'_v$ in context-insensitive analysis. Then, if the param-set of $\mathcal{T}$ forms a unique call chain $\alpha$ from $h(v)$ to $h(u)$, $\mathcal{T}$ is said to match the call chain $\alpha$.

Such a derivation matches the call chain in the sense that the derivation matches a data flow formed by a sequence of the procedure calls in the call chain.

The following lemma claims that if a trace formed by context-insensitive analysis matches a call chain, then, the assignment derived from the trace is non-spurious in the sense that context-sensitive analysis also derives a version of the assignment. The proof can be done by structural induction.

**Lemma 5.2.1** Consider a trace $\mathcal{T}$ in context-insensitive analysis that derives $\mathcal{P} \vdash_{ci} e_u := e'_v$. Then, if $\mathcal{T}$ matches a call chain, then, $\mathcal{P} \vdash_{cs} e_u := e'_{u \beta}$ for some $\alpha$ and $\beta$.

The following lemma shows that, when applied to a side-effect free program, context-insensitive analysis constructs only traces with matching call chains. Therefore, when combined with the above lemma, the only-if part of Theorem 5.2.1 is proven.

**Lemma 5.2.2** Let us consider a side-effect free program $\mathcal{P}$. In such a case, if $\mathcal{P} \vdash_{ci} e_u := e'_v$ by a trace $\mathcal{T}$, then, $\mathcal{T}$ has a matching call chain.

The proof is done by structural induction on the trace $\mathcal{T}$ built by context-insensitive analysis. There exist four cases, each corresponding to an inference rule in Figure 3.1.

1. When the last small-step trace taken in $\mathcal{T}$ is Rule (3.1). In such a case, it is the only trace in $\mathcal{T}$.
(a) If $e_u := e'_u$ is a parameter assignment, then, $\mathcal{T}$’s param-set forms a call chain of length 1 from $h(v)$ to $h(u)$. Therefore, by definition, $\mathcal{T}$ has a matching call chain.

(b) Otherwise, $\mathcal{T}$’s param-set is an empty set, thus, forms an empty call chain. Since $h(v) = h(u)$, by definition, $\mathcal{T}$ has a matching call chain.

2. When the last derivation taken in $\mathcal{T}$ is Rule (3.2) in the following form.

$$
\begin{array}{c}
  u := w \quad w := &v \\
  \hline \\
  u := &v
\end{array}
$$

Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be the subtraces of $\mathcal{T}$ that result in $u := w$ and $w := &v$, respectively. From the induction hypothesis, $\mathcal{T}_1$ and $\mathcal{T}_2$ have matching call chains $\alpha$ from $h(w)$ to $h(u)$, and $\beta$ from $h(v)$ to $h(w)$, respectively. Then, since the param-set of $\mathcal{T}$ is a union of the param-sets of disjoint $\mathcal{T}_1$ and $\mathcal{T}_2$, it corresponds to a unique call chain from $h(v)$ to $h(u)$. Therefore, by definition, $\mathcal{T}$ has a matching call chain.

3. When the last derivation taken in $\mathcal{T}$ is Rule (3.4) in the following form.

$$
\begin{array}{c}
  *w := v \quad w := &u \\
  \hline \\
  u := v
\end{array}
$$

According to the rules in Figure 3.1, a dereferencing assignment is derived only by itself. Therefore, $*w := v$ must be an assignment explicit in the program. It implies that $w$ and $v$ are declared in the same procedure. Also, since the program is side-effect-free, from $u := v$, we can infer that $u \subseteq v$, which implies in turn that
u \subseteq w$. By the induction hypothesis, from $w := \&u$, we can infer that $w \subseteq u$.

Therefore, since $u \subseteq w$ and $w \subseteq u$, variables $w$ and $u$ must be declared in the same procedure. In such a case, the param-set of $\mathcal{T}$ is an empty set, thus corresponds to an empty call chain. Since $h(u) = h(v)$, by definition, $\mathcal{T}$ has a matching call chain.

4. When the last derivation taken in $\mathcal{T}$ is Rule (3.3) in the following form.

$$
\frac{u := *w \quad w := \&v}{u := v}
$$

Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be the subtraces of $\mathcal{T}$ that result in $u := *w$ and $w := \&v$, respectively. For a reason similar to the case above, $u := *w$ must be an assignment explicit in the program, thus, $h(u) = h(*w)$. Therefore, by the induction hypothesis, the param-set of $\mathcal{T}_1$ must be an empty set. Also by the induction hypothesis, from $w := \&v$, we can infer that $w \subseteq v$, thus, $u \subseteq v$. Therefore, $\mathcal{T}$ and $\mathcal{T}_2$ match the same call chain.

5.3 Migration of Side Effects

In reality, most programs have procedural side effects. Therefore, to exploit this finding, the proposed algorithm migrates procedural side effects from callees to callers so that the procedural side effects made within callees can be safely removed without affecting overall pointer information. The migration continues until the program is completely free of procedural side effects. At that point, a single run of context-insensitive analysis would
be sufficient to generate all pointer information accurately. In this section, we use the code fragment in Figure 5.3 as an example illustrating the key aspects of the side-effect migration algorithm. We focus on the following aspects.

1. In \texttt{bar()} , the assignment \( *x := z \) is the direct cause of side effects. By migrating \( *x := z \) from \texttt{bar()} into \texttt{foo()} , \texttt{bar()} is now free of procedural side effects while overall pointer information remains unchanged.

2. In \texttt{foo()} , the assignment \( *p := r \) is the direct cause of side effects. In this case, it is important that the assignments leading to side effects are compacted to maintain overall scalability.

3. After \texttt{foo()} 's side effects are migrated into \texttt{main()} , the code fragment is completely free of side effects while overall pointer information remains unchanged. At this point, even the context-insensitive analysis produces accurate pointer information without experiencing spurious dataflow.

The bottom-up phase of the proposed algorithm traverses the call graph in a reverse-topological order, migrating procedural side effects along the way. First, all potential procedural side effects are identified based on the concept of criticality. Informally speaking, we say that an assignment is critical if it has the potential to cause a side effect. In Example 5.2, the assignment \( *x := z \) is directly responsible for side effects, thus, critical.

The role of a procedural summary is to capture the derivation of side effects, therefore, when they are copied into the caller’s space, all necessary side effects will be locally derived at the level of callers without experiencing spurious intermix between the calling
main() {
    foo(&a, &b);
    foo(&c, &d);
}

foo(p, q) {
    bar(&r, q);
    bar(&s, &t);
    *p := r;
}

bar(x, y) {
    z := y;
    *x := z;
    w := *x;
}

**Figure 5.2** Example illustrating the side-effect migration algorithm.

contexts from different call sites. In such a case, critical assignments are used as a seed in the creation of the summary. The propagation of summaries permits removal of critical assignments from callees, leaving them free of side effects without affecting overall pointer information.

Summary size is the single most important factor governing the scalability of the overall algorithm. In an extreme case, an entire body of the callee can be used as a summary to be copied into the caller’s space. However, considering their transitive impact on scalability, it is necessary that the summary size is kept as small as possible. For instance, in `bar()`, two assignments `*x := z` and `z := y` can be compacted into a single assignment `*x := y` reducing the size of the summary. In addition, since the assignment `w := *x` causes only local effects, we can leave it in `bar()` without incorporating it into
the summary further reducing the summary size. The back-tracing algorithm is used to construct compacted assignments to be added into the summary. The following describes the actual back-tracing process for bar().

1. Back-tracing is initiated from the critical assignment \(*x := z\). However, since its right-hand side \(z\) is not a parameter, the addition of \(*x := z\) into the summary is deferred and each assignment that modifies \(z\) is examined.

2. The assignment \(z := y\) is the only assignment modifying variable \(z\). Thus, by substituting \(z\) with \(y\) in \(*x := z\), the dataflow is compacted into \(*x := y\). Since both \(x\) and \(y\) are parameters, no further compaction is necessary. Therefore, \(*x := y\) is added into bar’s summary.

For correctness, back-tracing must consider all data flow making it necessary to determine every location in which a variable may be modified. In reality, aliasing complicates this determination because complete dataflow information is no longer explicit within the program text. Consider procedure foo() following the analysis of bar().

1. Since the assignment \(*p := r\) has the potential to modify an external variable, to construct foo’s summary, back-tracing is initiated from the assignment.

2. Note that no explicit dataflow exists into \(r\). However, since \(r\)’s address was taken at \(x_3 := \&r\), there may be implicit dataflow that modifies \(r\). In this case, all dataflow of \(r\) can be determined through a local examination of foo(), resulting in the derivation of the implicit dataflow \(r := q\) from the copy of bar()’s summary.
3. By continuing back-tracing, \( *p := r \) and the implicit dataflow \( r := q \) are compacted into \( *p := q \). Since there is no other modification of \( r \), the assignment \( *p := q \) suffices as \( \text{foo}() \)'s summary.

After \( \text{foo}() \) is processed, all side effects in the example in Figure 5.2 can be removed while leaving overall pointer information unchanged. Then, the second phase applies a single run of the context-insensitive analysis and computes all pointer information completely and accurately. Note that, during this step, all the effects of the original critical assignments, namely, \( *x := z \) in \( \text{bar}() \) and \( *p := r \) in \( \text{foo}() \), exist at the caller’s level and are propagated back into the callees through parameter passing. For instance, \( *x \) correctly evaluates to \&b, \&d, and \&t in the assignment \( w := *x \) in \( \text{bar}() \).

In the example in Figure 5.2, an important aspect in \( \text{foo}() \) that simplifies summary generation is that the aliases of the address-taken variables \( r \) and \( s \) could be completely and accurately determined without knowledge of \( \text{foo}() \)'s calling contexts. Unfortunately, this is not always possible.

5.4 Handling Escaping Objects

Consider the code fragment in Figure 5.3(a). This example illustrates a case in which accurate alias relations may not be determined without the knowledge of the actual calling contexts. The following are the key aspects of the example that complicate the side-effect migration process.
main() {
  p1 := &a;
  q1 := &b;
  r1 := &c;
  s1 := &c;
  x1 := q1;
  *r1 := &x1;
  *p1 := **s1;

  foo1(&a, &b, &c, &c, x1);
  foo1(&a, &b, &c, &c, x1);
}

foo(p,q,r,s) {
  1:   x := q;
  2:   *r := &x;
  3:   *p := **s;
  4:   y := *x;
}

foo2(&d, &e, &f, &g, x2);

foo(p,q,r,s,x) {
  1:   x := q;
  2:   *r := &x;
  3:   *p := **s;
  4:   y := *x;
}

(a)  (b)

Figure 5.3 Code fragment illustrating a case in which accurate alias relations may not be determined without the knowledge of the actual calling contexts.

1. In foo(), by the assignment *r := &x, the address of local variable x is assigned to a variable nonlocal to foo().
2. In the first call site, since r and s point to the same variable, namely c, x is copied into *p at Line 3. Therefore, there is a flow from q through x into *p, allowing a := 1.

3. In the second call site, since r and s point to distinct variables, x is not copied into *p at Line 3. Instead, the content of *e is read. Therefore, there is no flow from q to *p, thus, b does not acquire 3.

The last two aspects raise a dilemma. For the first call site foo1(), the flow *p := q must be reflected in the summary. However, for the second call site foo2(), it would degrade the summary’s accuracy. The problem is that the data flow in the summary depends on the calling contexts which are not available when analyzing foo() during the bottom-up phase.

There are multiple approaches to dealing with this dilemma. First, the assignment *p := q can conservatively be added in the summary. Despite of accuracy degradation, this approach is sound. Second, two versions of summaries can be explicitly provided: one version for the case *r and *s are aliases and the other version for the case they are not. The summaries in [22] fall into this category.

Instead, we take another approach. As shown in Figure 5.3(b), by introducing x := q, *r := &x, and *p := **s into the summary, the alias of x can be resolved at the caller’s level. Note that, to do so, &x and all potential writes to x have been included into the summary.
This decision has an impact on summary size. In an extreme case, where every variable behaves like \( x \), the summary size will explode rapidly. For local variables, it is less an issue, since the lifetime of local variables is bounded by their declaring procedure and this constraint greatly affects typical usage of local variables.

In particular, an address of local variables are rarely exposed to non-local memory space. On the other hand, heap objects tend to be exposed to non-local memory space. It implies that, unless a special care for heap objects is provided, inclusion of those objects in the summary could impact scalability.

There is another consequence. As shown in Figure 5.3(b), the assignment \( y := *x \) accesses variable \( x \), but only as a consumer. Thus, it causes only local effects and did not become a part of \( \text{foo()} \)'s summary. In order to keep the overall information unchanged, especially for variable \( y \), we merge the dataflow into all copies of \( x \) by treating them as if they are parameters to procedure \( \text{foo()} \) as shown in Figure 5.3(b). Note that, after merging, a single run of the context-insensitive analysis can compute the pointer information completely and accurately.

### 5.5 Simplification Opportunities

To accurately analyze programs with a deep call structure and extensive amount of heap usage, it is important to distinguish the effects of different calls to the same procedure. The key concern is scalability. It is apparent that at least a naive approach, such as explicit inlining all procedure calls, does not scale beyond small programs.
Fortunately, large programs are typically structured as a hierarchy of components, usually reflected in the call structure. In two-phase computation, this hierarchical nature is exploited during the bottom-up phase by inlining *observable equivalent summaries* instead of the entire procedural body from callees to callers. To reduce the overhead involved in inlining, procedural summaries are applied through a number of *simplification algorithms* before they are copied into the caller's space. Given a procedural summary, a simplification algorithm returns a concise yet observably equivalent summary.

In this section, we examine key simplification opportunities available in context-sensitive pointer analysis. We also present a framework to systematically approach the simplification algorithms. In particular, we develop the connection between observable equivalence and procedural side effects. Throughout this section, we use the example in Figure 5.4 as a running example. This example is a simplified version of the code fragment in Figure 2.6. For instance, all control-flow structures and the fields of objects are ignored in this example. This example depicts a typical usage of procedure calls and heap-allocated data structures. We focus on the following aspects.

1. In the top-level procedure `main()`, two tables A and B are allocated. By calling `add_info()` twice, two objects C and D are inserted into these tables, respectively. After adding information, the information stored in the table A, namely C, is fetched.

2. In `add_info()`, the information q is stored into the table p via two routes to the procedure `insert()`. Note that two calls to `add_info()` have the same overall effects.
main() {
    a := &A; // Table
    b := &B; // Table

    c := &C; // Info
d := &D; // Info

    add_info (a, c);
    add_info (b, d);

    e := *a;
f := *e; // f points to &C
}

add_info(p, q) {
    insert(p, q, &r);
    insert(p, q, &s);
}

insert(u, v) {
    x := *u;
y := *x;

    Z := v;
z := &Z; // Node
    *u := z;
}

Figure 5.4 Example demonstrating key simplification opportunities available in context-sensitive pointer analysis. This example is a simplified version of the code fragment in Figure 2.6.

3. In insert(), the information stored in the table u is fetched. Note that it does not have any visual effects to its caller add_info(). Then, the new node Z holding the information v, is allocated and stored into the table u.
In the following subsections, we briefly go over the following three simplification opportunities available while propagating procedural summaries within this example: elimination of local-only dataflow, collapsing copy assignments, and merging equivalent variables.

5.5.1 Elimination of Local-Only Dataflow

Much dataflow made in a callee is invisible to its callers. It is apparent that removal of such dataflow from the procedural summary does not affect the pointer information of the callers.

Let us examine the procedure `insert()` in Figure 5.4. The first two assignments, `x := *u` and `y := *x`, dereferences the pointer `u`. However, since the local variable `y` is not used anywhere, their effects remain invisible from the perspective of the caller `add_info()`. Therefore, they can be safely excluded from the procedural summary without affecting observable equivalence. The following is the resultant procedural summary of `insert()`.

\[
\{Z := v; z := &Z; *u := z;\}
\]

Figure 5.5 shows the procedure `add_info()` after the procedure calls to `insert()` are replaced with distinct copies of `insert()`'s procedural summary.

54
5.5.2 Collapsing Copy Assignments

In many cases, copy assignments introduce unnecessary indirection in dataflow. In particular, during the bottom-up phase, many copy assignments are introduced into procedural summaries to capture the effect of parameter passing. Removing those unnecessary indirection is an important simplification opportunity.

Let us examine the procedure `add_info()` after the procedural summary of `insert()` is propagated. It is apparent that the address assignments \( w_1 := \&r \) and \( w_2 := \&s \) are subject of elimination since they do not participate in the dataflow affecting the caller `main()`.

Moreover, the pseudo assignments for parameter passing result in unnecessary indirection in the dataflow made in the procedure `add_info()`. For instance, two assignments \( u_1 := p \) and \( *u_1 := z_1 \) can be compacted into a single assignment \( *p := z_1 \). Figure 5.6 shows the procedure `add_info()` after all such indirection is removed in addition to elimination of local-only dataflow.

```c
add_info(p, q) {
    u1 := p; v1 := q; w1 := \&r;
    Z1 := v1; z1 := \&Z1; *u1 := z1;

    u2 := p; v2 := q; w2 := \&s;
    Z2 := v2; z2 := \&Z2; *u2 := z2;
}
```

**Figure 5.5** Procedure `add_info()` after the procedural summary of `insert()` replaces the procedure calls.
add_info(p, q) {
    Z1 := q; z1 := &Z1; *p := z1;
    Z2 := q; z2 := &Z2; *p := z2;
}

Figure 5.6 Procedure add_info() after copy assignments are collapsed.

5.5.3 Redundancy Removal

In many cases, the procedure calls made through different call chains have the same
effects. In such cases, the procedural summaries introduced for different call sites intro-
duce redundancy in dataflow. Removing such redundancy is an important simplification
opportunity.

There is still room for improvement for the procedure add_info(). In particular, two
objects Z1 and Z2 behave equivalently in the sense that they have the same points-to and
pointed-to-by sets. They have the same points-to set since the sources of the assignments
defining Z1 and Z2 are identical, namely the parameter q. They have the same pointed-
to-by set since their addresses are written into the same external variables by the store
assignments *p := Z1 and *p := Z2, respectively. Therefore, it is safe to merge two
objects Z1 and Z2. Merging can be performed by renaming all instances of the object Z2
with Z1 as follows.

add_info(p, q) {
    Z1 := q; z1 := &Z1; *p := z1;
    Z2 := &Z1; *p := Z2;
}

56
Consequently, the local variables \( z1 \) and \( z2 \) becomes equivalent since they only point to the same object \( z1 \). Therefore, we can safely rename all instances of the local variable \( z2 \) with \( z1 \). The following is the procedure \textbf{add\_info()} after redundancy is removed.

\[
\text{add\_info}(p, q) \{
    z1 := q; z1 := &z1; *p := z1;
\}
\]

### 5.6 Overall Algorithm

We conclude this chapter with presenting the overall structure of the proposed algorithm. In C programs, indirect calls can be made using function pointers. In the presence of indirect calls, a cyclic dependency exists between call graph construction and pointer analysis. Without having a complete call graph, pointer information may be incomplete. On the other hand, without complete pointer information, the call graph may be incomplete.

The proposed algorithm breaks this cyclic dependency using an iterative approach as in [18, 23]. It begins with an incomplete call graph consisting of only direct calls. Based on this incomplete call graph, pointer information is constructed. Then, using this as feedback, the call graph is updated. This process continues until there are no more changes to the call graph.

Each iteration consists of three phases: recursion merging, computation of pointer information, and call-graph updating. Since the bottom-up phase does not terminate in the presence of recursive procedures, they are merged into a single procedure, while the
recursive calls between them are converted into a set of parameter-passing assignments. Effectively, recursion is handled context-insensitively.

Recursion merging renders the call graph acyclic. Given such a call graph, pointer information is computed in two phases. The bottom-up phase propagates procedural summaries from callees to callers followed by the removal of the assignments causing spurious dataflow at the level of callees. Then, the top-down phase computes the actual pointer information using a single run of the context-insensitive analysis.
CHAPTER 6

DISCOVERING CRITICAL ASSIGNMENTS

Procedural side effects are the change in pointer information visible to callers. There are two cases. The first case is when the callee modifies one of its caller’s variable. The second case is when the callee modifies one of its own local variable that escapes the scope of the callee.

In this chapter, we present an algorithm that systematically discovers generation of procedural side effects during the bottom-up phase. In particular, we focus on finding assignments directly responsible for generation of procedural side effects called critical assignments.

6.1 Symbolic Calling Contexts

The difficulty of identifying critical assignments during the bottom-up phase lies in the fact that, when a procedure is being analyzed during the bottom-up phase, no information about its calling contexts is available. In the following sections, we present a technique to capture the effects of calling contexts symbolically so that critical assignments can be identified accurately even without consulting actual calling contexts.
6.1.1 Holding Property

Let us examine the procedure $\text{foo}()$ defined in Figure 6.1. Among the assignments in $\text{foo}()$, the store assignment $\ast y := q$ is a critical assignment. Consider the following procedure call.

$$\text{foo}(&a, &b);$$  \hspace{1cm} (6.1)

From this procedure call and the definition of the procedure $\text{foo}()$ in Figure 6.1, we can build the following trace resulting in the procedural side effect $a := &b$. Note that we placed brackets around the pseudo assignments for parameter passing.

\[
\begin{array}{rcl}
\ast x := p & x := \&y \\
y := p & [p := \&a] \\
\ast y := q & y := \&a \\
a := q & [q := \&b] \\
a := \&b
\end{array}
\]

(6.2)

In this trace, the direct cause of the procedural side effect $a := \&b$ is the store assignment $\ast y := q$. Therefore, $\ast y := q$ is a critical assignment of the procedure $\text{foo}()$.

Let us identify the criticality of the store assignment $\ast y := q$ without consulting the actual calling contexts. To capture the propagation of external variables flown into the procedure $\text{foo}()$ through parameter passing, we introduce the holding properties to the local variables of the procedure $\text{foo}()$.

Intuitively, a variable has the holding property if it points to an external variable flown into the procedure through parameter passing. Therefore, all parameters has the holding property by default, since they could point to external variables by parameter passing.
\begin{verbatim}
foo(p, q) {
    x := &y; *x := p; *y := q;
}
\end{verbatim}

**Figure 6.1** Example illustrating the high level concepts behind symbolic treatment of calling contexts.

Using the holding property, we replace the propagation of external addresses with the propagation of holding properties through the local variables. For instance, for a copy assignment \( x := y \), the holding property of its source \( y \) is propagated into its destination \( x \) as follows.

\[
\frac{x := y}{\text{hold}(x)} \quad \frac{\text{hold}(y)}{\text{hold}(x)}
\]

In such a case, we can test the criticality of a store assignment \( *x := y \) by examining the holding property of the variable \( x \) as follows.

\[
\frac{*x := y}{\text{critical}(*x := y)} \quad \frac{\text{hold}(x)}{\text{critical}(*x := y)}
\]

Effectively, we have reduced the decision of the criticality of a store assignment into decision of the holding property of the variable \( y \).

Using the holding property, let us identify the criticality of the store assignment \( *y := q \) in the example in Figure 6.1. In particular, we build a symbolic trace similar to
\begin{verbatim}
foo(p, q) {
    x := p; y := &x; *q := y;
}
\end{verbatim}

**Figure 6.2** Example illustrating how to identify escaping objects.

The trace shown in (6.2) except that the pseudo assignments p := &a and q := &b are replaced with the holding properties of the parameters p and q and the propagation of the external addresses &a and &b are replaced with the propagation of holding properties through the local variables of \texttt{foo}().

\[
\begin{align*}
*x &:= q & x &:= &y & \text{param}(p) \\
y &:= p & \text{hold}(p) \\
*y &:= q & \text{hold}(y) & \text{critical}(*y := q)
\end{align*}
\]

### 6.1.2 Escaping Property

To identify all critical assignments, it is also important to determine the *escaping* property of the local variables. Recall that we call a local variable escaping if its address is copied into a non-local space or an escaping variable. Let us examine the procedure \texttt{foo()} defined in Figure 6.2. In this example, the local variable \texttt{z} is an escaping object. Consider the following procedure call.

\begin{verbatim}
foo(&a, &b);
\end{verbatim}

\[(6.6)\]
foo(p,q,r,s) {
    y := &x; *p := y; z := *q; *z := r; *x := s;
}

Figure 6.3 Example illustrating how to capture the effect of escaping variables.

From this procedure call and the definition of the procedure foo() in Figure 6.2, we can build the following trace making the local variable \( x \) escape the procedure foo() through the external variable \( b \).

\[
\begin{align*}
  \text{*q := y} & \quad \text{q := &b} \\
  \text{b := y} & \quad \text{y := &x} \\
  \text{b := &x} 
\end{align*}
\]  

(6.7)

Let us identify the escaping property of the local variable \( x \) without consulting the actual calling contexts. It is relatively straightforward to determine the escaping property provided that the holding properties of local variables are available.

\[
\begin{align*}
  \text{*x := y} & \quad \text{y := &z} \quad \text{hold(x)} \\
  \text{escape(z)} 
\end{align*}
\]  

(6.8)

\[
\begin{align*}
  \text{x := &y} \quad \text{escape(x)} \\
  \text{escape(y)} 
\end{align*}
\]  

(6.9)

Using these derivation rules, we can build the symbolic trace similar to the trace given in (6.7) that concludes the escaping property of the local variable \( x \) as follows.

\[
\begin{align*}
  \text{param(q)} & \quad \text{*q := y} \quad \text{y := &x} \quad \text{hold(q)} \\
  \text{escape(x)} 
\end{align*}
\]  

(6.10)
In certain cases, the escaping variable could be defined implicitly through the interaction between the procedure being analyzed and the calling contexts. Let us examine the example in Figure 6.3. In this example, even though it is less obvious, the store assignment \( *x := s \) is a critical assignment. Consider the following procedure call.

\[
\text{foo(&a, &a, &b, &c)};
\]

From this procedure call and the definition of the procedure \( \text{foo()} \) in Figure 6.3, we can build a trace shown in Figure 6.4 resulting in the procedural side effect \( b := &c \). From this trace, since \( *x := s \) is the direct cause of \( b := &c \), we can conclude that \( *x := s \) is a critical assignment.

The following is the complication involved in determining the criticality of the store assignment \( *x := s \) without consulting the actual calling contexts. To determine the criticality of the assignment \( *x := s \), it is important to find out that the local variable \( x \) has the holding property. Consider the following subtrace of the trace shown in Figure 6.4 assigning the external address \&b into the local variable \( x \).

\[
\begin{align*}
\text{z := } & \text{q} & \text{[q := &a]} & \text{a := y} & \text{y := &x} & \text{[p := &a]} \\
\text{z := a} & \text{a := &x} \\
\text{[z := r]} & \text{z := &x} \\
\text{x := r} & \text{[r := &b]} \\
& \text{x := &b}
\end{align*}
\]
Figure 6.4 Trace built from the example in Figure 6.3 and procedure call in the text resulting in the procedural side effect \( b := \&c \).

In this trace, the fact that both parameters \( p \) and \( q \) point to the same external variable \( a \) is crucial in the sense that the aliasing relation between \( *p \) and \( *q \) establishes the dataflow from the parameter \( r \) to the local variable \( x \). In other words, in a calling context in which \( p \) and \( q \) do not point to the same external variable, such a dataflow is not established, therefore, \( x \) may not point to an external variable.

The important aspect of the trace shown above is that the local variable is updated implicitly through a complex interaction between the procedure and its calling contexts. From the perspective of analyzing the procedure without consulting actual calling contexts, it is too expensive to consider all possible alias relations between dereferences of parameters to find out all potential dataflow made through interaction between the procedure and calling contexts.

Fortunately, such a complex interaction is possible only for escaping variables. (Note that the local variable \( x \) is an escaping variable, since its address escapes the procedure
\( \text{param}(x) \lor \text{escape}(x) \)
\[
\frac{}{\text{hold}(x)}
\]

**Figure 6.5** Relationship between holding and escaping properties.

`foo()` through the external variable `a`). Therefore, by assuming that escaping variables have the holding property by default, such complex interaction can be captured correctly (even though it could be slightly conservative).

\[
\frac{}{\text{hold}(x)}
\]

Using this derivation rule, we can build the following symbolic trace that correctly discovers that the local variable `x` could point to an external variable.

\[
\frac{\text{param}(p)}{*p := y \quad y := &x}{\text{hold}(p)}
\]

\[
\frac{}{\text{escape}(x)}
\]

\[
\frac{}{\text{hold}(x)}
\]

### 6.2 Details of Criticality Detection Algorithm

In the previous section, we presented the high-level concepts of the symbolic treatment of calling contexts through several examples. In this section, we develop the details of the algorithm that determines the criticality of assignments.
The derivation rules of the criticality discovery algorithm are described in Figures 6.5, 6.6, and 6.7. The main information computed by the algorithm is the properties 

\[ \text{hold, escape : } \text{Var} \rightarrow \text{Bool} \].

The property \( \text{hold} \) is used to capture the propagation of external addresses flown into the procedure through parameter passing. The property \( \text{escape} \) is used to track which variables are escaping. By default, we assume that parameters have the holding property. Since we do not track the dataflow beyond the scope of the procedure, the worst case is assumed for the escaping variables (Figure 6.5).

The holding property is propagated from source to destination of copy and load assignments. The escaping property is assigned in two cases: when it is pointed to by a variable that appears as a source of a store assignment whose destination's variable has the holding property and when it is pointed to by another escaping variable (Figure 6.6).

\[
\frac{x := y \quad \text{hold}(y)}{\text{hold}(x)}
\]

\[
\frac{x := *y \quad \text{hold}(y)}{\text{hold}(x)}
\]

\[
\frac{\ast x := y \quad y := \&z \quad \text{hold}(x)}{\text{escape}(z)}
\]

\[
\frac{x := \&y \quad \text{escape}(x)}{\text{escape}(y)}
\]

**Figure 6.6** Discovering holding and escaping properties.
\[
\begin{align*}
\text{output}(x) & \quad \text{critical}(x := e) \\
\text{hold}(x) & \quad \text{critical}(\ast x := y)
\end{align*}
\]

**Figure 6.7** Identifying critical assignments.

Using the holding and escaping properties, we can identify critical assignments as follows. A store assignment \(\ast x := y\) is critical if \(x\) has the holding property. On the other hand, an assignment of the form \(x := e\) is critical if \(x\) is an escaping variable (Figure 6.7).
CHAPTER 7

BACK-TRACING ALGORITHM

In the previous chapter, we presented a technique to capture the effect of calling contexts symbolically. It allows us to reason how procedural side effects are created without analyzing a callee under different calling contexts.

In this chapter, we build upon the technique presented in the previous chapter and present the back-tracing algorithm to construct a concise summary from these properties. Back-tracing is initiated from the assignments identified as being directly responsible for creation of procedural side effects. The back-tracing algorithm reduces the size of the summary by condensing as many intermediate copy assignments as possible. Consider the following procedure.

\[
\text{foo}(p, q) \{ u := p; \; *q := u; \}
\]  \hspace{1cm} (7.1)

In this procedure, \( *q := u \) is the only assignment that is directly responsible for creation of procedural side effects. By performing back-tracing from the source of this assignment, namely \( u \), towards the intermediate copy assignment \( u := p \), we can collapse the dataflow causing the procedural side effect into a single assignment \( *q := p \). Therefore, the
following procedure is observably equivalent to the procedure declared in (7.1).

\[ \text{foo}(p, q) \{ *q := p; \} \quad (7.2) \]

Back-tracing can be performed on dereferences of variables, too. For instance, consider the following procedure.

\[ \text{foo}(p, q) \{ u := p; *u := q; \} \quad (7.3) \]

In this procedure, the store assignment \( *u := q \) is directly responsible for procedural side effects. By performing back-tracing from the destination of this assignment, namely \( *u \) towards the copy assignment \( u := p \), the dataflow creating procedural side effects can be compacted into a single assignment \( *p := q \). Therefore, the following procedure is observably equivalent to the procedure declared in (7.3).

\[ \text{foo}(p, q) \{ *p := q; \} \quad (7.4) \]

In the examples shown so far, we were able to eliminate all variables except parameters from the compact summary. However, in certain cases, it could be impossible to bypass all intermediate variables. For instance, consider the following example.

\[ \text{foo}(p, q) \{ u := *q; *p := u; \} \quad (7.5) \]
In this procedure, the assignment \( *p := u \) is directly responsible for procedural side effects. However, the use of \( u \) in this assignment cannot be bypassed since the definition of \( u \) involves a dereference. Since we explicitly break up multiple dereferences into a sequence of single dereferences, both \( u := *q \) and \( *p := u \) are needed in the summary.

### 7.1 Coping with Aliases

The primary complicating factor in the backtracing algorithm is the implicit dataflow induced by aliases. For instance, consider the following example.

\[
\text{foo}(p, q) \{ u := \&v; \ u := q; \ *p := v; \} \quad (7.6)
\]

In this example, the assignment \( *p := v \) is directly responsible for procedural side effects. However, there is no explicit write into its source \( v \). On the other hand, there is an implicit dataflow \( v := q \) induced by aliases between \( *u \) and \( v \). Through this implicit dataflow, it is possible that a procedural side effect is created by the assignment \( *p := v \). Taking this into account, dataflow \( *p := q \) must be reflected in the summary.

Fortunately, when the back-tracing algorithm is invoked, the local closure of the procedure is already available, since the side-effect discovery algorithm needs to take the implicit dataflow into account as well. The following procedure corresponds to the locally
closed form of the procedure declared in (7.6).

\[ \text{foo}(p, q) \{ \ u := \&v; \ *u := q; \ v := q; \ *p := v; \} \]  

(7.7)

Note that the variable \( v \) is non-escaping. It implies that all the aliases of \( v \) are already explicit in the locally closed form. For this reason, the back-tracing algorithm can safely perform back-tracing from the source \( v \) of the assignment \( *p := v \) towards the copy assignment \( v := q \), producing a compacted assignment \( *p := q \). It is clear that the assignment \( *p := q \) is sufficient for the summary of the procedure, therefore, the following is observably equivalent to the procedure declared in (7.6).

\[ \text{foo}(p, q) \{ *p := q; \} \]  

(7.8)

In the following, we discuss some theoretical properties involved in the back-tracing algorithm. In particular, we discuss why we can bypass non-escaping variables even though they are address-taken, thus, have implicit dataflow through them.

Intuitively, we can bypass a variable if we can fully discover its role on the creation of procedural side effects. More specifically, if the aliases of a variable can be completely determined, we may be able to remove the variable from the summary.

**Definition 7.1.1** The pointed-to-by (PTB) set of a variable is a set of all variables pointing to the variable.
The key observation is that, if a variable is non-escaping, the PTB set of the variable can be fully determined simply by forming closure of the procedural summary.

**Definition 7.1.2** Let $A$ be a set of assignments. Then, we define the closed form of $A$, denoted $\text{cls}(A)$, is a set of assignments $a$ such that $A \Downarrow a$.

**Definition 7.1.3** Let $A$ be a summary of procedure $p$. Then, we define the *address-pruned* form of $A$, denoted $\text{ap}(A)$, be a summary induced by all the assignments in $\text{cls}(A)$ except the address assignments of non-escaping variables.

If all implicit assignments derivable from some $u := \&v$, where $v$ is a non-escaping object, are made explicit, it is possible to remove all such $u := \&v$ from the summary.

**Lemma 7.1.1** Let $A$ be a summary of procedure $p$. Then, its address-pruned form $\text{ap}(A)$ is observably equivalent to $A$.

**Proof** Let $u$ be a non-escaping object in $A$. Then, by definition, a variable $v$ points to $u$ if and only if $v := \&u$ is derivable from $A$, therefore, explicitly contained in $\text{cls}(A)$. It implies that all dataflow through $u$ has become explicit as copy assignments. Then, since the address of $u$ is not reachable from any variables non-local to $A$, removing address assignments of $u$ do not affect observable equivalence. 

Therefore, after forming the local closure of a procedure summary, all non-escaping variables can be treated in the same way regardless whether they are address-taken or not. On the other hand, it could be unsafe to bypass an escaping variable, since not all writes into the variable are explicit just by examining the procedure alone.
\[\begin{align*}
*p & := w \quad w := & u \\
\quad & \quad \quad [*q \equiv *p] & *p & := & u \\
\quad & \quad \quad \quad *q & := & u \\
\quad & \quad \quad \quad **q & \equiv & u & u := & p \\
\quad & \quad \quad \quad *r & := & **q & **q & := & p \\
\quad & \quad \quad \quad \quad \quad & \quad \quad \quad & *r & := & p 
\end{align*}\]

**Figure 7.1** Establishment of \( *r := p \) when \( *p \) and \( *q \) alias. The symbol \( \equiv \) is used to represent alias relation between expressions.

### 7.2 Handling Escaping Objects

In some cases, it may be impossible to determine the nature of aliasing without exact knowledge of the calling contexts. Consider the following example, where syntax is abused for clarity.

\[
\text{foo}(p,q,r) \{ u := q; *p := w; w := &u; *r := **q; \}
\]  \hspace{1cm} (7.9)

Since the address of \( u \) flows into \( *p \), aliases of \( u \) cannot be determined without consulting calling contexts. Consider a calling context where \( *p \) nad \( *q \) alias, and \( **q \) and \( u \) alias. In this case, as depicted in Figure 7.1, a dataflow \( *r := q \) is established. However, if \( *p \) and \( *q \) do not alias, such dataflow is not established.

This raises a dilemma. If \( *p := q \) is reflected in the summary, the summary could be correct but imprecise, since \( *p := q \) may not be established in some cases. On the other hand, exclusion of \( *p := q \) results in an incorrect summary, since there are cases in which \( *p := q \) is established. In the back-tracing algorithm, we overcome this dilemma
by deferring the decision until the exact calling contexts become available. In particular, we add the assignments \( u := q, w := &u \) in addition to the assignments \( *p := w \) and \( *r := **q \) that are directly responsible for procedural side effects.

7.3 Details of Back-Tracing Algorithm

In this section, we develop the details of the back-tracing algorithm. The goal of the back-tracing algorithm is to provide an equivalent summary yet with fewer number of copy assignments. To achieve the goal, the back-tracing algorithm initiates back-tracing from the assignments directly responsible for procedural side effects and collapses intermediate copy assignments while performing back-tracing. The following concepts are central in the back-tracing algorithm.

1. An assignment is traced if it is determined that the effect of the assignment must be reflected in the summary. However, the decision how to reflect the effect of traced assignments is deferred until back-tracing is further performed.

2. If a traced assignment cannot be further back-traced, it is decided that reflecting the effect of the assignment is impossible without having the assignment explicitly in the summary. Therefore, the traced assignment must be added into the summary.

7.3.1 Initiation

Back-tracing is initiated from the assignments that are directly responsible for creation of procedural side effects. There are following two cases.
\begin{align*}
*x := y & \quad hold(x) \\
trace(*x := y) \\
\end{align*}

\begin{align*}
x := e & \quad escape(x) \\
trace(x := e) \\
\end{align*}

**Figure 7.2** Initiation of back-tracing.

1. A store assignment \( *x := y \) such that \( x \) has the holding property is directly responsible for creation of procedural side effects.

\begin{align*}
*x := y & \quad hold(x) \\
trace(*x := y) \\
\end{align*}

2. An assignment \( x := e \) such that \( x \) is an escaping variable is directly responsible for creation of procedural side effects.

\begin{align*}
x := e & \quad escape(x) \\
trace(x := e) \\
\end{align*}

### 7.3.2 Address Assignments

In the following, we consider various situations and discuss how back-tracing is performed. Consider that an address assignment \( x := \&y \) is traced. Then, the most direct way to reflect the fact that \( x := \&y \) into the compact summary is to add \( x := \&y \) itself
\[
\begin{align*}
\text{trace}(x := & y) \\
\text{add}(x := & y)
\end{align*}
\]

**Figure 7.3** Back-tracing for address assignments.

... into the compact summary.

\[
\begin{align*}
\text{trace}(x := & y) \\
\text{add}(x := & y)
\end{align*}
\]

### 7.3.3 Copy Assignments

Consider that a copy assignment \( x := y \) is traced. If the variable \( y \) is a parameter, back-tracing has reached the boundary and cannot proceed further. Therefore, \( x := y \) is added into the compact summary. If the variable \( y \) is an escaping variable, it is not safe to bypass the variable \( y \) since we cannot determine all aliases of \( y \) accurately without having actual calling contexts available. Therefore, back-tracing is stopped here and \( x := y \) is added into the compact summary. From this perspective, both a parameter and escaping variable behave as a back-tracing stopper. From now on, we call such variables *dead ends*.

\[
\begin{align*}
\text{param}(x) \lor \text{escape}(x) \\
\Rightarrow \text{end}(x)
\end{align*}
\]

\[
\begin{align*}
\text{trace}(x := y) \land \text{input}(y) \\
\Rightarrow \text{add}(x := y)
\end{align*}
\]

If \( y \) is not a dead end, there is a chance that we can bypass a copy assignment defining \( y \). Since \( y \) is not an escaping variable, all the dataflow through the variable \( y \) is explicit...
after the local colsure is formed. There are following three cases. (Note that $y$ cannot be explicitly defined by a store assignment. If there is a store assignment implicitly defining $y$, its effect must be already present in the locally closed procedure $p$.)

1. The variable $y$ is defined by an address assignment $y := \&z$. In such a case, $x := \&z$ is present in the locally closed procedure $p$. It implies that $x := \&z$ is also present in the compact summary. Therefore, there is nothing additional to do.

2. The variable $y$ is defined by a copy assignment $y := z$. In such a case, $x := y$ and $y := z$ can be compacted into a single copy assignment $x := z$.

\[
\begin{array}{c}
\text{trace}(x := y) \quad y := z \\
\hline
\text{trace}(x := z)
\end{array}
\]

3. The variable $y$ is defined by a load assignment $y := *z$. In such a case, $x := y$ and $y := *z$ can be compacted into a single load assignment $x := *z$.

\[
\begin{array}{c}
\text{trace}(x := y) \quad y := *z \\
\hline
\text{trace}(x := *z)
\end{array}
\]

### 7.3.4 Load Assignments

Consider back-tracing a load assignment $x := *y$. If $y$ is a dead end, back-tracing must be stopped and $x := *y$ is added into the compact summary.

\[
\begin{array}{c}
\text{trace}(x := *y) \quad \text{end}(y) \\
\hline
\text{add}(x := *y)
\end{array}
\]
trace(\(x := y\)) \quad \text{input}(y) \\
\quad \frac{\text{add}(\(x := y\))}{\text{trace}(\(x := y\))}

\[\]

\[
\text{trace}(\(x := y\)) \quad y := z \\
\quad \frac{\text{trace}(\(x := z\))}{\text{trace}(\(x := y\))}
\]

\[
\text{trace}(\(x := y\)) \quad y := \ast z \\
\quad \frac{\text{trace}(\(x := \ast z\))}{\text{trace}(\(x := y\))}
\]

Figure 7.4 Back-tracing copy assignments.

If \(y\) is not a dead end, there is a further chance for back-tracing. Since \(y\) is not an escaping variable, all the dataflow defining \(y\) must be explicit in the locally closed procedure \(p\). There are following three cases.

1. The variable \(y\) is defined by an address assignment \(y := \& z\). Then, \(x := z\) must be present in the locally closed procedure \(p\) and back-tracing will be performed with respect to \(x := z\). There is nothing additional to do.

2. The variable \(y\) is defined by a copy assignment \(y := z\). Then, \(x := \ast y\) and \(y := z\) are compacted into a single assignment \(x := \ast z\).

\[\]

\[
\text{trace}(\(x := \ast y\)) \quad y := z \\
\quad \frac{\text{trace}(\(x := \ast z\))}{\text{trace}(\(x := \ast y\))}
\]

3. The variable \(y\) is defined by a load assignment \(y := \ast z\). Since we do not allow multiple dereferences of a variable in a single assignment, \(x := \ast y\) is added into the
\[
\begin{align*}
\text{trace}(x := *y) \quad \text{end}(y) \\
\quad \text{add}(x := *y)
\end{align*}
\]

\[
\begin{align*}
\text{trace}(x := *y) \quad y := z \\
\quad \text{trace}(x := *z)
\end{align*}
\]

\[
\begin{align*}
\text{trace}(x := *y) \quad y := *z \\
\quad \text{add}(x := *y) \land \text{trace}(y := *z)
\end{align*}
\]

**Figure 7.5** Back-tracing load assignments.

summary. Instead, back-tracing proceeds with the other assignment \( y := *z \).

\[
\begin{align*}
\text{trace}(x := *y) \quad y := *z \\
\quad \text{add}(x := *y) \land \text{trace}(y := *z)
\end{align*}
\]

### 7.3.5 Store assignments

Consider back-tracing a store assignment \( *x := y \). Unlike the other kinds of assignments, back-tracing needs to be performed in two directions: one from the destination \( *x \) and the other from the source \( y \).

#### 7.3.5.1 From Destination

Consider back-tracing a store assignment \( *x := y \) from its destination \( *x \). If \( x \) is a dead end, back-tracing cannot be performed further from the destination. If the source
$y$ happens to be a dead end, too, back-tracing cannot be performed in any direction. In such a case, we add the assignment $\star x := y$ into the compact summary.

\[
\begin{array}{c}
trace(\star x := y) \quad end(x) \quad end(y) \\
\hline
add(\star x := y)
\end{array}
\]

If $x$ is not a dead end, there is a further chance of back-tracing. Since $x$ is not an escaping variable, all dataflow defining $x$ must be explicit in the locally closed procedure $p$. There are following three cases.

1. The variable $x$ is defined by an address assignment $x := \& z$. In such a case, $z := y$ must be present in the locally closed procedure $p$. Therefore, back-tracing is performed through $z := y$ and there is no additional thing to do.

2. The variable $x$ is defined by a copy assignment $x := z$. In such a case, $\star x := y$ and $x := z$ can be compacted into a single assignment $\star z := y$.

\[
\begin{array}{c}
trace(\star x := y) \quad x := z \\
\hline
trace(\star z := y)
\end{array}
\]

3. The variable $x$ is defined by a load assignment $x := \star z$. In such a case, since we do not allow multiple dereference in a single assignment, back-tracing for $\star x := y$ is stopped and $\star x := y$ is added into the compact summary. Instead, back-tracing is initiated from the other assignment $x := \star z$.

\[
\begin{array}{c}
trace(\star x := y) \quad x := \star z \\
\hline
add(\star x := y) \land trace(x := \star z)
\end{array}
\]

81
\[
\begin{align*}
\frac{\text{trace}(x := y) \quad \text{end}(x) \quad \text{end}(y)\text{trace}(x := y) \quad x := z}{\text{add}(x := y) \quad \text{trace}(z := y)} \\
\frac{\text{trace}(x := y) \quad x := z}{\text{add}(x := y) \land \text{trace}(x := z) }
\end{align*}
\]

**Figure 7.6** Back-tracing store assignments from destination.

### 7.3.5.2 From source

Consider back-tracing a store assignment \(x := y\) from its source \(y\). The case in which \(y\) is a dead end is already covered when we discussed back-tracing from destination. If \(y\) is not a dead end, there is a further chance for back-tracing. Since \(y\) is not an escaping variable, all definition into \(y\) must be explicit in the locally closed procedure \(p\). There are following cases.

1. The variable \(y\) is defined by an address assignment \(y := &z\). Unlike the other cases, the effect of the address assignment \(y := &z\) is not explicit in the locally closed procedure \(p\) due to the lack of calling contexts. Therefore, we add \(x := y\) into the compact summary and initiates back-tracing from \(y := &z\), which will eventually add \(y := &z\) into the compact summary, too.

\[
\begin{align*}
\frac{\text{trace}(x := y) \quad y := &z}{\text{add}(x := y) \land \text{trace}(y := &z) }
\end{align*}
\]
\[
\text{\texttt{trace}}(\star x := y) \quad y := &z
\]
\[
\text{\texttt{add}}(\star x := y) \land \text{\texttt{trace}}(y := &z)
\]
\[
\text{\texttt{trace}}(\star x := y) \quad y := z
\]
\[
\text{\texttt{trace}}(\star x := z)
\]
\[
\text{\texttt{trace}}(\star x := y) \quad y := *z
\]
\[
\text{\texttt{add}}(\star x := y) \land \text{\texttt{trace}}(y := *z)
\]

**Figure 7.7** Back-tracing store assignments from source.

2. The variable \(y\) is defined by a copy assignment \(y := z\). In such a case, \(\star x := y\) and \(y := z\) can be compacted into a single assignment \(\star x := z\).

\[
\text{\texttt{trace}}(\star x := y) \quad y := z
\]
\[
\text{\texttt{trace}}(\star x := z)
\]

3. The variable \(y\) is defined by a load assignment \(y := *z\). In such a case, since we do not allow multiple dereferences in a single assignment, \(\star x := y\) is added into the compact summary and back-tracing is initiated from the other assignment \(y := *z\).

\[
\text{\texttt{trace}}(\star x := y) \quad y := *z
\]
\[
\text{\texttt{add}}(\star x := y) \land \text{\texttt{trace}}(y := *z)
\]
CHAPTER 8

REDUNDANCY REMOVAL ALGORITHM

The back-tracing algorithm presented in Chapter 7 is effective in reducing the number of non-escaping variables in procedural summaries. However, since it cannot discover the exact dataflow through escaping variables, it leaves all escaping variables in procedural summaries. Considering that many heap objects tend to escape repeatedly as they are specialized upwards in the call graph, this issue must be addressed to avoid explosion of procedural summaries.

In this chapter, we present the *redundancy removal algorithm* to address the issue of escaping objects. It is based on the observation that, from the perspective of flow-insensitive pointer analysis, objects specialized through different call chains often behave identically after the details differentiating the effects of distinct procedure calls, such as numeric values, branch conditions, and control-flow structures, are ignored. The redundancy removal algorithm exploits this observation and merges observably equivalent escaping objects as early as possible.

We approach the problem of finding equivalent variables in procedural summaries by translating procedural summaries into finite automata and finding equivalent states in the finite automata.
8.1 Automata-Theoretic Approach

We regard two variables as being equivalent if they have identical points-to and pointed-to-by sets. We approach the problem of finding equivalent variables in procedural summaries by translating procedural summaries into finite automata. In this section, we only consider procedural summaries without escaping variables. The techniques to handle the implicit dataflow involving escaping variables is discussed in the following section.

In the example in Figure 8.1, variables \( z_1 \) and \( z_2 \) are equivalent in any calling contexts for the following reasons. If \( z_1 \) obtains an address from the copy assignment \( z_1 := p \), the variable \( z_2 \) obtains the same address from the copy assignment \( z_2 := p \). On the other hand, if \( z_1 \) obtains an address from the load assignment \( z_1 := *q \), the variable \( z_2 \) obtains the same address from the load assignment \( z_2 := *q \). Therefore, the points-to set of \( z_2 \) is a superset of the points-to set of \( z_1 \). For a similar reason, the points-to set of \( z_1 \) is a superset of the points-to set of \( z_2 \). Therefore, the points-to set of \( z_1 \) and \( z_2 \) is identical. Moreover, since both \( z_1 \) and \( z_2 \) are address-free, their pointed-to-by-sets are empty sets and trivially identical. Therefore, by definition, \( z_1 \) and \( z_2 \) are equivalent variables.

The equivalence between \( z_1 \) and \( z_2 \) becomes more apparent when we translate the procedure \( \text{foo}() \) into a finite automaton as shown in Figure 8.2. The translation rules are as follows. (1) Parameters become the input states of the finite automaton. (2) An address assignment \( x := \&y \) is translated into an A-transition from the state \( y \) to the
foo(p,q,r) {
    z1 := p; z1 := *q;
    x1 := *z1;

    z2 := p; z2 := *q;
    x2 := *z2;

    z3 := p; z3 := *r;
    x3 := *z3;
    
    *p := x1;
    *p := x2;
    *p := x3;
}

**Figure 8.1** Example illustrating the automata-theoretic approach to finding equivalent variables in procedural summaries.

state $x$. (3) A copy assignment $x := y$ is translated into a C-transition from the state $y$ to the state $x$. (4) A load assignment $x := *y$ is translated into an L-transition from the state $y$ to the state $x$. (5) At this moment, we ignore store assignments $*p := x1$, $*p := x2$, and $*p := x3$ since they do not affect the points-to sets nor pointed-to-by sets of variables $x1$, $x2$, and $x3$.

We interpret the finite automaton in Figure 8.2 as follows. In the procedure `foo()`, the variables $p$ and $q$ are parameters and behave as the inputs to the procedure `foo()`. Therefore, in the finite automaton, the states $p$ and $q$ behave as input states. (Usually in a finite automaton, there is only one start state [47]. In the finite automaton translated from procedural summaries, we assume that there is an implicit start state, from which an implicit transition with a distinct label exists to each input state.)
The string induced from a path from an input state reflects how the pointer information reachable from the parameter represented by the input state is accessed within the procedural summary. For instance, there is a path from the input state $p$ to the state $x_1$ inducing a string $(C \cdot L)$. This string indicates a dataflow through which the address stored in $p$ is first copied then dereferenced. (The net effect of this dataflow can be represented by the load assignment $x_1 := \ast p$.)

In such a case, the equivalence between the variables in the procedure $\text{foo()}$ can be inferred from the \textit{inbound-equivalence} between the variables in the finite automaton in Figure 8.2. In a finite automaton, we say that two states $s$ and $t$ are \textit{inbound-equivalent} if, for each input state $p$, the set of strings induced from the paths from $p$ to $s$ are identical to the set of strings induced from the paths from $p$ to $t$. For instance, the states $z_1$ and $z_2$ in the finite automaton are inbound-equivalent while $z_1$ and $z_3$ are not.
foo(p, q) {
    N1 := *q; x1 := *p; z1 := &N1; *x1 = z1;
    N2 := *q; x2 := *p; z2 := &N2; *x2 = z2;
}

**Figure 8.3** Example illustrating equivalence between escaping variables. In this example, escaping variables N1 and N2 are equivalent.

### 8.2 Handling Escaping Variables

As described in the previous section, translation of a procedural summary into a finite automaton is straightforward if the procedural summary contains no escaping variable. It is because, for a non-escaping variable, all its definitions become explicit after the procedural summary is locally closed.

On the other hand, even after local closure, certain dataflow into escaping variables still remains implicit because it requires interaction between the procedure and its calling contexts. The most straightforward approach to cope with this fact is to treat each escaping variable as a distinct input state as shown in Figure 8.4. This approach is certainly correct in the sense that inbound-equivalent states in finite automata always imply equivalence between variables in procedural summaries. However, it does not serve the purpose of the redundancy removal algorithm since no distinct escaping objects would be found to be equivalent.

For the procedure `foo()` in Figure 8.3, we can show that escaping variables N1 and N2 are equivalent based on the following reasoning. Firstly, we can show that the pointed-to-by sets of N1 and N2 are identical as follows.
**Figure 8.4** Finite automaton corresponding to the procedure `bar()` in Figure 8.3 when each escaping object is treated as a distinct input state. In such a case, since each input state is distinct, no distinct escaping objects would be found to be equivalent.

- The variables `x1` and `x2` are inbound-equivalent. Therefore, the dereferences of these variables correspond to the same set of memory locations, regardless of calling contexts.

- The store assignments `*x1 := z1` and `*x2 := z2` behave as the interfaces through which `N1` and `N2` escape into callers. Since `x1` and `x2` are inbound-equivalent, `N1` and `N2` escape exactly in the same way. Therefore, the pointed-to-by sets of `N1` and `N2` are always identical regardless of calling contexts.

Since the pointed-to-by sets of `N1` and `N2` are identical, there is an implicit write into `N1` from some variable `x` if and only if there is also an implicit write into `N2` from the same variable `x`. Moreover, since the explicit writes into `N1` and `N2` are identical in the procedure `bar()`, the points-to set of `N1` and `N2` are also identical. Therefore, `N1` and `N2` are equivalent.
Figure 8.5 Finite automaton that accurately accounts for the implicit dataflow into escaping variables. (a) Finite automaton demonstrating that the pointed-to-by sets of escaping variables N1 and N2 are identical. (b) Finite automaton in which the implicit dataflow into the escaping variables N1 and N2 is expressed as I-transition from the escaping points x1 and x2.

Figure 8.5 reflects this observation by replacing the \( \langle A \cdot S \rangle \) path from N1 to x1 with the I-transition from x1 to N1. The meaning of the I-transition from x1 to N1 is that there could be an implicit dataflow into N1 derived by dereferencing a variable pointed to by x1. Similarly, we replace the \( \langle A \cdot S \rangle \) path from N2 to x2 with the I-transition from x2 to N2. After such consideration, inbound equivalence between the states in the finite automaton in Figure 8.5(b) implies equivalence between the variables in the procedure \texttt{bar()}. 

There is another case in which a variable escapes into a non-local space. It is when an address of a variable is assigned into another escaping object. For instance, in the procedure \texttt{foo()} shown in Figure 8.6, variables M1 and M2 escape into callers through another escaping object N. In this case, the assignments \( N := \&M1 \) and \( N := \&M2 \) behave as the
foo(p, q) {
    x := *p; z := &N; *x := z;
    M1 := q; N := &M1;
    M2 := q; N := &M2;
}

(a)

Figure 8.6  Another case in which a variable escapes into callers. (a) Code fragment. (b) Finite automaton showing that variables M1 and M2 escape by being pointed to by another escaping object N. (c) The observation in (b) is reflected as J-transition from N to M1 and M2.

interfaces through which M1 and M2 escape. To reflect this fact into the finite automaton, we translate the address assignments N := &M1 and N := &M2 into J-transitions from N to M1 and M2, respectively. With this consideration, inbound equivalence between the states in the finite automaton in Figure 8.6(c) implies equivalent between the variables in the procedure foo().
8.3 Partitioning Algorithm

In the previous section, we described how to reduce the problem of finding equivalent variables in procedural summaries into the problem of finding inbound-equivalent states in finite automata. In this section, we present an algorithm that finds inbound-equivalent states based on optimistic and iterative partitioning, the concept first introduced by Hopcroft in the state-minimization algorithm designed for deterministic finite automata (DFA) [48].

8.3.1 Hopcroft’s Partitioning Algorithm for DFA

A finite automata is said to be deterministic if all outbound transitions from the same state are associated with distinct alphabets. For instance, the automaton in Figure 8.9(a) is deterministic. However, the automaton in Figure 8.9(b) is non-deterministic because from the state z, there are two outbound transitions, one to x and the other to y, associated with the same alphabet L.

In a DFA, no two distinct states reachable from any input state are inbound-equivalent. If an automaton is deterministic, given an input state p and a string α, there exists exactly one path from p that induces α. Therefore, if the set of strings induced from the paths from p to s includes α and the set of strings induced from the paths from p to t also includes the same string α, two states s and t must be identical.

On the other hand, it is possible that two distinct states in a DFA are outbound-equivalent. We say that two states are outbound-equivalent if, for each output state p,
the set of strings induced from the paths from $s$ to $p$ are identical to the set of strings
induced from the paths from $t$ to $p$. For instance, in the DFA in Figure 8.9(a), assuming
that the state $z$ is the output state, two states $x$ and $y$ are outbound-equivalent.

For this reason, the partitioning algorithm for DFA by Hopcroft [48] finds outbound-
equivalent states in DFA. The algorithm is optimal in the sense that it finds all outbound-
equivalent states given a DFA and has a low complexity $O(n \log n)$ where $n$ is the size of
the DFA.

8.3.2 AWZ’s Partitioning Algorithm for SSA Graphs

Alpern, Wegman, and Zadeck (AWZ) adapted Hopcroft’s partitioning algorithm to
find equivalent variables in Static Single Assignments (SSA) graphs [49]. Figure 8.7
shows a control-flow graph in an SSA form and corresponding finite automaton. The
invariant kept by SSA graphs is the exact inverse of that kept by DFA. In an SSA graph,
all inbound transitions from the same state are associated with distinct alphabets (as
oppose to the outbound transitions as in the case of DFA).

Figure 8.8 presents a pseudo code of AWZ’s partitioning algorithm. This algorithm
is almost identical to Hopcroft’s algorithm except that partitioning is performed with
respect to the outbound transitions instead of inbound transitions. An interesting aspect
of the algorithm is that after an equivalent class is partitioned into two subclasses, if the
equivalent class is not already in the worklist, only one of the subpart needs to be placed
into the worklist (the label L in the pseudo code in Figure 8.8).
This optimization reduces the complexity of the algorithm from $O(n^2)$ to $O(n \log n)$, where $n$ is the size of the finite automaton. The correctness is guaranteed by the fact that, in an SSA graph, all inbound transitions from the same state are associated with distinct alphabets.

### 8.3.3 Partitioning Algorithm for Procedural Summaries

In general, the finite automaton translated from procedural summaries do not obey the invariant of SSA graphs. As a consequence, the application of AWZ’s partitioning algorithm on procedural summaries sometimes leads to wrong results in the sense that two inequivalent variables are identified as being equivalent. Consider the example in
partition := \emptyset;
commonpart := set;

for each \( s \) in SSAgraph
  if (\( s \) is an output state)
    add \( \{s\} \) in partition;
  else
    add \( s \) in commonpart;

  if (commonpart is not empty)
    add commonpart in partition;

worklist := empty list;

for each part in partition
  add part in worklist;

while worklist is not empty
  current := remove first from worklist;

  for each alphabet \( a \)
    \( \text{outset} := \text{set of the states } s \text{ that some state in } \text{current} \)
    has \( a \)-transition to \( s \).

  for each part in partition
    if (part \cap outset \neq \emptyset \&\& part \not\subseteq outset)
      newpart := part − outset;

      if (part is in worklist)
        add newpart in worklist;
      else
        M:
        if (|part| < |newpart|)
          add part in worklist;
        else
          add newpart in worklist;

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{awz_partitioning_algorithm_for_ssa_graphs}
\caption{AWZ partitioning algorithm for SSA graphs.}
\end{figure}

Figure 8.10. Figure 8.11 describes the application of AWZ’s partitioning algorithm on the procedure \texttt{foo()} in Figure 8.10.

1. In the finite automaton translated from \texttt{foo()}, the states \( p \) and \( q \) are the input states. Therefore, the initial partition consists of \( \{p\}, \{q\}, \text{ and } \{x, y, u, v, w\} \) and
these parts form the initial worklist (Figure 8.11(1)).

\[
\text{worklist} = \{x, y, u, v, w\}, \{p\}, \{q\}
\]

2. Consider that the part (a) is removed from the worklist and the alphabet L is chosen for consistency checking. The states to which some state in the part (a) has an
foo(p, q) {
    x := *p;
    y := *q;

    u := *x;
    v := *y;
    z := *x; z := *y;
}

**Figure 8.10** Example illustrating the difference between partitioning for SSA graphs and NFA.

L-transition are x1, x2, and x3. Accordingly, the part (a) needs to be divided into two parts (a) and (b). Among these two subparts, since (a) has a smaller number of states, (a) is inserted back into the worklist (Figure 8.11(2)).

\[ worklist = \{p\}, \{q\}, \{x, y\} \]

3. Consider that the state p is removed from the worklist and the alphabet L is chosen for consistency checking. The only state to which the state p has an L-transition is the state x. Accordingly, the part (a) is further divided into the states x and y and the state x is added into the worklist (Figure 8.11(3)).

\[ worklist = \{q\}, \{x\} \]
4. The part \{q\} does not incur any partitioning and is removed from the worklist without any change.

\[ \text{worklist} = \{x\} \]
partition := \emptyset;
commonpart := \emptyset;

for each s in NFA
  if (s is an input state
    add \{s\} in partition;
  else
    add s in commonpart;
if (commonpart \neq \emptyset)
  add commonpart in partition;

worklist := empty list;
for each part in partition
  add part in worklist;
while worklist is not empty
  current := remove first from worklist;
  for each alphabet a
    outset := set of the states s that some state in current
             has a-transition to s.
    for each part in partition
      if (part \cap outset \neq \emptyset \& part \not\subseteq outset)
        newpart := part \setminus outset;
M:
  add part in worklist;
  add newpart in worklist;

Figure 8.12 Partitioning algorithm for procedural summaries.

5. Consider that the state x is removed from the worklist and the alphabet 1 is chosen
   for consistency checking. The states to which the state p has an L-transition is the
   states u and w. Therefore, the state v is removed from the part (b) and is placed
   into the worklist (Figure 8.11(4)).

   \[ \text{worklist} = \{v\} \]
6. Since $v$ does not have any outbound transition, it is removed from the worklist without any change. Since the worklist is empty, the partitioning algorithm terminates (Figure 8.11(4)).

In the final partition in Figure 8.11(4), the states $u$ and $w$ belong to the same part, therefore, are considered inbound-equivalent. However, while the state $w$ has an incoming string $(L \cdot L)$ from the input state $q$, the state $u$ does not have any incoming string from the input state $q$. Therefore, they are not inbound-equivalent and the partitioning algorithm is incorrect.

The problem lies in the fact that only the state $x$ is placed into the worklist after the part (a) in Figure 8.11(2) is divided into $\{x\}$ and $\{y\}$. Note that the state $w$ has an incoming L-transition from both the states $x$ and $y$.

The solution to this problem is to put both subparts into the worklist after the original part is divided during partitioning. In such a case, the state $y$ is also placed in the worklist and when it is eventually removed from the worklist, the part (b) in Figure 8.11(4) is divided into the states $u$ and $w$ (Figure 8.11(5)).
CHAPTER 9

EXPERIMENTAL RESULTS

To demonstrate the usefulness of the proposed techniques, we evaluate them on a broad range of benchmarks selected from the SPEC92, SPEC95, and SPEC2000 integer benchmark suites. All experimental results are derived from FULCRA Pointer Analysis Framework [50]. FULCRA is complied with no optimization and run on a 2.8 GHz Pentium 4 with 1GB RAM.

9.1 Efficiency

In Section 9.1.1, we present the analysis time when the analysis is configured field-insensitively. In Section 9.1.2, we present the analysis time when the analysis is configured field-sensitively.

9.1.1 Field-Insensitive Setting

In this section, we present the analysis time when the analysis is configured field insensitively. In addition to the techniques proposed in this dissertation, the analysis time presented in this section reflects other techniques implemented in FULCRA framework, most notably, the treatment of global variables. For many benchmarks, one of the
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>LOC</th>
<th>CI (s)</th>
<th>CS (s)</th>
<th>HS (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>008.espresso</td>
<td>13505</td>
<td>1</td>
<td>4</td>
<td>&gt; 3 hour</td>
</tr>
<tr>
<td>023.eqntott</td>
<td>3393</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>099.go</td>
<td>28547</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>124.m88ksim</td>
<td>17251</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>129.compress</td>
<td>1426</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>130.li</td>
<td>6930</td>
<td>8</td>
<td>313</td>
<td>&gt; 3 hour</td>
</tr>
<tr>
<td>132.jpeg</td>
<td>25897</td>
<td>3</td>
<td>&gt; 3 hour</td>
<td>70</td>
</tr>
<tr>
<td>134.perl</td>
<td>23969</td>
<td>2</td>
<td>44</td>
<td>&gt; 3 hour</td>
</tr>
<tr>
<td>164.gzip</td>
<td>7759</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>175.vpr</td>
<td>16973</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>176.gcc</td>
<td>205747</td>
<td>26</td>
<td>1700</td>
<td>&gt; 3 hour</td>
</tr>
<tr>
<td>181.mcf</td>
<td>1909</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>186.crafty</td>
<td>18977</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>197.parser</td>
<td>10924</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>253.perlbmk</td>
<td>57541</td>
<td>4123</td>
<td>&gt; 3 hour</td>
<td>&gt; 3 hour</td>
</tr>
<tr>
<td>254.gap</td>
<td>39674</td>
<td>142</td>
<td>1873</td>
<td>&gt; 3 hour</td>
</tr>
<tr>
<td>255.vortex</td>
<td>52634</td>
<td>2</td>
<td>46</td>
<td>200</td>
</tr>
<tr>
<td>256.bzip2</td>
<td>4637</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>300.twolf</td>
<td>19749</td>
<td>1</td>
<td>2</td>
<td>5</td>
</tr>
</tbody>
</table>

**Figure 9.1** Analysis time for the field-insensitive setting (rounded to the nearest non-zero second). The column LOC shows the lines of code, CI the analysis time of the context-insensitive analysis, CS the analysis time of the context-sensitive analysis without heap object specialization.

The main reason behind the longer analysis time is found to be the dataflow through global variables. By default, global variables are escaping variables. Therefore, they don’t easily disappear from procedural summaries during the bottom-up phase. Fortunately, the dataflow purely between global variables can be promoted to the root of the call graph as soon as it is discovered. However, the dataflow between local variables and global variables remains in the summary unless a special technique is employed, such as the one
proposed by Nystrom [50]. With this technique, global variables are completely excluded from the context-sensitive part of the analysis.

The column LOC in Figure 9.1 shows the program size in terms of the lines of code. The column CI shows the analysis time of the context-insensitive analysis and the column CS and HS show the analysis time of the context-sensitive analyses without heap-object specialization and with heap-object specialization, respectively. Since scalability is of a primary concern, analysis time is limited to 3 hours.

When analyzed context-insensitively (the column CI), all benchmarks finish within the time limit. While most benchmarks take no more than a couple of minutes, 253.perlbmk takes almost 3 hours to finish. It is because 253.perlbmk has an extensive amount of address assignments that propagate collectively through various dataflow. This problem is addressed by Nystrom [50] in a more enhanced version of FULCRA framework (See the field-sensitive results in Section 9.1.2).

When analyzed context-sensitively without heap-object specialization (the CS field), many benchmarks still finish within time limit, although several benchmarks such as 132.ijpeg and 253.perlbmk fail to finish within 3 hours. The problem of 132.ijpeg is that without having enough heap objects specialized, the call graph is corrupted and context sensitivity becomes a pure burden to the analysis.

In pointer analysis, abstraction of heap objects greatly affects overall quality (both accuracy and efficiency). In many context-sensitive pointer analyses [23, 18, 24, 26, 27], heap specialization is performed by specializing heap objects along call chains. Even though imperfect, this simple approach is found to be useful in eliminating many sources
of spurious dataflow. On the other hand, uncontrolled heap object specialization can quickly overload the analysis process impacting scalability. Therefore, while heap-object specialization is necessary, a controlling method is needed to provide a finer resolution of heap object usage while leaving overall scalability unaffected.

When heap-object specialization is turned on, more benchmarks fail to finish within the time limit. For 008.espresso, the problem is that there are too many escaping objects and due to field insensitivity. However, due to the spurious dataflow caused by field insensitivity, the redundancy removal algorithm becomes largely ineffective and can't detect many equivalent heap objects. This problem disappears when field sensitivity is turned on as shown in the following subsection (See Figures 9.2 and 9.3).

9.1.2 Field-Sensitive Setting

In pointer analysis, whether or not one distinguishes among fields of data structures can greatly affect the analysis results. Despite its benefits, since the C language is not type-safe, using syntactic field information obtained from programs could result in unsafe analysis results. The implementation of field sensitivity in FULCRA framework [50] handles many abuses of C programs in a safe and accurate way. In the following, we briefly go over how fields are handled in FULCRA. For more details, we refer readers to [50].

For field sensitivity, FULCRA is offset-based, as opposed to using field names as in [19]. Each variable is composed of a size and accessed offsets. The size of named variables are limited to the size of the defined type. The size of heap objects are set to the size
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>CI (s)</th>
<th>CS (s)</th>
<th>HS (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>008.espresso</td>
<td>1</td>
<td>1</td>
<td>&gt; 1 hour</td>
</tr>
<tr>
<td>023.eqntott</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>099.go</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>124.m88ksim</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>129.compress</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>130.li</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>132.jpeg</td>
<td>4</td>
<td>36</td>
<td>2</td>
</tr>
<tr>
<td>134.perl</td>
<td>3</td>
<td>29</td>
<td>&gt; 1 hour</td>
</tr>
<tr>
<td>164.gzip</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>175.vpr</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>176.gcc</td>
<td>45</td>
<td>190</td>
<td>&gt; 1 hour</td>
</tr>
<tr>
<td>181.mcf</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>186.crafty</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>197.parser</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>253.perlbmk</td>
<td>129</td>
<td>161</td>
<td>168</td>
</tr>
<tr>
<td>254.gap</td>
<td>1</td>
<td>249</td>
<td>232</td>
</tr>
<tr>
<td>255.vortex</td>
<td>6</td>
<td>11</td>
<td>744</td>
</tr>
<tr>
<td>256.bzip2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>300.twolf</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Figure 9.2** Analysis time for the field-sensitive setting when the techniques proposed in [50] are integrated. Despite increased accuracy due to field sensitivity, the analysis time is in general reduced compared to the field-insensitive results in Figure 9.1.

of the largest type in the program. The accessed offsets for each variable are initially set to empty, but are adjusted throughout the analysis as new updates are discovered. Thus the analysis may detect accessed offsets beyond those explicit to the variable’s defined type. Finally, the analysis can disable field sensitivity for individual variables and will exercise this ability to further improve safety or speed (at the expense of accuracy).

Aside from improving safety, the mechanism for field sensitivity in FULCRA permits variable-sized assignments for object accesses. For example, structure-to-structure copy-
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>CI</th>
<th>CS</th>
<th>HS (s)</th>
<th>HS* (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>008.espresso</td>
<td>1</td>
<td>1</td>
<td>&gt; 1 hour</td>
<td>32</td>
</tr>
<tr>
<td>134.perl</td>
<td>3</td>
<td>29</td>
<td>&gt; 1 hour</td>
<td>&gt; 1 hour</td>
</tr>
<tr>
<td>176.gcc</td>
<td>45</td>
<td>190</td>
<td>&gt; 1 hour</td>
<td>722</td>
</tr>
<tr>
<td>255.vortex</td>
<td>6</td>
<td>11</td>
<td>744</td>
<td>149</td>
</tr>
</tbody>
</table>

**Figure 9.3** Effect of merging equivalent heap objects on analysis time (column HS*). For 008.espresso, 134.perl, and 255.vortex, this technique greatly reduces the analysis time while 134.perl still does not finish within an hour even with this technique.

Field sensitivity also requires significant extension to all components the context-sensitive analysis including the criticality-detection algorithm (in Chapter 6), the back-tracing algorithm (in Chapter 7) and the redundancy removal algorithm (in Chapter 8). For instance, for the back-tracing algorithm, the challenge is how to handle varying width assignments. For the redundancy removal algorithm, the challenge is how to handle offsets associated with assignments. For more details, we refer readers to [50].

Figure 9.2 shows the analysis time for the field-sensitive setting. This analysis time also reflects many other innovations on context sensitivity made by Nystrom such as faster call-graph discovery, incremental flow computation between call-graph updates and context-insensitive treatment of global variables already mentioned in Section 9.1.1. For more details, we refer readers to [50].

The columns CI and CS show the analysis time for the context-insensitive analysis and context-sensitive analysis, respectively. In general, the analysis time in the field-
sensitive setting is much better than the one in the field-insensitive setting despite of the increased accuracy due to field sensitivity. It clearly shows the effectiveness of the techniques to handle field sensitivity in FULCRA framework For more details, we refer readers to [50].

The column HS in Figure 9.2 shows the analysis time of the context-sensitive analysis when heap objects are cloned along call chains under the field-sensitive setting. In this column, the redundancy removal algorithm for address-taken variables (described in Section 8.2) is not applied. Therefore, the back-tracing algorithm exploiting escape information of address-taken variables is the only mechanism reducing the overhead involved in heap-object specialization along call chains (described in Section 7.1).

The analysis time in the column HS indicates that the back-tracing algorithm is surprisingly effective in keeping the cost of heap-object specialization under control. However, for a number of benchmarks, the back-tracing algorithm itself is found to be insufficient to suppress the explosive nature of heap-object specialization.

The column HS* in Figure 9.3 shows the effect of the redundancy removal algorithm designed for address-taken variables (presented in Section 8.2) on the five benchmarks that heap-object specialization affects the analysis time significantly in the column HS in Figure 9.2. The analysis time in the column HS* shows that the redundancy removal algorithm is effective for three benchmarks (008.espresso, 176.gcc, 255.vortex) while 134.perl still doesn’t finish within 1 hour. For these benchmarks, approximation such as selective context sensitivity proposed in [50] seems to be unavoidable when a higher degree of efficiency is desired.
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>CS/CI</th>
</tr>
</thead>
<tbody>
<tr>
<td>008.espresso</td>
<td>0.89</td>
</tr>
<tr>
<td>023.eqntott</td>
<td>0</td>
</tr>
<tr>
<td>099.go</td>
<td>0</td>
</tr>
<tr>
<td>124.m88ksim</td>
<td>0.04</td>
</tr>
<tr>
<td>129.compress</td>
<td>0</td>
</tr>
<tr>
<td>130.li</td>
<td>0</td>
</tr>
<tr>
<td>132.jpeg</td>
<td>0.8</td>
</tr>
<tr>
<td>134.perl</td>
<td>0.1</td>
</tr>
<tr>
<td>164.gzip</td>
<td>0.02</td>
</tr>
<tr>
<td>175.vpr</td>
<td>0.35</td>
</tr>
<tr>
<td>176.gcc</td>
<td>0.02</td>
</tr>
<tr>
<td>181.mcf</td>
<td>0</td>
</tr>
<tr>
<td>186.crafty</td>
<td>0.33</td>
</tr>
<tr>
<td>197.parser</td>
<td>0</td>
</tr>
<tr>
<td>253.perlbmk</td>
<td>0.04</td>
</tr>
<tr>
<td>254.gap</td>
<td>0.01</td>
</tr>
<tr>
<td>255.vortex</td>
<td>0</td>
</tr>
<tr>
<td>256.bzzip2</td>
<td>0</td>
</tr>
<tr>
<td>300.twolf</td>
<td>0</td>
</tr>
</tbody>
</table>

**Figure 9.4** Fractional decrease in the points-to set size under the context-sensitive analysis without heap-object specialization (CS) when compared to that under the context-insensitive analysis (CI).

### 9.2 Accuracy

In this section, we present the accuracy comparison between the context-insensitive and context-sensitive analyses. Accuracy is measured in two different ways. In Section 9.2.1, we present the points-to set sizes of the context-insensitive analysis and context-sensitive analysis without heap-object specialization. The comparison based on points-to set sizes is appropriate when objects generated by both analyses are identical or comparable. However, it is difficult to do so when heap objects are specialized. There-
Figure 9.5 Calculation of accesses per object. Each node represents a heap object created during the analysis. The edge between heap objects represent the fact that the object at the head of the edge is a specialized version of the object at the tail along the call site labeled on the edge. The number in each node represents the number of loads (or stores) that appear to access the object. Then, the accesses per object is the average of those numbers across all objects. For this diagram, the accesses per object is $(5 + 5 + 10 + 30 + 50)/5 = 20$.

Therefore, in Section 9.2.2, for the comparison between the context-insensitive analysis and context-sensitive analysis with heap-object specialization, we use a different metric called *accesses-per-object* proposed by Nystrom [50].

### 9.2.1 Points-to Set Size

Figure 9.4 compares the accuracy of the context-insensitive analysis and context-sensitive analysis under the field-sensitive setting in terms of the reduction in points-to set sizes. This is calculated by summing the points-to set size of every location (variable offset) in the benchmark. While this may not directly correlate to beneficial accuracy improvement, we feel it provides useful comparison. There is a substantial range in
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>NON-HEAP CI</th>
<th>NON-HEAP HS</th>
<th>HEAP CI</th>
<th>HEAP HS</th>
</tr>
</thead>
<tbody>
<tr>
<td>008.espresso</td>
<td>1.4</td>
<td>1.4</td>
<td>363.8</td>
<td>33.9</td>
</tr>
<tr>
<td>023.eqntott</td>
<td>1.5</td>
<td>1.5</td>
<td>na</td>
<td>na</td>
</tr>
<tr>
<td>099.go</td>
<td>2.4</td>
<td>2.4</td>
<td>na</td>
<td>na</td>
</tr>
<tr>
<td>124.m88ksim</td>
<td>1.6</td>
<td>1.6</td>
<td>37.2</td>
<td>37.2</td>
</tr>
<tr>
<td>129.compress</td>
<td>0.6</td>
<td>0.6</td>
<td>na</td>
<td>na</td>
</tr>
<tr>
<td>130.li</td>
<td>51.3</td>
<td>51.3</td>
<td>395.3</td>
<td>395.3</td>
</tr>
<tr>
<td>132.ijpeg</td>
<td>11.5</td>
<td>10.6</td>
<td>672.7</td>
<td>14.7</td>
</tr>
<tr>
<td>134.perl</td>
<td>20.9</td>
<td>7.8</td>
<td>3720.5</td>
<td>na</td>
</tr>
<tr>
<td>164.gzip</td>
<td>1.4</td>
<td>1.4</td>
<td>6.0</td>
<td>4.9</td>
</tr>
<tr>
<td>175.vpr</td>
<td>1.3</td>
<td>1.3</td>
<td>541.5</td>
<td>3.4</td>
</tr>
<tr>
<td>176.gcc</td>
<td>10.1</td>
<td>9.1</td>
<td>12676.7</td>
<td>3927.4</td>
</tr>
<tr>
<td>181.mcf</td>
<td>1.3</td>
<td>1.3</td>
<td>52.0</td>
<td>26.0</td>
</tr>
<tr>
<td>186.crafty</td>
<td>2.0</td>
<td>2.0</td>
<td>15.7</td>
<td>15.5</td>
</tr>
<tr>
<td>197.parser</td>
<td>3.7</td>
<td>3.7</td>
<td>802.5</td>
<td>802.5</td>
</tr>
<tr>
<td>253.perlbmk</td>
<td>864.4</td>
<td>841.5</td>
<td>5066.0</td>
<td>5059.5</td>
</tr>
<tr>
<td>254.gap</td>
<td>88.8</td>
<td>88.8</td>
<td>5106.0</td>
<td>5106.0</td>
</tr>
<tr>
<td>255.vortex</td>
<td>4.6</td>
<td>4.6</td>
<td>1936.0</td>
<td>1089.4</td>
</tr>
<tr>
<td>256.bzip2</td>
<td>1.2</td>
<td>1.2</td>
<td>12.7</td>
<td>12.7</td>
</tr>
<tr>
<td>300.twolf</td>
<td>2.3</td>
<td>2.3</td>
<td>3017.0</td>
<td>77.1</td>
</tr>
</tbody>
</table>

**Figure 9.6** Loads-per-object for non-heap variables (NON-HEAP) and heap objects (HEAP) under the context-insensitive analysis (CI) and context-sensitive analysis without (CS) and with heap-object specialization (HS). 023.eqntott and 099.go use no heap object, therefore, the corresponding fields are marked “na”. The context-sensitive analysis with heap-object specialization does not finish for 134.perl, therefore, the corresponding field is also marked “na”.

benefit with some benchmarks showing little or no benefit (256.bzip and 300.twolf) while others show a fairly substantial reduction (008.espresso and 176.vpr).
9.2.2 Accesses per Object

When comparing two analyses that generate the same number of objects, points-to set sizes provide a convenient way to compare the accuracy of those analyses. However, when the numbers of objects generated by them are different, it is difficult to compare accuracy based on points-to set sizes.

For this reason, in this dissertation, we use a different metric called *accesses per object* [50] to measure the accuracy for comparison between the context-insensitive analysis and context-sensitive analysis with heap-object specialization. Figure 9.5 describes how accesses per object are calculated. In this figure, each node represents a heap object created during the analysis. The edge between heap objects represent the fact that the object at the head of the edge is a specialized version of the object at the tail along the call site labeled on the edge. The number in each node represents the number of loads (or stores) that appear to access the object. Then, the accesses per object is the average of those numbers across all objects. For this diagram, the accesses per object is
\[(5 + 5 + 10 + 30 + 50)/5 = 10.\]

Figure 9.6 compares the loads-per-object for non-heap variables (NON-HEAP) and heap objects (HEAP) obtained from the context-insensitive analysis (CI) and context-sensitive analysis with heap-object specialization (HS). Figure 9.7 shows the fractional decrease of the loads-per-object for non-heap variables (a) and heap objects (b) between CI and HS.
Figure 9.7 Fractional decrease in the loads-per-object under CS and HS. (a) for non-heap variables and (b) for heap objects when compared to CI.

Figure 9.7(a) indicates that context sensitivity and heap-object specialization have little impact on loads-per-object except for a couple of benchmarks such as 132.ijpeg and 176.gcc. In particular, the reduction in 132.ijpeg is substantial (about 0.8). It is believed that the quality of the call graphs constructed by CI and HS greatly affects the
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>NON-HEAP</th>
<th></th>
<th>HEAP</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CI</td>
<td>HS</td>
<td>CI</td>
<td>HS</td>
</tr>
<tr>
<td>008.espresso</td>
<td>0.6</td>
<td>0.6</td>
<td>114.3</td>
<td>10.2</td>
</tr>
<tr>
<td>023.eqnottt</td>
<td>0.6</td>
<td>0.6</td>
<td>na</td>
<td>na</td>
</tr>
<tr>
<td>099.go</td>
<td>0.5</td>
<td>0.5</td>
<td>na</td>
<td>na</td>
</tr>
<tr>
<td>124.m88ksim</td>
<td>0.6</td>
<td>0.6</td>
<td>5.1</td>
<td>5.1</td>
</tr>
<tr>
<td>129.compress</td>
<td>0.3</td>
<td>0.3</td>
<td>na</td>
<td>na</td>
</tr>
<tr>
<td>130.li</td>
<td>12.3</td>
<td>12.3</td>
<td>92.7</td>
<td>92.7</td>
</tr>
<tr>
<td>132.ijpeg</td>
<td>5.0</td>
<td>4.6</td>
<td>293.4</td>
<td>9.0</td>
</tr>
<tr>
<td>134.perl</td>
<td>4.5</td>
<td>1.9</td>
<td>784.5</td>
<td>na</td>
</tr>
<tr>
<td>164.gzip</td>
<td>0.6</td>
<td>0.6</td>
<td>3.0</td>
<td>2.0</td>
</tr>
<tr>
<td>175.vpr</td>
<td>0.4</td>
<td>0.4</td>
<td>189.8</td>
<td>1.7</td>
</tr>
<tr>
<td>176.gcc</td>
<td>1.2</td>
<td>1.1</td>
<td>1087.0</td>
<td>299.1</td>
</tr>
<tr>
<td>181.mcf</td>
<td>0.4</td>
<td>0.4</td>
<td>39.5</td>
<td>17.8</td>
</tr>
<tr>
<td>186.crafty</td>
<td>0.8</td>
<td>0.8</td>
<td>15.9</td>
<td>13.9</td>
</tr>
<tr>
<td>197.parser</td>
<td>1.1</td>
<td>1.1</td>
<td>246.5</td>
<td>246.5</td>
</tr>
<tr>
<td>253.perlbmk</td>
<td>145.8</td>
<td>143.5</td>
<td>908.0</td>
<td>906.0</td>
</tr>
<tr>
<td>254.gap</td>
<td>0.8</td>
<td>0.8</td>
<td>502.3</td>
<td>502.3</td>
</tr>
<tr>
<td>255.vortex</td>
<td>1.1</td>
<td>1.1</td>
<td>454.9</td>
<td>254.1</td>
</tr>
<tr>
<td>256.bzip2</td>
<td>0.4</td>
<td>0.4</td>
<td>6.4</td>
<td>6.4</td>
</tr>
<tr>
<td>300.twolf</td>
<td>0.6</td>
<td>0.6</td>
<td>760.3</td>
<td>17.6</td>
</tr>
</tbody>
</table>

**Figure 9.8** Stores-per-object for non-heap variables (NON-HEAP) and heap objects (HEAP) under the context-insensitive analysis (CI) and context-sensitive analysis without (CS) and with heap-object specialization (HS).  

accuracy even for non-heap variables. Figure 9.7(b) indicates that context-sensitive and heap-object specialization lead to a substantial accuracy improvement on the usage of heap objects.

Figures 9.8 and 9.9 show a similar comparison for stores per object between CI and HC. The overall tendency remains almost identical. For non-heap variables, except 132.ijpeg and 176.gcc, context sensitivity and heap-object specialization have little
Figure 9.9 Fractional decrease in the stores per object under CS and HS. (a) for non-heap variables and (b) for heap objects when compared to CI.

impact on accuracy. On the other hand, for heap objects, they lead to a substantial improvement in accuracy.
CHAPTER 10

CONCLUSION

Context sensitivity is becoming increasingly more important in modern programming practices. In pointer analysis, the impact of context sensitivity is increased further when analyzing programs that use an extensive amount of heap-allocated data structures.

In this dissertation, we address the scalability problem of context-sensitive pointer analysis based on Andersen’s inclusion-based approach [1]. The key observation is that procedures without any observable effects on callers, roughly termed procedural side effects, are never involved in the generation of spurious interprocedural dataflow. Therefore, by carefully migrating procedural side effects from callees to callers, we can transform a program into a side-effect free form yet with the same overall pointer information. After such transformation, a single run of the context-insensitive analysis is sufficient in producing accurate pointer information.

In such a case, the scalability of the overall algorithm is determined by the size of procedural summaries propagated during the bottom-up phase. To reduce the size of procedural summaries, we develop a number of simplification algorithms that exploit the following opportunities.
• Much dataflow made in callees has no effect on the dataflow made in callers. Therefore, we can perform garbage collection to eliminate such dataflow from procedural summaries.

• Even within the dataflow visible to callers, there exists much indirection through a sequence of copy assignments. By collapsing such copy assignments, the size of procedural summaries can be further reduced.

• In many cases, the effects of two calls to the same procedure are indistinguishable from the perspective of the flow modeling based on Andersen’s approach. In such cases, by eliminating redundant dataflow made through different calls, the size of procedural summaries can be further reduced.

In overall, the accuracy of the proposed context-sensitive analysis is equivalent to that achieved by exhaustively inlining all procedure calls with the exception for recursive calls. On the other hand, thanks to the simplification algorithms, the scalability of the proposed context-sensitive analysis is comparable to that achieved by context-insensitive analysis.
REFERENCES


AUTHOR’S BIOGRAPHY

Hong-Seok Kim was born in Kwangju, Korea on May 17, 1976. He graduated from Seoul Science High School in February 1994 and received his B.S. degree in February 1998 from the School of Electrical Engineering in Seoul National University. He worked in Microsoft as a software development engineer from September 2000 to September 2001. After finishing his Ph.D, he will go back to Korea and work for Samsung Advanced Institute of Technology.