박 사 학 위 논 문
Doctoral Thesis

업데이트 기록에 기반한
상향방식 포인터 분석

A Bottom-up Pointer Analysis Using the Update History

강 현 구 (姜 炫求 Kang, Hyun-Goo)
전자전산학과 전산학전공
Department of Electrical Engineering and Computer Science
Division of Computer Science

한 국 과 학 기 술 원
Korea Advanced Institute of Science and Technology

2009
A Bottom-up Pointer Analysis Using the Update History
A Bottom-up Pointer Analysis Using the Update History

Advisor : Professor Han, Taisook

by

Kang, Hyun-Goo
Department of Electrical Engineering and Computer Science
Division of Computer Science
Korea Advanced Institute of Science and Technology

A Dissertation submitted to the faculty of the Korea Advanced Institute of Science and Technology in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Electrical Engineering and Computer Science Division of Computer Science

Daejeon, Korea
2008. 11. 28.
Approved by

Professor Han, Taisook
Advisor
업데이트 기록에 기반한
상향방식 포인터 분석

강현구

위 논문은 한국과학기술원 박사학위논문으로 학위논문심사
위원회에서 심사 통과하였음.

2008년 11월 28일

심사위원장 한태숙 (인)
심사위원 최광무 (인)
심사위원 이광근 (인)
심사위원 도경구 (인)
심사위원 김문주 (인)
Abstract

Pointer analysis is an important part for the source code analysis of C programs. Most of pointer analysis algorithms are developed as top-down analyses, where the analyses are performed from callers to callees. Since such a scheme disables the analysis to operate on incomplete programs, existing top-down pointer analyses are not readily applicable to the modular software development process where individual software components are separately developed and subsequently linked with other components. Bottom-up pointer analyses can address this problem by performing analyses from callees to callers. However, analyzing the behavior of a procedure without utilizing any information on callers makes the different pointer behaviors of a procedure for different calls difficult to be traced precisely.

In this thesis, we propose a bottom-up and flow- and context-sensitive pointer analysis algorithm based on a new bottom-up pointer analysis domain named the update history. The update history can not only abstract memory states of a procedure independently of the information on aliases between memory locations, and but also keep information on the order of side effects performed. Such characteristics of the update history not only enable pointer analyses to be formulated as bottom-up analyses, but also help bottom-up pointer analyzers to effectively identify killed side effects and relevant alias contexts, both of which are the main contributions of this thesis. Our bottom-up and flow- and context-sensitive pointer analysis is formulated as an inference algorithm of a type system for the update history. Then we formally prove the soundness of our pointer analysis algorithm, which has not been satisfac-
torily investigated for this kind of problem previously, using the well-developed type soundness machinery from the type system framework.

The experiments performed on a pilot implementation show that our approach is cost-effective in a sense that the precision was improved without sacrificing the performance significantly. A client analysis that detected either uninitialized value or null pointer dereference errors using our pointer analysis was able to identify a maximum of 37% more pointer dereferences as safe than the client analyses using other bottom-up approaches experimented.
Contents

Abstract i

Contents iv

List of Tables vii

List of Figures viii

1 Introduction 1

1.1 Preliminaries ............................................. 1

1.1.1 Pointer Analysis ....................................... 1

1.1.2 Context-sensitive Analysis .......................... 2

1.1.3 Flow-sensitive Analysis ............................... 2

1.1.4 Bottom-up Analysis .................................. 3

1.2 Problems and Goal ....................................... 5

1.3 Contribution ............................................. 7

1.4 Thesis Organization ...................................... 8

2 Overview of our approach 9

2.1 Memory Location Abstraction: Access Path ............ 9

2.2 Memory State Abstraction: Update History ........... 10

2.3 Input Context and Summary ............................ 11

2.4 Strong Update .......................................... 11

2.5 Lazy Partitioning ........................................ 12

2.6 Analysis Example ...................................... 13
3 The Memory Type System 15
3.1 The Language ............................................. 15
3.2 Types ....................................................... 18
3.3 Checking Aliases between Access Paths ..................... 18
3.4 Points-to Set of $M$: Read Operation ....................... 19
3.5 Subtyping: Orders between Types .......................... 21
3.6 Memory Updates ........................................... 24
3.7 Typing Rules for the Program ............................... 25
3.8 Typing Example ............................................ 29
3.9 Correctness of the Type System ............................. 31

4 The Analysis Algorithm 35
4.1 Type Inference without Partitioning ......................... 35
4.2 Type Inference with Lazy Partitioning ....................... 36
4.3 Intraprocedural Analysis Stage: infer_e ...................... 37
4.4 Interprocedural Analysis Stage: infer_p ...................... 39
4.5 Termination and Soundness ................................ 40
4.6 Complexity .................................................. 41

5 Experiments 43
5.1 Implementation .............................................. 43
5.2 Experimental results ........................................ 45

6 Related Work 49
6.1 Bottom-up and Flow-sensitive Pointer Analysis ............... 49
6.2 Bottom-up and Flow-insensitive Pointer Analysis ............ 50
6.3 Top-down Analysis .......................................... 51
   6.3.1 Alias context sensitivity ............................... 51
   6.3.2 Strong update ......................................... 52
6.3.3 Top-down analyses utilizing the reusable summary 52

7 Conclusions 54

A Correctness of the Type System 55
   A.1 Extensions and Assumptions for the Proof 55
   A.2 The Proof 57

Summary (in Korean) 85

References 87
List of Tables

5.1 Experimental results . . . . . . . . . . . . . . . . . . . . . . . . . . . 46
List of Figures

2.1 Analysis example ............................................. 10
3.1 The language .................................................. 15
3.2 Formal semantics ............................................. 17
3.3 Types .......................................................... 18
3.4 Typing rules for input context .............................. 19
3.5 Typing rules for memory reads ............................. 19
3.6 Typing rules for subtype relation (order between memory types) 22
3.7 Orders between types ......................................... 23
3.8 Typing rules for memory updates ......................... 24
3.9 Auxiliary operations ......................................... 26
3.10 Typing rules for program ................................... 27
3.11 Typing example ............................................... 30
4.1 Non-partitioning operations ............................... 35
4.2 Partitioning operations ..................................... 36
4.3 Intraprocedural analysis stage ............................ 38
4.4 Interprocedural analysis stage ............................ 40
5.1 Effects of using different approaches on a client analysis .... 47
1. Introduction

1.1 Preliminaries

Program analysis is the process of automatically analyzing the program behavior either for verifying the correctness of the program or for automatically optimizing the program. In this section, we overview several program analysis techniques that are closely related to this thesis.

1.1.1 Pointer Analysis

Pointer analysis is essential for the effective source code analysis of C programs. Assume that we are designing a program analyzer which can prove that following C program has no runtime error at line l4.

```c
int x, y;
dec(int *p) {
  l1: *p = *p - 1;
}
main() {
  l2: x = 1; y = 1;
  l3: dec(&x);
  l4: y = x / y;
  l5: dec(&y);
}
```

If the analyzer does not trace the pointer values of the program, the assignment at line l1 should be safely considered as a potential update to every variable since the analyzer do not know to which memory location variable p points at line l1.
Therefore, the procedure call at line \( l_3 \) should be safely considered as changing either the value of \( x \) or the value of \( y \) to 0, and thus the line \( l_4 \) will be identified as the potential runtime error (division by zero). On the other hand, if the analyzer knows that the procedure call at line \( l_3 \) updates only the variable \( x \) by tracing the pointer values precisely, safe execution of line \( l_4 \) can be proved. Therefore, a precise program analyzer that can statically prove this program has no runtime error needs to trace the pointer values (pointer analysis).

1.1.2 Context-sensitive Analysis

Context-sensitive analyses distinguish the behavior of a procedure on each procedure call whereas context-insensitive analyses do not. Consider the example in Section 1.1.1 again. A context-insensitive analyzer considers the assignment at line \( l_1 \) as a potential update to either \( x \) or \( y \) because it does not distinguish the behaviors of procedure \( \text{dec} \) for different calls at line \( l_3 \) and \( l_5 \). Thus, the line \( l_4 \) will be identified as a potential error for the similar reason described in Section 1.1.1. Therefore, a precise analyzer that can prove this program has no runtime error needs to distinguish behaviors of procedure calls at line \( l_3 \) and \( l_5 \) (context-sensitive). In general, a cost/precision trade-off exists: the context-sensitive analyses are more precise, but less efficient than context-insensitive analyses.

1.1.3 Flow-sensitive Analysis

Flow-sensitive analyses take the statement ordering into account and thus compute analysis results at each program point. To the contrary, flow-insensitive analyses do not consider the statement ordering and usually compute one analysis result for a whole program. Consider the example in Section 1.1.1 again. A flow-insensitive analyzer considers the value of \( y \) at the right-hand side of line \( l_4 \) as either 1 or 0 because the variable \( y \) is respectively updated at lines \( l_2 \) and \( l_4 \) with the integer
value of 1 and 0, and the analyzer does not take the statement ordering into account. Thus, the line \( l4 \) will be identified as a potential runtime error. Therefore, a precise analyzer that can prove this program has no runtime error needs to know that reading the value of \( y \) at line \( l4 \) is not relevant to updating \( y \) at line \( l4 \) (flow-sensitive). In general, a cost/precision trade-off exists: the flow-sensitive analyses are more precise, but less efficient than flow-insensitive analyses.

### 1.1.4 Bottom-up Analysis

Most of program analyses are developed as top-down analyses that perform analyses from callers to callees. To the contrary, bottom-up program analyses perform analyses from callee modules to caller modules, where a module in a C program is a single procedure or a set of mutually recursive procedures. A bottom-up analyzer computes the summary transfer function for each procedure without utilizing any information on caller modules. For each call site where a procedure is invoked, such an analyzer approximates the behavior of the procedure call, not by analyzing source code of the callee, but by applying the summary transfer function of the callee that has been separately computed in advance.

In practice, a bottom-up analysis has several advantages. First, it enables the analysis to be performed on incomplete programs such as library code, by exploiting its ability to analyze and summarize behaviors of procedures without callers. Thus, the analysis can be applied to the modular development process of software in which individual software components are separately developed and subsequently linked with other components. Second, the analysis can often be scaled to large programs because the whole-program and results of analysis need not to be in memory simultaneously during the analysis, and the reuse of summaries of modules dispenses with the need to reanalyze the procedure body at every call site. Third, when there is a code change, reusing the summaries of modules that are irrelevant to this mod-
ification means that only the dependent parts of the code need to be reanalyzed.

Unfortunately, it is not trivial to design a precise bottom-up analysis since the summary construction algorithm in a bottom-up analysis should operate without utilizing any information on callers (calling contexts). Consider the example in Section 1.1.1. A top-down analyzer can analyze the behavior of procedure call at line 13 with the information such that the memory location to which the formal parameter p initially points is the address of x, and the value stored at the variable x is 1 for this call. Thus, the behavior of assignment l1 for the procedure call at line 13 can be trivially identified as updating the variable x with the value 0. However, the bottom-up analyzer which attempts to summarize the behavior of procedure dec before analyzing procedure main cannot trivially identify the memory location updated by the assignment l1 and the value stored at this memory location.

Such a problem is usually addressed by parameterizing the analysis, which is briefly demonstrated as follows. When the summary of a procedure is computed, symbolic names are introduced for the unknown values that are decided at the procedure call (e.g., assume that symbolic names V1 and V2 are introduced to denote the initial values of expressions p and *p at the start of procedure dec, respectively). Then, when the caller procedure is analyzed using the summaries of callee procedures computed in advance, the symbolic names contained in the summary are resolved to the value of actual parameters (e.g., V1 and V2 are resolved to the address of x and the integer value 1 for the procedure call at line 13, respectively). In this way, the bottom-up analyzer can identify different behaviors of a procedure for different calls. However, the parameterized analysis domain makes the abstract transfer functions of the analysis (the summary construction algorithm) necessary to be parameterized as well, which often causes over-approximations in bottom-up analyses. For example, consider the value of the right-hand side of assignment l1. The summary construction algorithm should determine the value of the expression *p.
- 1 with a parameterized value $V_2$ (expression $*p$ here is computed as $V_2$). Therefore, a precise bottom-up analysis, which attempts not to introduce over-approximation here, may need to introduce a new parameterized representation of this value such as $V_2 ⊕ 1$, and to perform the delayed subtraction ($⊕$) when the real value of $V_2$ is determined at the call site as in line l3. However, it is not trivial to design a bottom-up analysis that can deal with every behavior of a procedure in such a symbolic way without introducing over-approximations that are not necessary in the top-down analysis. For this reason, most of existing researches focusing on improving precision of the program analysis were proposed for top-down analyses, and top-down analyses are usually more precise than their counterpart bottom-up analyses.

### 1.2 Problems and Goal

The goal of this thesis is to design and implement an effective bottom-up pointer analysis that can be used in a modular software development environment. Although several bottom-up approaches for analyzing the pointer behaviors of programs [10, 3, 11, 28, 14, 26, 13, 27, 2, 31] have been proposed, several issues remain to be addressed; our thesis focus on two of these issues.

Consider the following example code, which gives fundamental insight into these issues:

```c
int g1, g2;
f(int **p, int **q) {
    l1: *p=&g1;
    l2: *q=&g2;
    l3: return *p;
}
main() {
    int *i, *j;
```
Assume that each memory location represented by expressions &p, &q, &g1, &g2, p, q at the start of procedure f are designated (symbolically named) as p, q, g1, g2, p., and q. respectively.

**Strong update.** The assignment l2 kills the effect of assignment l1 if both p and q initially point to the same memory location, as in the cases of procedure calls at l4 and l5. In this case, the analyzer can safely discard the effect of assignment l1, which is called a *strong update* [1]. Otherwise, the analyzer needs to keep the side effect of l1 (e.g., the procedure call at l6). Therefore, a precise bottom-up analysis that distinguishes these two cases needs to effectively identify actually killed side effects, which generally requires a flow- and context-sensitive analysis. However, the existing bottom-up pointer analyses [13, 27, 2, 31] that are flow- and context-sensitive cannot identify interprocedurally killed side effects of this type.

**Alias context sensitivity.** The behavior of pointers in a procedure generally depends on the alias context, which is the set of alias relations between unknown locations\(^1\) that can be determined from the calling context. For example, procedure f returns g2 if it is invoked with a calling context such that p. and q. are identical (e.g., call sites l4 and l5), whereas it returns g1 if p. and q. are different (e.g., call site l6). Therefore, a precise bottom-up analysis that distinguishes these two cases must be alias context sensitive, and the summary transfer function should reflect this input–output dependency. Since computing a full summary for all possible alias contexts of a procedure can be prohibitively expensive and some alias relations do not affect the output of a procedure, the alias-context-sensitive summary-construction

---

\(^1\)We call the memory locations that are decided at the procedure invocation, such as p. and q., unknown locations.
algorithm needs to identify only relevant alias contexts\textsuperscript{2} that change the behavior of a procedure.

### 1.3 Contribution

Our approach is based on the insight that both the killed side effects and relevant alias contexts discussed in Section 1.2 can be effectively identified if we have information on the order of side effects performed. For example, if the summary of procedure $f$ contains information that memory location $p.\ast$ is updated before $q.\ast$ is updated, we can kill the side effect of $p.\ast$ when the analyzer becomes aware that $p.\ast$ and $q.\ast$ are resolved to the same memory location, as in $l4$. Based on this idea, we introduce a new memory representation that keeps information on the order of side effects performed, which is named the update history. Our bottom-up and flow-and context-sensitive pointer analysis is formulated based on this memory representation, which is used to identify killed side effects and relevant alias contexts. We formally prove the soundness of the analysis by formalizing the analysis as an inference algorithm of a type system, which is a typical framework used to design a sound and comprehensive bottom-up program analysis.

We summarize our contributions as follows:

1. We propose a bottom-up pointer analysis that can effectively identify killed side effects and relevant contexts using a new memory representation called the update history. The update history can abstract the set of memory states independently of the information on aliases between memory locations, and keep the information on the order of side effects performed.

\textsuperscript{2}Even though considering only relevant alias contexts during the analysis removes the overhead of computing meaningless summaries, a procedure can involve an exponential number of relevant alias contexts. In this thesis, we focus on a method for effectively identifying relevant alias contexts, where the number of relevant alias contexts considered in the analysis is subject to a constant bound $k$. 

2. We formalize the analysis using the type system framework, and provide a formal proof of the soundness which has not been satisfactorily investigated for this kind of problem previously.

3. We illustrate the effectiveness of our approach with empirical results obtained from experiments performed on a pilot implementation of the method.

1.4 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 gives an informal overview of our approach, Chapter 3 presents the memory type system that computes the side effects of a program, and Chapter 4 presents the bottom-up pointer analysis algorithm that is formulated as an inference algorithm for the memory type system. Chapter 5 discusses our implementation and shows the experimental results therefrom. We relate our work to previous research in Chapter 6, and draw conclusions in Chapter 7.
2. Overview of our approach

2.1 Memory Location Abstraction: Access Path

We name memory locations using the notion of the *symbolic access path* [7], which makes the alias-context-independent naming of memory locations possible. An access path $AP$ of a procedure $f$ represents a set of addresses reachable by the access method at the start of $f$, which has the following form:

$$s \in \text{Selector} ::= \text{fld} | *$$

$$AP \in \text{AccessPath} ::= x | l | AP.s$$

Access path $x$ represents the address of variable $x$, access path $l$ represents the set of addresses dynamically allocated at program point $l$, access path $AP.fld$ represents the address of field $fld$ of access path $AP$, and access path $AP.*$ represents the address to which access path $AP$ initially points at the start of a function. If access path $AP$ has at least one dereference ($.* \in AP$), we call it an unknown access path; otherwise, we call it a known access path.

In the presence of recursive data structures, the number of possible access paths for a procedure may be unbounded. For such recursive access paths, we follow the well-known [2, 3] abstraction technique that limits the depth of the recursive access for a variable to some constant $k$. For example, a set of memory locations represented by expressions $x->\text{next}, x->\text{next}->\text{next}, \ldots$ at the start of a procedure are abstracted into a single abstract access path $x.*.\text{next}$, when the field $\text{next}$ means the recursive access, and $k = 1$.

When $k = 1$, we say that the access path $x.*.\text{next}$ has collapsed since it represents several memory locations at run-time, while a unique access path (e.g., $p.*$)
This represents the memory state after the sequence of symbolic updates, abstract memory $M$.

2.2 Memory State Abstraction: Update History

An abstract memory $M$ of a procedure $f$ has the following form:

$$[AP_1 \mapsto V_1] \cdots [AP_n \mapsto V_n]$$

This represents the memory state after the sequence of symbolic updates, $AP_i$ to $V_i$, are performed with the initial memory state of procedure $f$, where value $V_i$ represents the set of access paths. For example, the memory $[p. \mapsto \{i\}][q. \mapsto \{j\}][r. \mapsto \{k\}]$ computed on line 4 of Figure 2.1 abstracts the memory state after update operations $p. \mapsto \{i\}$, $q. \mapsto \{j\}$, and $r. \mapsto \{k\}$ are successively performed with the initial
memory state of procedure \( f \).

Note that the key problem of using such a memory representation as an analysis domain is that the order \( (\subseteq_M) \) and join operation \( (\sqcup_M) \) are not trivially defined. To simplify the overview of this chapter, we postpone the presentation of our approach to solve this problem until Section 3.5.

### 2.3 Input Context and Summary

An input context (alias context) \( B \) is defined as a set of alias relations, where each alias relation \( AP \# AP' \) means that there is no intersection (alias) between the memory locations represented by \( AP \) and \( AP' \). \( B \) abstracts a set of calling contexts by constraining the alias status between access paths. For example, an input context \( \emptyset \) represents all calling contexts since there is no constraint on the alias status. An input context \( \{(q.* \# r.*)\} \) of a procedure \( f \) represents a set of calling contexts such that there is no intersection between the memory locations represented by expressions \( q \) and \( r \) at the start of procedure \( f \).

The summary computed for each control flow graph (CFG) node \( n \) of a procedure \( f \) are defined as a set of inout elements, where each inout \( (B, M) \) means that \( M \) is a safe approximation of the memory state obtained after \( n \) is executed with a calling context abstracted by input context \( B \). For example, \([p.* \mapsto \{i\}][q.* \mapsto \{j\}][r.* \mapsto \{k\}]\) computed on line 4 of Figure 2.1 abstracts every memory state that is obtainable after line 4 has been executed, since \( B = \emptyset \) means that \( B \) abstracts all calling contexts.

### 2.4 Strong Update

Consider an update operation that updates the set of access paths \( A \) to the value \( V \) with the memory \( M \) and the input context \( B \), which is caused either by an
assignment statement or a procedure invocation. If $A$ is a singleton set \{AP\} and $AP$ is a unique access path, we can safely kill the old side effect on $AP$. In other words, we append a new side effect $[AP \mapsto V]$ to $M$ while deleting the old side effect $[AP \mapsto V'] \in M$. Note that we kill only the side effect for $AP$ from $M$ and do not kill the side effect for an unknown access path $AP'$ in $M$, which can be identical to $AP$ depending on the input context. The analyzer does not attempt to eagerly kill those side effects, instead postponing the decision to kill them until the procedure is invoked by simply keeping them in the summary of the procedure. If the procedure is invoked and both $AP$ and $AP'$ are resolved as a unique access path $AP''$, the analyzer kills those intermediate side effects lazily using the information on the order of updates that is kept in the summary.

### 2.5 Lazy Partitioning

Our analyzer lazily introduces a new alias relation into the analysis only when it is determined to be relevant. A relevant alias relation differentiates the meaning of a procedure $f$ since the points-to set computed inside the procedure is changed by this alias relation. This dependence arises only when all of following three conditions are satisfied:

1. Either $AP$ or $AP'$ represents an unknown location.

2. $AP$ is read after $AP'$ is updated.

3. $AP$ is not updated after $AP'$ is updated.

Such conditions can be explicitly identified by our memory representation since it keeps the information on the order of side effects performed. For example, if there is a read operation on an access path $AP_i$ with the memory $[AP_1 \mapsto V_1] \cdots [AP_n \mapsto V_n]$, alias relations between $AP_i$ and \{AP$_{i+1}$, $\cdots$, $AP_n$\} are considered in the analysis.
Note that alias relations between $AP_i$ and access paths $\{AP_1, \cdots, AP_{i-1}\}$ are not relevant because they do not satisfy condition 3. Alias relations between $AP_i$ and access paths that are not updated yet and will be updated later inside the procedure are also not relevant because they do not satisfy condition 2.

### 2.6 Analysis Example

The example in Figure 2.1 shows the analysis result at each program point computed in the bottom-up manner using our pointer analysis algorithm. The analyzer first computes the summary of callee procedure $f$ with no particular calling context. Statements 2, 3, and 4 simply append side effects $[p.\ast \mapsto \{i\}]$, $[q.\ast \mapsto \{j\}]$, and $[r.\ast \mapsto \{k\}]$ to the initial memory $\epsilon$ successively. This means postponing the decision about what side effect of the old update history is killed by an update operation if it depends on the calling context. As mentioned in Section 2.5, the expression $*q$ on line 5 may be interpreted differently depending on the input context of procedure $f$. So, the analyzer introduces a new input context $\{q.\ast \#r.\ast\}$ and computes the corresponding summary for this input context as well. The side effect for $p.\ast$ in the old update history can be killed since the $l$-value of this assignment is a singleton set $\{p.\ast\}$, and $p.\ast$ is a unique access path. Finally, procedure $f$ is summarized with the inout elements computed on line 5, which is the return point of $f$.

Next, the analyzer computes the summary of caller procedure $g$ using the summary of procedure $f$ (which was computed separately). Unknown access paths $p.\ast$, $q.\ast$, and $r.\ast$ in the summary for procedure $f$ are resolved as $x.\ast$, $y.\ast$, and $y.\ast$, respectively, at the procedure call on line 7. Among two input contexts of the summary of procedure $f$, only $\emptyset$ is satisfiable for this calling context. So, the analyzer applies corresponding side effects $[q.\ast \mapsto \{j\}] [r.\ast \mapsto \{k\}] [p.\ast \mapsto \{j, k\}]$ for this particular calling context with initial memory $\epsilon$ of procedure $g$. The underlining in the figure shows the sequence of updates performed for this application. The first line resolves
the access path, in which access paths of the callee’s summary are substituted with those of the caller’s summary. The $\leadsto$ command on the second line shows the sequence of update operations performed. This can be interpreted as performing the sequence of updates in the context of caller, which are postponed when procedure $f$ is summarized. Note that such a lazy update scheme increases the opportunity of strong updates in the bottom-up analysis. For example, we can safely kill side effect $[q.\star \mapsto \{j\}]$ since the successive side effect $[r.\star \mapsto \{k\}]$ is determined to be a side effect on the same unique access path $y.\star$ under this particular calling context on line 7. Note that performing this strong update when we summarized procedure $f$ is unsafe because the $B$ does not guarantee that $q.\star$ and $r.\star$ are identical. On line 8, $p.\star$, $q.\star$, and $r.\star$ are resolved as $x.\star$, $y.\star$, and $z.\star$, respectively. In this case, both $\{y.\star \# z.\star\}$ and $\emptyset$ are satisfiable depending on the alias context of procedure $g$. So, the analyzer introduces a new input context $\{y.\star \# z.\star\}$ and computes the corresponding summary for both input contexts.
3. The Memory Type System

A type system [23] can be regarded as an equation that specifies the type that safely approximates the execution result of a program. In this chapter, we formalize the update-history-based bottom-up pointer analysis that was informally presented in Chapter 2 as a type system for the memory. We first formalize our update-history-based memory representation as a typical type element (modular analysis domain) of the subtype system. Then we formalize the subtype system\(^1\) for the memory type and prove the soundness of our memory type system. The target bottom-up pointer analysis is formulated as an inference algorithm of this memory type system in Chapter 4.

3.1 The Language

\[
p \in \text{program} ::= \text{fundec} ; e
\]
\[
\text{fundec} \in \text{fundec} ::= \text{fun } f(\overline{x}) = e
\]
\[
e \in \text{expression} ::= \& x | \& (e.\text{fld}) | \& (e.e) | e.\text{fld} | \text{new}_l |
\]
\[
e_1 := e_2 | f \overline{y} | e_1 ; e_2 | \text{if}(e_1, e_2)
\]
\[
x, \text{fld}, f \in \text{Identifier}
\]

Figure 3.1: The language

We will use the following notation: Let \( X \) denote an arbitrary syntactic element

\(^1\)Our type system is a nonstandard subtype system in the following senses: (1) it does not perform any checking (verification), instead simply specifies the safe approximation of a program’s execution result as a type element \( \tau \); and hence, we do not need the property “progress” [34] in our type system; and (2) the type of a procedure is a set of input-output pairs, each of which is a sound type of the procedure.
in our type system, with \( X \) used to abbreviate various syntactic enumerations (e.g., \((X, Y)\) abbreviates \((X_1, Y_1), \ldots, (X_n, Y_n)\)).

The language used to model the pointer behaviors of the C program is defined in Figure 3.1. A program \( p \) is a sequence of procedure declarations followed by expression \( e \) which represents the body of the main procedure. Assignment statements such as \( x=y \) and \( s->f=*y \) are represented as \( &x := *(&y) \) and \( &(((*(&s)).f) := *(&y)) \) respectively. Statements such as \( x[i]=&y \) and \( *(x+j)=&y \) are translated into one representative statement \( *(&x) := &y \), where the access path for the variable \( x \) is safely regarded as the collapsed access path in the analysis. The condition of an \( if \) statement is disregarded because we are interested in a path-insensitive analysis. The loop is explained by recursive call. To simplify the presentation we exclude the type casting and function pointer in the language. In our implementation, these features are correctly handled using the same method as [3], which is a bottom-up and flow-insensitive pointer analysis.

The formal semantics of our language is defined in Figure 3.2. Since the goal of the pointer analysis is not to verify any property but rather to conservatively approximate the correct execution result of a program, we defined the formal semantics to ignore all the stuck-execution paths by looking at them as \( \perp \) and to collect the results of all the execution paths that are probably correct. For this reason, we regard the initial value of each uninitialized variable as \( \perp \), under the assumption that this results in a stuck configuration. Note that the presented formal semantics is an abstract semantics, since there are some abstract components such as the collapsed access path. For example, the address \( x \) for the array typed variable \( x \) is treated as the collapsed access path, and weakly-updated. We prove the correctness of our memory type system in the context of proving that typing result of the memory type system safely approximates the execution result of this abstract transition semantics. This approach is safe as long as the formal semantics safely models the pointer
We assume a global environment $\gamma$ for $p$ is given where $\gamma(f_i) = (\overline{x}, e_i)$ and fundec_i = fun(f_i(x) = e_i)

\[
\begin{align*}
\text{[ Domains ]} \\
ap & \in \text{address} ::= x | l | ap.fld \\
a, v & \in \text{value} = 2^{ap} \\
clos & \in \text{closure} = Id^* \times \text{expression} \\
\gamma & \in \text{environment} = Id^* \rightarrow \text{closure} \\
mem & \in \text{memory} = \text{address} \rightarrow \text{value} \\
m & \in \text{memories} = 2^{mem} \\
K & \in \text{continuation} ::= \epsilon | e.K | \text{ret}(\overline{x}).K | \text{deref}.K | \text{field}(\overline{x}).K \\
& | \text{assign}(e).K | \text{assign}(v).K \\
& | \text{jointo}(e, m).K | \text{join}(v, m).K \\
C & \in \text{configuration} ::= (p) | (m, e, K) | (m, v, K)
\end{align*}
\]

\[
\begin{align*}
\text{[ Operations ]} \\
\text{mem}(ap) & = \text{if } ap \in \text{Dom}(mem) \text{ then mem}(ap) \text{ else } \emptyset \\
m(\{ap_1, \cdots, ap_n\}) & = \bigcup_{ap \in m} \text{mem}(ap_i) \quad m(\{\}) = \emptyset \\
\perp \downarrow m(v) = \downarrow \\
\text{update-mem}(m, \emptyset, v) & = \\
\text{if unique}(ap) \text{ then } \{\text{mem}[ap \mapsto v] | \text{mem} \in m\} \\
\text{else } \{\text{mem}[ap \mapsto v \cup \text{mem}(ap)] | \text{mem} \in m\} \\
\text{update-mem}(m, \{ap_1, \cdots, ap_n\}, v) & = \\
\{\text{mem}[ap_1 \mapsto v \cup \text{mem}(ap_1)] \cdots [ap_n \mapsto v \cup \text{mem}(ap_n)] | \text{mem} \in m\}
\end{align*}
\]

\[
\begin{align*}
\text{[ Transition Rules ]} \\
\text{We assume a global environment } \gamma \text{ for } p \text{ is given where } \gamma(f_i) = (\overline{x}, e_i) \text{ and fundec}_i = \\
\text{fun}(f_i(x) = e_i)
\end{align*}
\]
related behavior of the C language.

3.2 Types

\[
\begin{align*}
A, V & \in \text{Value} \quad ::= \{AP_1, \cdots, AP_n\} \\
M & \in \text{Memory} \quad ::= \epsilon \mid M[AP \mapsto V] \\
as & \in \text{Assumption} \quad ::= AP\#AP \\
B & \in \text{InputContext} \quad ::= \{as_1, \cdots, as_n\} \\
P & \in \text{Inout} \quad ::= (B, (M, V)) \\
D & \in \text{Summary} \quad = 2^P \\
\Gamma & \in \text{TypeEnv} \quad = \text{Identifier} \mapsto \text{Summary}
\end{align*}
\]

Figure 3.3: Types

The types (analysis domains) in our memory type system are defined in Figure 3.3. The Value type \(A\) (or \(V\)) is a set of access paths. The meanings of the Memory type \(M\), the InputContext type \(B\), the Inout type \(P\), and the Summary type \(D\) are explained in Chapter 2. The TypeEnv type \(\Gamma\) is a finite mapping of procedure name to procedure summary.

3.3 Checking Aliases between Access Paths

Judgements \(B \vdash_{as} as\) and \(B \vdash_B B'\) check the disjointness between access paths under the given input context \(B\). Deduction rules for these judgements are given in Figure 3.4. An alias relation \(as\) is satisfied under \(B\) if it is implied by \(B\) (rule [hypoth]) or \(as\) is known inside the procedure (rule [\(AP\#AP\)]). For example, the disjointness between the access paths \(x.*\) and \(y.*\) of procedure \(f\) can be proved only when \(x.*\#y.*\) is in the given input context \(B\), since both of \(x.*\) and \(y.*\) are unknown

---

\(^2\)The extension of the type system for the other alias relations such as “must be aliased” is direct, and is not considered in this thesis for brevity.

\(^3\)The definition of \(P\) is extended to a tuple \((B, (M, V))\), where value \(V\) represents a return value.
$$\frac{\text{as} \in B}{B \vdash_{\text{as}} \text{as}}$$

$$\frac{\text{known}(\text{AP}_1) \quad \text{known}(\text{AP}_2) \quad \text{AP}_1 \neq \text{AP}_2}{B \vdash_{\text{as}} \text{AP}_1 \neq \text{AP}_2}$$

$$\frac{\forall \text{as} \in B': B \vdash_{\text{as}} \text{as}}{B \vdash_{B'} B'}$$

Figure 3.4: Typing rules for input context

access paths. If $x.* \# y.* \notin B$, this means that memory locations abstracted by $x.*$ and $y.*$ are not disjoint (may be aliased).

### 3.4 Points-to Set of $M$: Read Operation

<table>
<thead>
<tr>
<th>Rule</th>
<th>Judgement</th>
</tr>
</thead>
<tbody>
<tr>
<td>[read-$\epsilon$]</td>
<td>$B, \epsilon \vdash_{\text{AP}} \text{AP} \Rightarrow {\alpha(\text{AP}.*})}$</td>
</tr>
<tr>
<td>[read-=$=$]</td>
<td>$B, M[\text{AP} \mapsto V_1] \vdash_{\text{AP}} \text{AP} \Rightarrow V_1$</td>
</tr>
<tr>
<td>[read-$#!]$</td>
<td>$B \vdash \text{AP} \neq \text{AP}<em>1 \quad B, M \vdash</em>{\text{AP}} \text{AP} \Rightarrow V_2$</td>
</tr>
<tr>
<td>[read-safe]</td>
<td>$B, M \vdash_{\text{AP}} \text{AP} \Rightarrow V_2$</td>
</tr>
<tr>
<td>[read-$A$]</td>
<td>$B, M \vdash_{\text{AP}} \text{AP}<em>1 \Rightarrow V_1 \quad \cdots \quad B, M \vdash</em>{\text{AP}} \text{AP}_n \Rightarrow V_n$</td>
</tr>
</tbody>
</table>

$$\frac{B, M \vdash_{\text{AP}} \text{AP}_1 \Rightarrow V_1 \quad \cdots \quad B, M \vdash_{\text{AP}} \text{AP}_n \Rightarrow V_n}{B, M \vdash_r \{\text{AP}_1, \cdots, \text{AP}_n\} \Rightarrow V_1 \cup \cdots \cup V_n}$$

Figure 3.5: Typing rules for memory reads

The judgement $B, M \vdash_{\text{AP}} \text{AP} \Rightarrow V$ specifies the safe approximation of pointed access paths $V$ of $\text{AP}$ with memory $M$ under input context $B$. In other words, it specifies the conditions for the safe read operation that computes the set of access
paths \( V \) to which access path \( AP \) points with \( M \) under \( B \). Typing rules for this judgement are given in Figure 3.5.

When there is a read operation on \( AP \), the value of \( AP \) is safely approximated by examining the given memory from the most-recently updated entry to the first updated entry.

If the update history of the examined memory is \( \epsilon \) (rule \([\text{read-}\epsilon]\)), this indicates that we are reading the value to which \( AP \) initially points. This value is represented as abstract access path \( \alpha(\ast) \), where \( \alpha \in AP \rightarrow AP \) is a well-known [2, 3] access path abstraction function that maps the concrete access path of each procedure to the abstract access path by limiting the depth of the recursive access to some constant \( k \). In general, it is safe to use any kind of abstraction (grouping) for the access paths as long as the uniqueness of each abstract access path that is implied by the used abstraction function is safely dealt with. A predicate “\( \text{collapsed}(\alpha, AP) \)” is satisfied if one of the following conditions is satisfied: (1) \( l \in \text{prefix}(AP) \), where \( \text{prefix} \in AP \rightarrow 2^{AP} \) is a function that computes the set of prefix access paths of the given access path \( AP \); (2) \( \exists AP' \in \text{prefix}(AP) : AP' \) is used in pointer arithmetic or array indexing; or (3) \( \exists AP_1, AP_2 \in \text{Dom}(\alpha) : AP_1 \neq AP_2 \land \alpha(AP_1) = \alpha(AP_2) = AP \). Otherwise, \( AP \) is not a collapsed access path, which is denoted as “\( \text{unique}(\alpha, AP) \)”.

There are two cases if the examined current update history is \( M[AP \mapsto V_1] \): (1) if \( AP \) is a unique access path, then the value of \( AP \) is \( V_1 \), since \([AP \mapsto V_1] \) must update the \( AP \) (rule \([\text{read-}\mapsto]\)); and (2) if \( AP \) is a collapsed access path, then we conservatively join value \( V_1 \) to previous value \( V_2 \) of \( AP \) in \( M \) since \([AP \mapsto V_1] \) may update the \( AP \) (rule \([\text{read-safe}]\)).

There are also two cases if the examined current update history is \( M[AP_1 \mapsto V_1] \): (1) if \( AP_1 \) is disjoint with \( AP \), then we read the value of \( AP \) in memory \( M \) since \([AP_1 \mapsto V_1] \) does not affect the value of \( AP \) (rule \([\text{read-}#]\)); and (2) if \( AP_1 \) is not
disjoint with \( AP \) (it is possibly aliased), then we conservatively join value \( V_1 \) to previous value \( V_2 \) of \( AP \) in \( M \) since the last update history \([AP_1 \mapsto V_1]\) may update the location represented by \( AP \) (rule [read-safe]\(^4\)).

Now, judgement \( B, M \vdash A \Rightarrow V \) is added to the type system that specifies the safe approximation of pointed access paths \( V \) of the set of access paths \( A \) with \( M \) under \( B \). Typing rule [read-A] for this judgement in Figure 3.5 computes the pointed access paths of \( \{AP_1, \cdots, AP_n\} \) by joining the pointed access paths \( V_i \) of each access path \( AP_i \) with \( M \) under \( B \).

### 3.5 Subtyping: Orders between Types

The key problem of keeping the information on the order of side effects performed in the memory type is that the order between memory types (subtype relation \( M_1 \sqsubseteq_M M_2 \)) is not trivially defined since \( M_1 \) and \( M_2 \) can have a different order of side effects. In order to handle such cases, we define the subtype relation between memories in the context of safely rearranging the order of side effect in \( M_1 \) and \( M_2 \) into the same order.

Typing rules for the subtype relation between memories, which generally depends on the input context \( B \), are given in Figure 3.6. If the orders of side effects of \( M_1 \) and \( M_2 \) are same, the subtype relation is determined by the rule \( [\sqsubseteq_M (3)] \), which inductively compares two memories with the same order of updates. If the orders of side effects of \( M_1 \) and \( M_2 \) are different and the domain of the last side effect of \( M_2 \) is \( AP \) (rule \( [\sqsubseteq_M (2)] \)), \( M_3 \sqsubseteq_M M_2 \) determines \( M_1 \sqsubseteq_M M_2 \), where the domain of the last side effect of \( M_3 \) is \( AP \) and the set of memory states abstracted by \( M_3 \) contains those of the \( M_1 \). The shift(\( B, M_1, AP \)) operation of Figure 3.6 is devised to find such an \( M_3 \) by moving the side effect on \( AP \) in \( M_1 \) to the rightmost position.

\(^4\)The ambiguity between rules [read-#] and [read-safe] is not problematic in terms of the type system. The trivial problem of making the rules deterministic is addressed in Chapter 4.
[ shift ]

\[
\text{shift}(B, \epsilon, AP) = [AP \mapsto \{ \alpha(AP, \epsilon) \}]
\]
\[
\text{shift}(B, M[AP \mapsto V], AP) = M[AP \mapsto V]
\]
\[
\text{shift}(B, M[AP' \mapsto V'], AP) =
\]
\\
let shift(B, M, AP) = M'[AP \mapsto V'] in
\\
if B \vdash AP \# AP' \lor \text{collapsed}(\alpha, AP) then M'[AP' \mapsto V'][AP \mapsto V]
\\
else M'[AP' \mapsto V'][AP \mapsto V \cup V']
\\
[ Subtype relation for the memory type ]

\[
\text{[} \sqsubseteq_M \text{(1)} \text{]} \quad B \vdash M \sqsubseteq_M \text{shift}(B, M, AP)
\]
\[
\text{[} \sqsubseteq_M \text{(2)} \text{]} \quad B \vdash \text{shift}(B, M_1, AP) \sqsubseteq_M M_2[AP \mapsto V]
\]
\[
\text{[} \sqsubseteq_M \text{(3)} \text{]} \quad B \vdash M \sqsubseteq_M M' \quad V \subseteq V'
\]
\[
\text{[} \sqsubseteq_M \text{(4)} \text{]} \quad B \vdash \epsilon \sqsubseteq_M \epsilon \Rightarrow \epsilon
\]

[ Join operation for the memory type ]

\[
\text{[} \sqcup_M \text{(1)} \text{]} \quad B \vdash M_1 \sqcup_M M_2 \Rightarrow M_3
\]
\[
\text{[} \sqcup_M \text{(2)} \text{]} \quad B \vdash M_1[AP \mapsto V] \sqcup_M M_2[AP \mapsto V'] \Rightarrow M_3[AP \mapsto V \cup V']
\]
\[
\text{[} \sqcup_M \text{(3)} \text{]} \quad B \vdash \text{shift}(B, M_1, AP) \sqcup_M M_2[AP \mapsto V] \Rightarrow M_3
\]
\[
\text{[} \sqcup_M \text{(4)} \text{]} \quad B \vdash \epsilon \sqcup_M \epsilon \Rightarrow \epsilon
\]

Figure 3.6: Typing rules for subtype relation (order between memory types)

such that the set of memory states abstracted by \( \text{shift}(B, M_1, AP) \) always contains those of \( M_1 \) under input context \( B \). In other words, \( \text{shift}(B, M_1, AP) \) computes an \( M_3 \) such that the last update history of \( M_3 \) is \( [AP \mapsto V'] \), and for every abstract access path \( AP_i \) of procedure \( f \), if \( B \vdash AP \Rightarrow V_i \) and \( B \vdash \text{shift}(B, M_1, AP) \Rightarrow V_i' \),
then $V_i \subseteq V'_i$.

For example, let us consider the subtype relation between two memories $M_1 = [x.\ast \mapsto \{i\}] [y.\ast \mapsto \{j\}]$ and $M_2 = [y.\ast \mapsto \{j\}] [x.\ast \mapsto \{i\}]$. If the input context $B$ is $\{x.\ast \#y.\ast\}$, then $B \vdash M_1 \sqsubseteq M_2$ is satisfied since $\text{shift}(B, M_1, x.\ast) = [y.\ast \mapsto \{j\}] [x.\ast \mapsto \{i\}]$ and $B \vdash \text{shift}(B, M_1, x.\ast) \sqsubseteq M_2$ is satisfied by the rule $[\sqsubseteq_M (3)]$. If the input context $B$ is $\emptyset$ and $\text{unique}(\alpha, x.\ast)$, then $\text{shift}(B, M_1, x.\ast) = [y.\ast \mapsto \{j\}] [x.\ast \mapsto \{i, j\}]$ since $y.\ast$ and $x.\ast$ may be aliased. In this case, we cannot conclude $B \vdash M_1 \sqsubseteq M_2$.

The typing rules for the memory join operation ($\sqcup_M$) are given in Figure 3.6 and defined in a similar way to the order between memories ($\sqsubseteq_M$) explained above. Note that even though we declaratively (non-deterministic for the rules $[\sqcup_M (2)]$ and $[\sqcup_M (3)]$) presented the memory join operation for generality, a deterministic version of $\sqcup_M$ for the fixpoint iteration in the inference algorithm of Chapter 4 is trivially derived from these rules. For example, rule $[\sqcup_M (3)]$ can have precedence when rule $[\sqcup_M (2)]$ is also applicable.

\[
B_1 \sqsubseteq_B B_2 \quad (B_1, (M_1, V_1)) \sqsubseteq_P (B_2, (M_2, V_2)) \quad D \sqsubseteq_D D' \quad D_1 \sqcup_D D_2 = \left\{ (B_{ij}, (M_{ij}, V_i \cup V_j)) \mid \begin{array}{l}
(B_i, (M_i, V_i)) \in D_1, \ (B_j, (M_j, V_j)) \in D_2, \\
B_{ij} = B_i \cup B_j, \ B_{ij} \vdash M_i \sqcup_M M_j \Rightarrow M_{ij}
\end{array} \right\}
\]

Figure 3.7: Orders between types

The orders and deterministic join operations between types other than the memory type, are defined in the standard manner as shown in Figure 3.7. Since all the orders between types are reflexive and transitive (preorder), the partial order (analysis domain for the fixpoint iteration) for each type $X$ can be automatically constructed using the equivalence relation $\sim_X$ such that $X_1 \sim_X X_2$ iff $X_1 \sqsubseteq_X X_2$.
and $X_2 \subseteq X_1$ [5]. Note that we omitted the definition of the greatest element $\top_X$ and the least element $\bot_X$ for each type $X$, where the corresponding extensions are trivial.

### 3.6 Memory Updates

![Figure 3.8: Typing rules for memory updates](image)

The judgement $B, M, V \vdash_u A \Rightarrow M'$ specifies the safe approximation of memory $M'$ after updating the set of access paths $A$ to value $V$ with memory $M$ and input context $B$, which is caused either by an assignment statement or by a procedure invocation. Typing rules for this judgement are given in Figure 3.8.

The typing rule for strong update ([update-s]) is explained in Section 2.4, where $M - \{AP\}$ refers to the memory after removing mapping $[AP \mapsto V]$ (if it exists) from $M$.

If we are updating a collapsed access path $AP$ to $V$ with $M$ ([update-w]), we do not know for sure which concrete access path represented by $AP$ is updated by this update operation. Therefore, the old side effect $[AP \mapsto V_{\text{old}}] \in M$ cannot be safely killed when the new side effect $[AP \mapsto V]$ is added to $M$. On the other hand, simply adding $[AP \mapsto V]$ to $M$ to keep the old side effect $[AP \mapsto V_{\text{old}}] \in M$ for this case can be problematic since a loop can cause an unbounded number of
side effects on \( AP \). Therefore, we conservatively bound the length of the update history by approximating the set of updates performed on one access path into one symbolic update as follows. We first determine \((M - \{AP\})[AP \mapsto V']\), where the side effect for \( AP \) is safely (i.e., \( M \sqsubseteq_M (M - \{AP\})[AP \mapsto V'] \)) moved to the rightmost position of \( M \) using the shift operation of Figure 3.6. Then, we join the new update value to the old value by adding merged side effect \([AP \mapsto V \cup V']\) to \( M - \{AP\} \). The update operation for the multiple access paths ([update-w]) is explained similarly.

### 3.7 Typing Rules for the Program

In this section, we explain the main judgements of our memory type system:

1. \( \vdash_p p \Rightarrow (M, V) \)

2. \( \Gamma \vdash_f dec \texttt{fun } f(x)=e \)

3. \( B, M \vdash_e e \Rightarrow M', V \)

Judgement 1 is read as “The type of a program \( p \) is \((M, V)\)”

This means that the safe approximation of the execution result of program \( p \) is \( M \) and \( V \). Judgement 2 is read as “\( \Gamma \) types \( f \)”, which means that \( \Gamma \) is a type environment that safely approximates the behavior of procedure \( f \). Judgement 3 is read similarly – intuitively it specifies the safe approximation of the memory and the memory location obtained after executing \( e \) under a calling context abstracted by \( B \) and a memory abstracted by \( M \). Typing rules for these judgements are given in Figure 3.10, and the auxiliary operations used in those rules are given in Figure 3.9.

The [Program] rule computes the safe approximation of execution results \( M_p \) and \( V_p \) of given program \( p \) using type environment \( \Gamma \), which provides the safe summary of each procedure \( f_i \) invoked in \( e \). In the type system, we simply assume that such
\[ \begin{align*}
M \xrightarrow{B} M'[S] & \quad \text{the memory after we iteratively perform each instantiated} \\
& \quad \text{update} [A_i \mapsto V_i] \text{ of } M'[S] \text{ to } M \text{ under } B \\
\text{field}(A, fld) & = \{AP’ \mid AP \in A \land \alpha(AP.fld) = AP’\} \\
AP[\overline{y/x}] & = \begin{cases} 
\text{if } AP = x_i.X \text{ then } y_i.X \text{ else } AP 
\end{cases}
\end{align*} \]

### Substitution

\[ \begin{align*}
AP[S] & = S(AP) \\
\{AP_1, \cdots, AP_n\}[S] & = S(AP_1) \cup \cdots \cup S(AP_n) \\
([AP \mapsto V])[S] & = [AP[S] \mapsto V[S]] \\
(h_1 \cdots h_n)[S] & = h_1[S] \cdots h_n[S] \quad \text{where } h_i = [AP_i \mapsto V_i] \\
\{as_1, \cdots, as_n\}[S] & = as_1[S] \cup \cdots \cup as_n[S] \\
(AM_{1 \#}AP_2)[S] & = \{AP_i \in S(AP_1)\#AP_j \in S(AP_2)\}
\end{align*} \]

### Access path resolving

\[ \begin{align*}
\text{[resolve(1)]} & \quad (AP = x \lor AP = l) \\
& \frac{B, M \vdash_\text{res} AP \Rightarrow \{AP\}}{B, M \vdash_\text{res} AP \Rightarrow V} \\
\text{[resolve(2)]} & \quad AP = AP’.* \\
& \frac{B, M \vdash_\text{res} AP’ \Rightarrow V’ \quad B, M \vdash_\text{res} V \Rightarrow V}{B, M \vdash_\text{res} AP \Rightarrow V} \\
\text{[resolve(3)]} & \quad AP = AP’.fld \\
& \frac{B, M \vdash_\text{res} AP’ \Rightarrow V’ \quad \text{field}(V’, fld) = V}{B, M \vdash_\text{res} AP \Rightarrow V}
\end{align*} \]

Figure 3.9: Auxiliary operations

\(\Gamma\) for a program are given globally. The inference algorithm of Chapter 4 describes
the algorithm to infer such \(\Gamma\). In [Fundec], we first prepare a set of input contexts
\(B_1, \cdots, B_n\), where each \(B_i\) is considered to be relevant. If each \((B_i, \epsilon)\) types \(e\) as
\((M_i, V_i)\), the summary of procedure \(f\) is \(\{(B_1, (M_1, V_1)), \cdots, (B_n, (M_n, V_n))\}\). Note
that \(\Gamma\) used to type procedure \(f\) contains its own summary, which explains the typing
for recursion and mutually recursive procedures.
Figure 3.10: Typing rules for program
[Addr-id] computes the access paths \( \{x\} \) represented by expression \&x, [Addr-∗] computes access paths \( V \) represented by the expression \( e \), and [Addr-fld] computes access paths \( A \) for the field \( fld \) of the access paths represented by expression \( e \). [Deref-∗] computes memory \( M_1 \) and value \( V_1 \) by typing expression \( e \), and then computes the set of access paths \( V \) to which \( V_1 \) points using the typing rules for memory reads in Figure 3.5. [Deref-fld], which computes the value of field \( fld \) of expression \( e \), is similar to [Deref-∗].

[Asgn] successively types each expression \( e_1 \) and \( e_2 \), computing the memory \( M_2 \), the value (l-value) \( V_1 \) of \( e_1 \), and the value (r-value) \( V_2 \) of \( e_2 \). Then it computes the result memory \( M_3 \), which is obtained after updating \( V_1 \) to \( V_2 \) with \( M_2 \) using the typing rules for memory updates in Figure 3.8.

In [App], we first compute the substitution \( S \) that represents the calling context using the access path abstraction function \( \alpha \) (explained in Section 3.4) used to type callee procedure \( f \). \( AP'[\overline{y}/\overline{x}] \) represents all concrete access paths of the caller that are abstracted into abstract access path \( AP \) of procedure \( f \), whose formal parameters are \( \overline{x} \). So, the caller’s access paths for \( AP \) are computed by iteratively reading the sequence of access path selectors of \( AP'[\overline{y}/\overline{x}] \) with the caller’s memory \( M \) according to the access path resolving rules of Figure 3.9. Next, we determine which input–output relation of the summary of procedure \( f \) is correct to type the procedure call expression. The purpose of \( B \vdash B_i[S] \) is not to check any property for verification as in the typical type system, but to select an \textit{inout} \((B_i,(M_i,V_i))\), which is proved to be a safe approximation of the behavior of the callee procedure for the calling context that satisfies \( B_i \). Such an input context always exists in the summary as long as the input context \( \emptyset \) (which means all calling contexts) exists in the summary. Finally, we iteratively perform each instantiated update \([A_i \mapsto V_i]\) of \( M_i[S] \) with the caller’s memory \( M \) and input context \( B \) (see the definition of \( M \prec B M_i[S] \) and the substitution in Figure 3.9).
[New] can be simply regarded as a procedure application whose summary is 
\{((\emptyset, (\epsilon, \{l\})))\}, where \(l\) is a representative (collapsed) access path for the memory locations allocated at \(l\). Note that although we have used a simple abstraction for the dynamically allocated memory addresses, a variety of well-known techniques to increase the precision can be adapted in our type system without loss of generality: We can distinguish offsets of a memory chunk allocated at one program point using the size or C-type information used in the dynamic allocation. For example, consider \texttt{malloc(sizeof(struct list))}, where \texttt{list} is a structure that has two fields \(d\) and \texttt{next}. We can use a refined \(\alpha\) that distinguishes the fields of the structure \texttt{list}. We can also relate the call sequence \(\sigma\) to the allocation point \(l\) using a tuple \((\{l\}, \sigma)\), where the type variable \(\sigma\) is instantiated with the caller’s label \(l’\) at the call site, which yields \((\{l, l’\}, \sigma)\). In this case, we limit the size of the call sequence set to some constant \(k\).

[If] merges the execution results of \(e_1\) and \(e_2\) using the typing rules for the memory join explained in Section 3.5.

### 3.8 Typing Example

Figure 3.11 shows the example typing (type derivation trees) for the following C code:

```c
int ***g1, ***g2, ***g3;

f1(int **p, int **q, int **r) {
    if(?) \{ l1 : *p = malloc(sizeof(int));\} else \{ l2 : *q = *r; \}
}

f2() \{ g2 = g1; f1(g1, g2, g3); \}
```

Note that the C program is translated into our language as described in Section 3.1.
Figure 3.11: Typing example
For example, $q=r$ is translated into $q \mathrel{:=} r$ since the left-hand side and right-hand side of the assignment compute the l-value of $q$ and the r-value of $r$, respectively.

$[t_2]$ and $[t_4]$ are the type derivation trees for the assignments $l_1$ and $l_2$ of procedure $f_1$, respectively. Note that we simply assume that access path abstraction function $\alpha$ and set of relevant input contexts $B_1, \ldots, B_n$ of each procedure are given for the type system as we mentioned in Sections 3.4 and 3.7. In this example, we assume an $\alpha$ such that access paths $p.*$ and $q.*$ are unique access paths and the relevant input contexts of $f_1$ are $\emptyset$ and $\{p.* \# q.*\}$.

$[t_5]$ computes the join of $[p.* \mapsto \{l_1\}]$ and $[q.* \mapsto \{r.*.*\}]$ which are the typing results of each branch. Note that the result of $\text{shift}(B, [p.* \mapsto \{l_1\}], q.*)$ which is used in the first depth of the type derivation tree is dependent on input context $B$. If $B = \{p.* \# q.*\}$, then $V_1 = \{q.*.*\}$. If $B = \emptyset$, then $V_1 = \{l_1, q.*.*\}$.

$[t_6]$ is the type derivation tree for the if expression, which is typed by the typing rule $[\text{If}]$ with the type derivation trees $[t_2]$, $[t_4]$, and $[t_5]$.

$[t_7]$ is the type derivation tree for computing the summary of procedure $f_1$. For each relevant input context $\emptyset$ and $\{p.* \# q.*\}$, the corresponding typing results of $[t_6]$ are added to the summary.

$[t_8]$ is the type derivation tree for the procedure call in procedure $f_2$. Since the typing result of $g_2 = g_1$ is $[g_2 \mapsto \{g_1.*\}]$, access paths $p.*$ and $q.*$ of the summary of procedure $f_1$ are resolved as $g_1.*$ and $g_1.*$, respectively. Therefore, the output for the input context $\emptyset$ is applied for this calling context.

### 3.9 Correctness of the Type System

In this section, we prove that the typing result of our memory type system safely approximates the execution result of the dynamic semantics, which is formally proved in Theorem $1$ with following definitions.

---

$^5$We marked the used rule at the right-hand side of each type derivation tree, and the abbreviation $\ldots$ $[t_1]$ is used to denote the type derivation tree $[t_1]$. 

---

31
Definition 1 [ Abstraction Relation ] We relate the semantic domains of the formal semantics of Figure 3.2 to the type domains of the type system of Figure 3.3 using an abstraction relation $X \preceq Y$ – intuitively, $X \preceq Y$ means that type $Y$ of the type system safely approximates (abstracts) domain $X$ of the dynamic semantics. The formal definition of the relation is defined below.

\[
\begin{align*}
\preceq &\quad \{\alpha(ap) \mid ap \in v\} \subseteq V \\
&\quad v \preceq V \\
&\quad \text{known}(M) \\
&\quad \forall ap \in m : \neg \text{unique}(ap) \Rightarrow \text{collapsed}(\alpha(ap), \alpha) \\
\end{align*}
\]

\[
\preceq \\
\forall mem \in m : \forall ap \in \text{Dom}(mem) : \\
(mem(ap) = v \land \alpha(ap) = AP \land \emptyset, M \vdash AP \Rightarrow V \land v \preceq V) \\
\]

Definition 2 [ Continuation Typing ] Judgement $M, V \vdash K : (M_K, V_K)$ is read as “The type of a continuation $K$ is $(M_K, V_K)$ under the memory $M$ and the value $V$ – intuitively it specifies the safe approximation of the memory and the memory location obtained after executing the left computation $K$ when the current computation is finished with a memory abstracted by $M$ and a memory location
abstracted by $V$. Typing rules for this judgement is defined below.

\[ K-\epsilon \]
\[
M, V \vdash \epsilon : M, V
\]

\[ K\text{-}\text{return} \]
\[
\emptyset, M \vdash e \Rightarrow M_1, V_1 \quad M_1, V_1 \vdash K : M_2, V_2
\]
\[
M, V \vdash K : M_2, V_2
\]

\[ K\text{-}\text{seq} \]
\[
\emptyset, M \vdash r \Rightarrow V_1 \quad M, V_1 \vdash K : M_2, V_2
\]
\[
M, V \vdash deref.K : M_2, V_2
\]

\[ K\text{-}\text{field} \]
\[
\emptyset, M \vdash e \Rightarrow M_1, V_1 \quad M, V_1 \vdash K : M_2, V_2
\]
\[
M, V \vdash field(fld).K : M_1, V_1
\]

\[ K\text{-}\text{assign} \]
\[
\emptyset, M \vdash e \Rightarrow M_1, V_1 \quad M, V_1 \vdash V \Rightarrow M_2 \quad M_2, V \vdash K : M_3, V_3
\]
\[
M, V \vdash assignl(e).K : M_3, V_3
\]

\[ K\text{-}\text{assign} \]
\[
v_1 \preceq V_1 \quad \emptyset, M \vdash u \Rightarrow V_1 \quad M, V_1 \vdash K : M_2, V_2
\]
\[
M, V \vdash assign(u).K : M_2, V_2
\]

\[ K\text{-}\text{join} \]
\[
m \preceq M \quad \emptyset, M \vdash e_2 \Rightarrow M_2, V_2 \quad M_1 \cup M, M_2, V_1 \cup V_2 \vdash K : M'', V''
\]
\[
M_1, V_1 \vdash jointo(e_2, m).K : M'', V''
\]

\[ K\text{-}\text{join} \]
\[
v_1 \preceq V_1 \quad m_1 \preceq M \quad M_1 \cup M, M_1 \cup V \vdash K : M'', V''
\]
\[
M, V \vdash join(v_1, m_1).K : M'', V''
\]

**Definition 3 [ Configuration Typing ]** Judgement $\vdash C : (M, V)$ is read as “The type of a configuration $C$ is $(M, V)$ – intuitively it specifies the safe approximation of the memory and the memory location obtained after executing the configuration $C$ with the formal semantics of Figure 3.2. Typing rules for this judgement is defined
below.

\[
\frac{m \leq M \quad \emptyset, M \vdash e \Rightarrow M_1, V_1 \quad M_1, V_1 \vdash K : (M_2, V_2)}{\vdash (m, e, K) : (M_2, V_2)}
\]

\[
\frac{m \leq M \quad v \leq V \quad M, V \vdash K : (M', V')}{\vdash (m, v, K) : (M', V')}
\]

**Theorem 1 [Soundness of the memory type system]** If \(\vdash_p p \Rightarrow (M, V)\) and \((p) \rightarrow (m, v, \epsilon)\), then \(m \leq M\) and \(v \leq V\).

**Proof:**

By assumptions and the formal semantics of Figure 3.2, we have

1. \((\text{fundec}; e) \rightarrow (\emptyset, e, \epsilon) \rightarrow (m, v, \epsilon)\) and
2. \(\vdash_p \text{fundec}; e \Rightarrow (M, V)\).

By (2) and rule [Program], we have

3. \(\Gamma \vdash_f \text{fundec_i}\) and
4. \(\emptyset, \epsilon \vdash_e e \Rightarrow M, V\).

Since we have \(\{\emptyset\} \leq \epsilon\) (by definition 1), (4), and \(M, V \vdash \epsilon : (M, V)\) (by rule \([\text{K-}\epsilon]\)),

5. \(\vdash (\emptyset, e, \epsilon) : (M, V)\) is satisfied by rule \([\text{C-}\epsilon]\).

By (1), (5), and the lemma 25 (we moved the formal proof of this lemma to Appendix A for presentation clarity), we have

6. \(\vdash (m, v, \epsilon) : (M, V)\).

By (6), rule \([\text{C-v}]\), and rule \([\text{K-}\epsilon]\), we have

\(m \leq M\) and \(v \leq V\).  \(\square\)
4. The Analysis Algorithm

In this chapter, we formulate the bottom-up pointer analysis algorithm as a type inference algorithm for the memory type system in Chapter 3. The primary goal of this inference algorithm is computing a safe $\Gamma$ which is simply assumed to be given in the type system. Note that [Fundec] recursively uses $\Gamma$ to specify the conditions for $\Gamma$ to safely approximate the behavior of procedure $f$, which means that it is simply a recursive equation that is usually solved by a fixpoint iteration algorithm [21]. Since we are interested in an effective solution among several safe solutions for the equation, the method to let only a bounded number of relevant input contexts be considered when computing a safe $\Gamma$, is the secondary goal of our inference algorithm.

4.1 Type Inference without Partitioning

```plaintext
np_infer_r(B, M, \{AP_1, \cdots, AP_n\}) =
return (np_infer_AP(B, M, AP_1) \cup \cdots \cup np_infer_AP(B, M, AP_n));

np_infer_AP(B, \epsilon, AP) = return (\{\alpha(AP,*)\})

np_infer_AP(B, M[AP \mapsto V_1], AP) =
if unique(\alpha, AP) then return (V_1);
else return (V_1 \cup np_infer_AP(B, M, AP));

np_infer_AP(B, M[AP_1 \mapsto V_1], AP) =
if infer_{as}(B, AP_1 \# AP) then return (np_infer_AP(B, M, AP));
else return (V_1 \cup np_infer_AP(B, M, AP));
```

Figure 4.1: Non-partitioning operations

Since the memory type system does not have any declarative deduction rules such

---

1The access path abstraction function $\alpha$ discussed in Section 3.4 can be determined before computing the summary of each procedure using the C-type information of each variable. For example, the abstract access paths for a variable whose C-type is a recursive structure are statically determined in the manner discussed in Section 2.1.
as a subsumption rule, the non-partitioning operations (abstract transfer functions) np_infer_{as} ∈ B × as → Bool, np_infer_{B} ∈ B × B → Bool, np_infer_{AP} ∈ B × M × AP → V, np_infer_{r} ∈ B × M × A → V, np_infer_{M} ∈ B × M × M → Bool, np_infer_{\sqcup M} ∈ B × M × M → M, np_infer_{u} ∈ B × M × V × A → M, np_infer_{res} ∈ B × M × AP → V, and np_infer_{e} ∈ B × M × e → M × V that respectively compute the safe typing results for the given input context B of judgements B ⊢ as as, B ⊢ B B′, B, M ⊢ AP ⇒ V, B, M ⊢ r A ⇒ V, B, M ⊢ r ⊑ M ⇒ M, B, M ⊢ r ⊔ M ⇒ M, B, M ⊢ u A ⇒ M, B, M ⊢ e e ⇒ M′, V, and B, M ⊢ res AP ⇒ V, and B, M ⊢ e e ⇒ M′, V, are trivially defined. For example, the non-partitioning read operation np_infer_{r} ∈ B × M × A → V which computes the typing result V for the judgement B, M ⊢ r A ⇒ V using the given B, M, and A, can be trivially defined as shown in Figure 4.1. As a result, the non-partitioning type inference algorithm that computes the summary of each procedure for the fixed input context \emptyset, which abstracts every input context, is trivially defined using the fixpoint iteration for Γ explained in Section 4.4.

4.2 Type Inference with Lazy Partitioning

\[
\text{partition}(B, M, A) = \\
\{B\}; \\
\text{for each } AP \in A \{ \\
\text{assume } M = M'[AP \mapsto V][AP_1 \mapsto V_1] \cdots [AP_n \mapsto V_n] \\
B_d = B \cup \{AP_1 \# AP_1 | AP_1 \in \text{aliasable}(AP, \{AP_1, \cdots, AP_n\})\}; \\
\text{if } \neg (\exists B' \in BS_g : B_d \subseteq B') \land |BS_g| < k \text{ then} \\
BS = BS \cup \{B_d\}; \\
BS_g = BS_g \cup \{B_d\}; \\
k = k + 1; \\
\} \\
\text{return } (BS); \\
\text{infer}_{r}(B, M, A) = \text{return } (\{(B_i, \text{np_infer}_{r}(B_i, M, A)) | B_i \in \text{partition}(B, M, A)\});
\]

Figure 4.2: Partitioning operations

In this section, we extend the non-partitioning operations of Section 4.1 to introduce new input contexts when they are identified as relevant.
The partition($B, M, A$) operation of Figure 4.2 computes a set of input contexts \(\{B_1, \cdots, B_n\}\) partitioned from \(B\) (i.e., \(B_i \subseteq_B B\)) that are identified as relevant for determining the set of pointed access paths of \(A\) when the sequence of side effects is \(M\). The global variable \(BS_g\) in this operation is used to improve the efficiency of the inference algorithm by avoiding duplicated partitioning, and the global variable \(k\) means the partitioning bound.

The partitioning read operation \(\text{infer}_r \in B \times M \times A \rightarrow 2^{(B,V)}\) of Figure 4.2, which introduces the relevant contexts \(B_i\)s by need and computes the corresponding read values \(V_i\)s, are defined as follows. When there is a read operation that reads the set of access paths \(A\) with \(M\) under \(B\), we first determine the relevant contexts using the partition operation. Then, for each relevant context \(B_i \in \text{partition}(B, M, A)\), we compute read value \(V_i\) using the non-partitioning read (\(\text{np}_{\text{infer}}_r \in B \times M \times A \rightarrow V\)) operation of Section 4.1.

Partitioning operations for other non-partitioning operations are defined in a similar manner to \(\text{infer}_r\).

### 4.3 Intraprocedural Analysis Stage: \(\text{infer}_e\)

The intraprocedural analysis stage of our inference algorithm is defined by operation \(\text{infer}_e \in B \times M \times e \rightarrow D\) as shown in Figure 4.3. \(\text{infer}_e(B, M, e) = D\) summarizes the behavior of expression \(e\) of procedure \(f\) for a given \(B\), \(M\), and \(\Gamma\) (\(\Gamma\) are given globally from the interprocedural stage). In terms of data flow analysis, \(\text{infer}_e\) can be considered as an abstract transfer function for node \(n_e\) of a procedure \(f\). \(\text{infer}_e\) computes the safe approximation \(D\) for the execution results of node \(n_e\) when the side effect from the start of procedure \(f\) to the predecessor of \(n_e\) is \(M\) under \(B\). If the behavior of \(e\) for \(M\) is unique under \(B\), \(D\) is computed to be a singleton set \(\{(B, (M_e, V_e))\}\). If \(e\) behaves differently depending on \(B_i\) such that \(B_i \subseteq B\), which can be effectively identified by the partitioning operations of Section 4.2, we introduce such \(B_i\)s into the analysis and compute the corresponding results as \(\{(B, (M_e, V_e)), (B_1, (M_1, V_1)), \cdots, (B_n, (M_n, V_n))\}\).

For \(e_1 := e_2\), we first summarize the behavior of \(e_1\) partitioning \(B\) to \(B_i\)s if they
infer_e(B, M, e_s) =

\[ D_B = \emptyset; \]

\{

match e_s with

| e_1 := e_2 \rightarrow 
  for each (B_1, (M_1, V_1)) \in \text{infer}_e(B, M, e_1) \quad (* B_1 \subseteq B *)
  for each (B_2, (M_2, V_2)) \in \text{infer}_e(B_1, M_1, e_2) \quad (* B_2 \subseteq B_1 *)
  for each (B_3, M_3) \in \text{infer}_e(B_2, M_2, V_2, V_1) \quad (* B_3 \subseteq B_2 *)
  \[ D_B = D_B \cup \{(B_3, (M_3, \perp))\}; \]

| &x \& (e.fld) \& (*e) \& e.fld \& new | e_1; e_2 \rightarrow 
  similar partitioning with e_1 := e_2

| if(e_1, e_2) \rightarrow 
  D_B = \text{trim}(\text{infer}_e(B, M, e_1) \cup_D \text{infer}_e(B, M, e_2))

| f \overline{y} \rightarrow 
  D = \Gamma(f);

B_S = \{(B', S) \in \text{reach}(B, M, A[\overline{y}/\overline{x}]) | A \text{ appears in } D\};

for each \((B', S) \in B_S\) \{

(* select one satisfied P \in D for the given calling context \((B', S)\) *)

D' = \{(B_i, (M_i, V_i)) \in D | B' \vdash B_i[S]\};

P = (B_i, (M_i, V_i)); \text{ where } (B_i, (M_i, V_i)) \in D' \land \exists P' \in D' : P \sqsubseteq_p P'

D_B = D_B \cup \{(B, (M, V_i[S])) | (B, M) \in \text{iupdate}(B', M, M_i[S])\};

(* partitioning if needs *)

for each \((B_i, (M_i, V_i)) \in (D - D')\) \{

  if \(|BS_g| < k \land \neg \exists B'' \in BS_g : B' \cup B_i[S] \subseteq B''\) \{
    BS_g = BS_g \cup \{(B' \cup B_i[S])\};
    D_B = \{(B, (M, V_i[S])) | (B, M) \in \text{iupdate}(B' \cup B_i[S], M, M_i[S])\};
    D_B = D_B \cup D_B';
    else skip;
  \}
\}

\}

return \((D_B)\);

\[ \]

Figure 4.3: Intraprocedural analysis stage

38
are relevant to type (determine the behavior of) $e_1$. Then for each partitioned $B_i$ of $\text{infer}_e (B,M,e_1)$, we summarize the behavior of $e_2$, partitioning each $B_i$ to $B_j$s again if they are relevant to type $e_2$. Finally, we update $V_1$ of $\text{infer}_e (B,M,e_1)$ to $V_2$ of $\text{infer}_e (B_1,M_1,e_2)$ with memory $M_2$ obtained after executing $e_2$, partitioning $B_j$ to $B_k$s again if they are identified as relevant.

For $\text{if}(e_1,e_2)$, we type each expression $e_1$ and $e_2$, and then join the resulting summary $D_1$ and $D_2$ using the $\sqcup_D$ operation defined in Figure 3.7. In terms of data flow analysis, $\text{if}(e_1,e_2)$ can be considered as the join operation for a CFG node $n$ whose predecessors are $e_1$ and $e_2$. Note that $D_1 \sqcup_D D_2$ operation can introduce $k^2$ number of *inout* elements in the worst case when $k$ is the partitioning bound. In this case, we limit the size of $D_1 \sqcup_D D_2$ to $k$ using the trim operation $\text{trim}(D)$. Note that an *inout* $(B_i,(M_i,V_i))$ of $D$ is meaningless if $(B_j,(M_j,V_j)) \in D$, $B_i \sqsubseteq B_j$, $B_i \vdash M_j \sqsubseteq M_i$, and $V_j \subseteq V_i$. The trim operation first removes such meaningless summary from $D$ that are possibly introduced by $\sqcup_D$. If every *inout* element of $D$ is meaningful but the partitioning bound $k$ is exceeded, the trim operation discards arbitrary elements of $D$, except the top *inout* (i.e., $B = \emptyset$) so as to maintain the overall input coverage of the procedure summary.

For $f \bar{y}$, the $\text{reach}(B,M,A[\bar{y}/\bar{x}])$ operation first computes the access paths of the caller for each $AP \in A$ that are reachable by access path $AP'$ with $M$ such that $\alpha(AP') = AP[\bar{y}/\bar{x}]$. This introduces new input contexts $\{B_1, \cdots, B_n\}$, where $B_i \sqsubseteq B$, if they are relevant to computing the value of each $AP$. For each introduced input context $B_i$, this operation computes the corresponding substitution $S_i$. The $\text{update}(B,M,[A_1 \mapsto V_1] \cdots [A_n \mapsto V_n])$ operation performs the sequence of updates $[A_i \mapsto V_i]$ iteratively with $M$ partitioning $B$ by needs.

### 4.4 Interprocedural Analysis Stage: infer$_p$

The interprocedural analysis stage of our inference algorithm, given in Figure 4.4, traverses the procedure declarations of program $p$ in a bottom-up manner computing the summary of each procedure. We assume that a pre-order enumeration of the
\[
\text{infer}_{p}(\text{fundec}; e_{\text{main}}) = \\
\quad \text{assume } \Gamma \text{ initially contains summaries of all library procedures} \\
\quad \text{for each } SCC \in p \text{ in the bottom-up order} \{ \\
\quad \quad \text{initialize the summary of each procedure } f \in SCC \text{ to the bottom type} \\
\quad \quad \quad \text{(i.e., } \Gamma(f) = \{(\top, \bot, \bot)\}) \\
\quad \quad \text{do } \{ \\
\quad \quad \quad \text{for each } \text{fun } f(\overline{x}) = e \in SCC \{ \\
\quad \quad \quad \quad BS_g = \{\top\}; \quad \Gamma = \Gamma[f \mapsto (\Gamma(f) \sqcup \text{infer}_e(\top, \epsilon, e))]; \\
\quad \quad \quad \} \text{ while (} \Gamma \text{ is changed in the previous iteration) } \\
\quad \quad \} \text{ while (} \Gamma \text{ is changed in the previous iteration) } \\
\quad \text{return } (\text{infer}_e(\emptyset, \epsilon, e_{\text{main}}));
\]

Figure 4.4: Interprocedural analysis stage

strongly connected components in the call graph is given\(^2\). The summary of each procedure is computed using the fixpoint iteration whose termination is discussed in Section 4.5. Note that \(\Gamma\), which contains the summary of every callee procedure in procedure body \(e\), includes the prefixpoint solution summary for the inferred procedure itself and mutually recursive functions.

### 4.5 Termination and Soundness

If we never partition \(B\) in \(\text{infer}_e\) of Section 4.3, the abstract transfer function \(\text{infer}_e\) is \emph{monotonic} since this case is identical to the memory type system where the [Fun-dec] rule considers only one input context \(\top\) (i.e., \(\emptyset\)). We refer to Appendix A (lemma 11) for the formal proof of the monotonicity of our memory type system. It is trivial to show that \(\text{infer}_e\) is still monotonic even in the presence of the partition operation, since this simply introduces finite input contexts \(B_i\)s, and the corresponding output for each input context \(B_i\) is computed in the same manner as in

\(^2\)We refer to existing work [3] for the incremental construction of the call graph in the presence of the function pointers in the bottom-up pointer analysis, which could be adapted in our pilot implementation of Chapter 6 without loss of generality.
the memory type system (which is monotonic). However, when we composite the extensive function trim to infer\(\varepsilon\) (i.e., infer\(\varepsilon\) is a function such that infer\(\varepsilon\) = \(E \circ F\), where \(F\) is monotonic and \(E\) is extensive), as in Figure 4.3, infer\(\varepsilon\) does not guarantee the monotonicity. In order to ensure that the fixpoint iteration of Figure 4.4 always terminates with the postfixpoint (safe) solution of \(F\), we define the sequence \(D_0, \ldots, D_i\) of our fixpoint iteration based on Theorem 2.

**Theorem 2** [Termination with postfixpoint] If \(D\) is finite, \(F : D \rightarrow D\) is monotonic, \(E : D \rightarrow D\) is extensive, then the sequence

\[
D_0 = \bot_D \\
D_{i+1} = D_i \sqcup_D (E \circ F)(D_i)
\]

is stationary and its limit is the postfixpoint of \(F\).

**Proof:** The sequence \(D_i\) is obviously increasing by its definition with \(\sqcup_D\). There exists \(i\) such that \(D_{i+1} \sqsubseteq D_i\) since \(D\) is finite. So, the fixpoint iteration always terminates. Moreover, such \(D_{i+1}\) is the postfixpoint of \(F\) since \(D_i \sqsupseteq D_{i+1} = D_i \sqcup_D (E \circ F)(D_i) \sqsupseteq (E \circ F)(D_i) \sqsupseteq F(D_i)\).

Now, the soundness of the inference algorithm is proved in Theorem 3. Note that we use the monotonic function \(F\) (infer\(\varepsilon\) without trim) in the proof since the limit of sequence \(D_0, \ldots, D_i\) in Theorem 2 is guaranteed to be the postfixpoint of \(F\).

**Theorem 3** [Soundness of the algorithm] If \(\vdash p \Rightarrow (M', V')\), infer\(p\)(\(p\)) = \(D\), and trim is not used, then \(\forall (B, (M, V)) \in D : B \vdash M' \subseteq_M M \land V' \subseteq V\).

**Proof:** The proof is trivial by simple induction on the structure of \(p\) and \(\varepsilon\), since the typing result computed by infer\(\varepsilon\) without the trim operation is the same as the memory type system.

### 4.6 Complexity

For the given input context, the complexity of our inference algorithm depends on the number of different access paths used in a procedure, partitioning bound \(k\),
and the size of procedures (i.e., not the size of the whole program). Suppose that a procedure to be analyzed has $L$ abstract access paths, $M$ procedure calls, and $N$ memory dereferences (counting `**x` as two dereferences when it is an $r$-value expression). First, a read operation costs $O(L)$ and the maximum value size is $L$, and so a dereference operation costs $O(L^2)$. Second, resolving one abstract access path costs $O(L^3)$, and so the total resolution costs $O(PL^3)$, where $P$ is the number of access paths of the callee. Third, checking one $B_i[S]$ costs $O(L^2)$, and so choosing one satisfiable $B_i$ costs $O(kL^2)$. Fourth, since the update operation costs $O(L^2)$, the iterative updates $M \triangleright M_i[S]$ cost $O(PL^2)$. Finally, a procedure call costs $O(L^4)$ if we assume (for simplicity) that $P = L$ and $k < L$.

Because the memory domain size is $L$, the height of the value domain is $L$, and the order of side effects is fixed for a program point (CFG node) in our $\sqcup_M$ operation, there can be a total of $O(L^2)$ fixpoint iterations for the memory under a given input context $B$. Let us consider a chain $D_0 \sqsubseteq_D \cdots \sqsubseteq_D D_i$ that is iteratively computed for a program point using our fixpoint iteration algorithm. By the definition of $\sqsubseteq_D$, every $P'$ in $D_{i+1}$ should refine one $P$ in $D_i$. So, regardless of the cardinality change of the chain $D_0 \sqsubseteq_D \cdots \sqsubseteq_D D_i$ due to trim and partition operations, the maximum number of refinements for one $B$ throughout the chain is $L^2$. Since the maximum number of intermediate alias contexts introduced into the analysis by $\sqcup_D$, our $k$-partitioning inference algorithm can perform a total of $O(k^2L^4)$ fixpoint iterations for $D$.

Taking all the above processes into account, the worst-case complexity when analyzing a procedure is $O((NL^2 + ML^4)(k^2L^4))$. 
5. Experiments

5.1 Implementation

To assess the effects of using the update history, we implemented\footnote{We added about 7500 lines of OCaml code to CIL [20].} various bottom-up and flow- and context-sensitive pointer analyses where the update history is utilized only in the implementation for our analysis. Each analysis is named as follows according to the ability to perform strong updates in (1) direct assignments (e.g., $x = &i$), (2) indirect assignments (e.g., $*x = &i$), and (3) procedure calls (summary applications):

- Analysis “OOO” is our inference algorithm that can kill the side effects in direct assignments, indirect assignments, and procedure calls, by utilizing the update history.
- Analysis “OOX” does not kill the side effects at procedure calls. For the partitioning, only conditions 1 and 3 of Section 2.5 are utilized. Taking all the above-described characteristics into account, the theoretical precision of analysis “OOX” is similar to RCI [2] except that the more-precise condition 2 of Section 2.5 is utilized in “OOX”.
- Analysis “OXX” does not kill the side effects in indirect assignments and procedure calls. Note that the base analysis of [13, 27] is similar to “OXX” in terms of the ability to perform strong updates only in direct assignments although they did not consider partitioning as discussed in Section 6.1.
- Analysis “XXX” never kills the side effects. This was included to demonstrate the overall effects of strong updates in analysis “OOO”.

We were careful to safely extend the inference algorithm of Chapter 4 for real C language. The methods for safely dealing with function pointer, type casting, and
union in the bottom-up pointer analysis were taken directly from a previous study [3]: (1) the call graph, which is needed for the bottom-up traversal of a program, was incrementally constructed considering the function pointer; and (2) the field selector of an access path was modeled as the offset from the base access path, which enabled the analysis to safely deal with arbitrary type casting and union\(^2\). The pilot implementation cannot safely deal with the multi-threading and setjmp/longjmp functions in C code, and hence the programs without these features were used in our experiments. For the safe and efficient aliasable operation of Figure 4.2, the analyzer examined all assignment and procedure call statements of whole-program and safely assumed the disjointness between \(t_1\)- and \(t_2\)-type access paths if there was no value flow between two different C-types \(t_1\) and \(t_2\). For the library procedures used, we manually wrote 281 hand-coded stubs that modeled the side effects of each procedure. For example, the summary of the library procedure realloc was hand-coded to return the allocation point \(l\) joined with its first argument.

The partition and trim operations of the inference algorithm were implemented to keep the \(k\) most-precise partitions when the number of partitions exceeds the given partitioning bound. Note that the number of relevant input contexts in a procedure can be much larger than the given partitioning bound, in which case considering every relevant alias relation contained in the procedure during the analysis might not be useful since a bottom-up analyzer cannot identify which alias relations might actually be used later by a caller. Instead of focusing on this problem in the thesis, we attempted to avoid such a situation by adding the following timeout feature to our inference algorithm: if the analyzer could not finish the summary computation within the given time limit, it immediately stopped the fixpoint iteration and regarded the intermediately computed set of relevant input contexts as the final set, with the expectation that a sufficient number of meaningful input contexts had already been identified. Then, the analyzer finished the summary computation by computing the corresponding outputs for these fixed input contexts. This approach is sound since any relevant input context can be predicted. Note that a time limit of 1 second was used in our experiments, which gave us the same precision as in

\(^2\)Details of the safe field-sensitive analysis in the presence of type casting can be found elsewhere [3, 36, 22].
experiments without the timeout feature, while it ran up to 6.29 times faster.

5.2 Experimental results

Table 5.1 lists the experiments performed with C programs chosen from the SPEC2000 benchmark suite and GNU textutils and preprocessed (merged) by CIL. The characteristics of these programs are summarized in columns 2–5 of the table. Column 6 (kind) lists the performed analysis, and column 7 (k) lists the partitioning bound for each experiment.

The performance of our inference algorithm is quantified in columns 8 (time) and 9 (mem), which include the total cost of preprocessing and postprocessing for each experiment. The experimental results show that our approach is relatively cost-effective in the sense that the precision of the client analysis was improved without sacrificing the performance significantly. However, for the reasonable time and memory limits applied in the experiments\(^3\), all of the tested approaches were unable to analyze large programs such as 254.gap and 176.gcc of the SPEC2000 benchmark suite, which were often used as scalability references.

Column 10 (avg) in Table 5.1 lists the average number of targets for a pointer dereference\(^4\) in a program. Since we did not know how an unknown access path of a procedure would actually be resolved at the invocation point with the results of bottom-up pointer analysis, the counting was performed after propagating the known access paths from the main procedure in a top-down manner, which was similar to phases II and III described by Wilson and Lam [2]. When there were multiple contexts for a program point, the counting was performed after conservatively joining the analysis result for each context. These results demonstrate that the precision of the analysis can be improved when the opportunity for strong updates increases or alias-context-sensitive summarization (partitioning) is considered.

In order to assess the effects of applying different approaches to the client analyses, we also implemented a simple client analysis that detected either uninitialized-

\(^3\)We applied a criterion of scalability such that analysis time and memory are limited to 1 hour and 1 Gigabyte, respectively.

\(^4\)Multilevel dereferencing was counted separately (e.g., **x comprises two dereferences).
<table>
<thead>
<tr>
<th>Program</th>
<th>N (L)</th>
<th>P</th>
<th>S</th>
<th>C</th>
<th>Analysis</th>
<th>Performance</th>
<th>Precision</th>
<th>N-update %</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mem</td>
<td>time</td>
<td>mem</td>
<td>avg</td>
</tr>
<tr>
<td>mcf</td>
<td>13(1.5)</td>
<td>26</td>
<td>1</td>
<td>1</td>
<td>XXX</td>
<td>1</td>
<td>2</td>
<td>1.6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OXX</td>
<td>2</td>
<td>1</td>
<td>1.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OXX</td>
<td>5</td>
<td>1</td>
<td>1.4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OOO</td>
<td>2</td>
<td>1</td>
<td>1.4</td>
</tr>
<tr>
<td>pr</td>
<td>17(2.7)</td>
<td>41</td>
<td>4</td>
<td>2</td>
<td>XXX</td>
<td>1</td>
<td>2</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OXX</td>
<td>2</td>
<td>1</td>
<td>1.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OXX</td>
<td>5</td>
<td>1</td>
<td>1.4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OOO</td>
<td>2</td>
<td>6</td>
<td>1.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OOO</td>
<td>5</td>
<td>5</td>
<td>1.3</td>
</tr>
<tr>
<td>sort</td>
<td>26(3.6)</td>
<td>45</td>
<td>1</td>
<td>2</td>
<td>XXX</td>
<td>1</td>
<td>2</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OXX</td>
<td>2</td>
<td>1</td>
<td>1.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OXX</td>
<td>5</td>
<td>8</td>
<td>1.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OOO</td>
<td>2</td>
<td>4</td>
<td>1.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OOO</td>
<td>5</td>
<td>5</td>
<td>1.2</td>
</tr>
<tr>
<td>gzip</td>
<td>45(3.7)</td>
<td>89</td>
<td>1</td>
<td>1</td>
<td>XXX</td>
<td>1</td>
<td>3</td>
<td>1.4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OXX</td>
<td>2</td>
<td>1</td>
<td>1.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OXX</td>
<td>5</td>
<td>10</td>
<td>1.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OOO</td>
<td>2</td>
<td>2</td>
<td>1.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OOO</td>
<td>5</td>
<td>5</td>
<td>1.2</td>
</tr>
<tr>
<td>tsmp2</td>
<td>33(3.9)</td>
<td>74</td>
<td>1</td>
<td>1</td>
<td>XXX</td>
<td>1</td>
<td>1</td>
<td>1.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OXX</td>
<td>1</td>
<td>1</td>
<td>1.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OXX</td>
<td>5</td>
<td>10</td>
<td>1.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OOO</td>
<td>2</td>
<td>1</td>
<td>1.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OOO</td>
<td>5</td>
<td>1</td>
<td>1.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OXX</td>
<td>1</td>
<td>1</td>
<td>1.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OXX</td>
<td>5</td>
<td>10</td>
<td>1.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OOO</td>
<td>2</td>
<td>1</td>
<td>1.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OOO</td>
<td>5</td>
<td>1</td>
<td>1.5</td>
</tr>
<tr>
<td>anmp</td>
<td>349(14.7)</td>
<td>179</td>
<td>2</td>
<td>3</td>
<td>XXX</td>
<td>1</td>
<td>3</td>
<td>1.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OXX</td>
<td>1</td>
<td>3</td>
<td>1.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OXX</td>
<td>1</td>
<td>3</td>
<td>1.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OXX</td>
<td>5</td>
<td>10</td>
<td>1.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OOO</td>
<td>2</td>
<td>1</td>
<td>1.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OOO</td>
<td>5</td>
<td>1</td>
<td>1.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OXX</td>
<td>1</td>
<td>3</td>
<td>1.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OXX</td>
<td>5</td>
<td>10</td>
<td>1.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OOO</td>
<td>2</td>
<td>1</td>
<td>1.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OOO</td>
<td>5</td>
<td>1</td>
<td>1.5</td>
</tr>
<tr>
<td>xpr</td>
<td>151(17.4)</td>
<td>400</td>
<td>7</td>
<td>11</td>
<td>XXX</td>
<td>1</td>
<td>2</td>
<td>1.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OXX</td>
<td>1</td>
<td>2</td>
<td>1.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OXX</td>
<td>5</td>
<td>4</td>
<td>1.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OXX</td>
<td>1</td>
<td>2</td>
<td>1.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OXX</td>
<td>5</td>
<td>4</td>
<td>1.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OOO</td>
<td>2</td>
<td>1</td>
<td>1.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OOO</td>
<td>5</td>
<td>5</td>
<td>1.2</td>
</tr>
<tr>
<td>crafty</td>
<td>192(22.0)</td>
<td>109</td>
<td>2</td>
<td>1</td>
<td>XXX</td>
<td>1</td>
<td>2</td>
<td>1.9</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OXX</td>
<td>1</td>
<td>2</td>
<td>1.9</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OXX</td>
<td>5</td>
<td>4</td>
<td>1.9</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OXX</td>
<td>1</td>
<td>2</td>
<td>1.9</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OXX</td>
<td>5</td>
<td>4</td>
<td>1.9</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OOO</td>
<td>2</td>
<td>1</td>
<td>1.9</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OOO</td>
<td>5</td>
<td>5</td>
<td>1.9</td>
</tr>
<tr>
<td>twolf</td>
<td>252(24.9)</td>
<td>191</td>
<td>1</td>
<td>1</td>
<td>XXX</td>
<td>1</td>
<td>2</td>
<td>1.7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OXX</td>
<td>1</td>
<td>2</td>
<td>1.7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OXX</td>
<td>5</td>
<td>4</td>
<td>1.7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OXX</td>
<td>1</td>
<td>2</td>
<td>1.7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OXX</td>
<td>5</td>
<td>4</td>
<td>1.7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OOO</td>
<td>2</td>
<td>1</td>
<td>1.7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OOO</td>
<td>5</td>
<td>5</td>
<td>1.7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OOO</td>
<td>2</td>
<td>1</td>
<td>1.7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OOO</td>
<td>5</td>
<td>5</td>
<td>1.7</td>
</tr>
</tbody>
</table>

N: number of CFG nodes (1=1000), L: lines of code excluding comments and blank lines (1=1000), P: number of procedures, S: maximum size of SCC (strongly connected components), C: maximum number of callees by a function pointer, time: execution time on a Linux-based computer with a 3.2-GHz Intel Xeon CPU (in seconds), mem: maximum size of the major heap (in megabytes).

Table 5.1: Experimental results
value or integer-value (e.g., null pointer) dereferences using the results of the pointer analysis. Column 11 (safe) in Table 5.1 lists the percentages of pointer dereferences that were shown to be safe using this type of client analysis, and these are depicted as a graph in Figure 5.1. These results show that our method of pointer analysis was able to prove that more pointer dereferences were safe.

Columns 12 (indirect) and 13 (inter) in Table 5.1 list the percentages of intraprocedural strong updates in indirect assignments and interprocedural strong updates in procedure invocations during the analysis, which influenced the precision results in columns 10 (avg) and 11 (safe). This demonstrates the importance of strong updates during bottom-up pointer analysis. For example, analysis “XXX” was unable to show whether 32% of pointer dereferences of program pr were safe accesses, whereas this was possible using analysis “OXX”. Note that the difference between analyses “OOX” and “OOO” shows the effect of the interprocedural strong updates, which could not be performed in previous bottom-up approaches. Not considering interprocedural strong updates resulted in the client analysis not being able to show
that a maximum of 37% (in \textit{vpr}) of pointer dereferences were safe, even though this was possible with our analysis “OOO”.
6. Related Work

6.1 Bottom-up and Flow-sensitive Pointer Analysis

There are several bottom-up pointer analysis algorithms [13, 27, 2, 31] that are flow- and context-sensitive on which we are focusing in this thesis.

Harrold and Rothermel [13] and Rountev et al. [27] extend the work of Landi and Ryder [19], which is a top-down and flow- and context-sensitive analysis, to design their summary based pointer analysis methods. For the strong updates, both Harrold and Rothermel [13] and Rountev et al. [27] follow the method of Landi and Ryder [19] in which alias information is killed only when the left side of the assignment does not contain a pointer dereference. For the alias context sensitivity, Harrold and Rothermel [13] consider every possible alias context for a procedure in a brute-force manner. The usefulness of this approach for analyzing real C programs is unclear since computing the summary for the exhaustive number of possible alias contexts of a procedure can be prohibitively expensive, as our empirical results demonstrate. Rountev et al. [27] do not consider the problem of inferring the relevant input contexts of a procedure since they assume that this information is provided by an inexpensive front-end whole-program analysis.

Whaley and Rinard [31] proposed a compositional pointer and escape analysis algorithm for Java programs, based on a graph based memory state abstraction. Points-to information is killed only when the left side of the assignment represents a single unique access path that does not escape from the analysis scope (e.g., procedure). Therefore, the intraprocedural strong updates on the unique unknown access paths and the access paths for the global variables are not performed. The interprocedural strong updates are not performed. They extract a single analysis result for the calling context with no aliases and merge nodes for the calling contexts with aliases. In other words, the alias context sensitive summarization is not considered in this work.
The work of Chatterjee et al. (RCI) [2] is the most closely related to our work since they consider both the problems of strong updates on unknown access paths and relevant alias context inference. For the strong updates, the interprocedural strong updates, which are shown to be effective by the difference between “OOX” and “OOO” in our experiments, are not applicable in RCI since they do not keep the information on the order of side effects in the summary of procedure. For the relevant input context inference, precise conditions to be a relevant alias relation, such as conditions 2 and 3 in Section 2.5, cannot be identified in RCI since the decision for these conditions can be made using the information on the order of side effects performed that contributes to the given memory state (points-to information).

Rountev et al. [27] investigated the problem of formalizing a sound analysis for program fragments. As an example fragment analysis, they presented the flow- and context-sensitive pointer analysis discussed above. The techniques they used to formally present the analysis and to prove the soundness of the analysis were based on a data flow analysis framework [30, 4, 6], whereas our technique is based on the subject reduction lemma [34] of type system framework [23].

6.2 Bottom-up and Flow-insensitive Pointer Analysis

Most of pointer analysis algorithms, which involve the notion of constructing the summary of each procedure without the caller procedure and then using this summary to analyze the callers of that procedure, are proposed as flow-insensitive analyses [10, 3, 11, 28, 14, 26]. The advantage of this approach is that it leads to a simpler summary construction algorithm and lower analysis cost than the flow-sensitive approach in general. The disadvantage of this approach is that the analysis results are too imprecise when it is important to take the statement ordering into account. Consider the following C program fragment as a simple example:

1: int* p = NULL; int i;
2: p = &i;
3: *p = 1;
Only the results of the flow-sensitive pointer analysis algorithm, which destructively updates \( p \) at line 2 such as “OXX”, “OOX”, and “OOO” of our experiments, can prove that this program fragment has no null-pointer dereference, since the points-to set of \( p \) at line 3 includes \textit{null} when it is computed by a flow-insensitive pointer analysis or a flow-sensitive pointer analysis without strong updates.

6.3 Top-down Analysis

The top-down pointer analysis has been extensively reported in the literature, with [15, 16] providing extensive lists of the previous work in this field. Most of flow- and context-sensitive pointer analysis algorithms are proposed as top-down analyses [19, 7, 9, 32, 33, 16, 29, 37]. The disadvantage of the top-down approach is that it cannot operate on incomplete programs. Therefore, existing top-down pointer analyses are not readily applicable to the modular software development environment (our target environment). The advantage of top-down analysis is that it generally computes more precise analysis results than bottom-up analysis when the whole program is available and scalable.

6.3.1 Alias context sensitivity

Since top-down analysis is performed from the callers to the callees and hence every access path is a known access path, they need not take the potential aliases between \textit{unknown} access paths into account which potentially causes the conservative approximations in the bottom-up analysis. Consider the example program in Section 1.2. If a bottom-up analysis does not consider the problem of distinguishing the different behaviors of a procedure which are dependent on the calling contexts (i.e., only considers the input context \( \top_B \)), the return value of procedure \( f \) is conservatively approximated as \( \{g_1, g_2\} \) due to the potential alias between \( p.* \) and \( q.* \). Although we attempted to avoid such approximations by distinguishing the behaviors of a procedure for the relevant alias contexts, such kinds of approximations still remain since a procedure can involve an exponential potential number of relevant alias contexts, and we limit the number of alias contexts considered in the summary.
to a constant bound $k$. On the contrary, top-down analyses need not introduce such kinds of approximations since the concrete calling context can be always determined before analyzing the behavior of the procedure.

### 6.3.2 Strong update

The effectiveness of flow-sensitive analysis is affected by the opportunity of the strong updates during the analysis. Our method to identify the killed side effects is similar with Wilson and Lam [32, 33] in the following senses: (1) the memory locations are named according to the allocation site; (2) we distinguish between different fields within a structure but not the different elements of an array; (3) the access paths for the recursive structures are distinguished based on $k$-limiting; and (4) the side effects for the unique unknown access paths can be killed inside the procedure independently of the calling contexts. Taking all of these into account, the stack memory location, which is abstracted into an access path such that the base is a variable and the selectors involve neither a recursive access nor an array, can be destructively updated. All the other locations can be destructively updated only by (4). There are several top-down analyses that provide the methods to increase the opportunity of strong updates that cannot be performed in our approach. For example, Sagiv et al. [29] can kill the side effects on heap allocated objects and recursive data structures by precisely modeling and tracing the shape of memory states, and Yong et al. [35] can distinguish between different elements of an array by precisely modeling and tracing the value of the array index. However, such abilities to precisely trace the uniqueness of abstract memory locations for strong updates are dependent on the analysis results which can be computed in the top-down manner. Therefore, they are not readily adaptable to the bottom-up analysis that operates on incomplete programs, which forms the focus of this thesis.

### 6.3.3 Top-down analyses utilizing the reusable summary

There are various forms of pointer-related top-down analyses [32, 33, 25, 24, 12, 8] that involve constructing the calling-context-dependent summary for each procedure and then using this summary to analyze the callers of that procedure, with the sum-
mary normally being used to improve the performance. Although these previous analyses utilized the summary, they are not readily applicable to bottom-up pointer analysis since the algorithms used to construct the summary can work only when the calling contexts are available. For example, Wilson and Lam [32, 33] considered similar problems to those that we have addressed, such as introducing only the relevant alias contexts into the summary on demand. However, such an ability is restricted to top-down analysis, since the parameterized memory locations (extended parameters [32, 33]) referenced inside a procedure and the alias relations between these extended parameters can be determined only when the calling context is given. In contrast, the update history can abstract memory states of a procedure independently of the information on the calling context, helping the analysis to effectively identify relevant alias contexts. The lazy strong update scheme (postponing the decision to kill the side effects performed on the must-aliasable access paths until the procedure is invoked using the update history) did not need to be considered in [32, 33], since the summary-construction algorithms used operated with given calling contexts. However, those studies inspired the method we used to perform strong updates on the unique unknown access paths inside a procedure independently of the calling context.
7. Conclusions

We have presented an update-history-based approach for the bottom-up and flow-and context-sensitive pointer analysis of C programs. The update-history-based memory representation could effectively guide the strong updates and relevant context searches of the presented bottom-up pointer analysis. We have also provided a formal proof of the soundness of the analysis and the experimental results obtained from a pilot implementation thereof.

There are at least two directions for future work:

- There could be several ways to utilize our update history based states abstraction and analysis techniques: (1) it could be used to abstract side effects of other languages such as Java; (2) it could be used to design a bottom-up analysis that traces state changes of resources where the aliases between abstract resources can occur, which is our ultimate goal [18].

- Our memory type system could represent a framework for designing (formalizing) a bottom-up target analysis for C programs that can be formally proved to be sound based on the techniques used in our work. Only a bottom-up pointer analysis can provide the points-to information for the bottom-up target analysis, since such information should be computed in the bottom-up manner and be represented in a modular way.
A. Correctness of the Type System

In this appendix, we provide the formal proof for the soundness (lemma 25) and monotonicity (lemma 11) of our memory type system.

A.1 Extensions and Assumptions for the Proof

1. We add the subsumption rule to each typing judgement that are abbreviated for presentation brevity.

   \[
   \frac{X \vdash Y : \tau \quad X \vdash \tau \subseteq \tau'}{X \vdash Y : \tau'}
   \] (Subs)

2. We add \([\text{read-}\perp]\) rule and \([\text{update-}\perp]\) rule that are abbreviated for presentation brevity.

   \[
   \begin{align*}
   [\text{read-}\perp] & \quad B, \perp \vdash_{AP} AP \Rightarrow \emptyset \\
   [\text{update-}\perp(1)] & \quad B, \perp, V \vdash_u A \Rightarrow \perp \\
   [\text{update-}\perp(2)] & \quad B, M, V \vdash_u \emptyset \Rightarrow \perp
   \end{align*}
   \]

3. The type of function (\textit{Summary} type) is extended to be quantified by the name of formal parameters of the procedure such as

   \[
   D \in \textit{Summary} ::= \forall \pi : \{(B_1, (M_1, V_1)), \cdots, (B_n, (M_n, V_n))\}
   \]

   that was abbreviated for presentation brevity.

4. We assume that names of formal parameters of all function are different in order to use a global access path abstraction function \(\alpha\) in the proof.

5. We say that a substitution \(S\) preserves the uniqueness of \(\alpha\) when \(\forall AP : \text{unique}(\alpha, AP) \Rightarrow AP[S] = \{AP_1\}\) and denotes it as \(\alpha \vdash S\).
6. We write known$(M)$ iff $\forall AP \in M : \text{known}(AP)$ (i.e., $AP^*.\not\in M$) and $\forall AP_i, AP_j \in M : AP_i \neq AP_j \Rightarrow \emptyset \vdash AP_i \# AP_j$.

7. We use an abstraction function $\alpha$ such that $\alpha(AP.\ast) = \text{uninitialized value} = \bot = \emptyset$ for the main expression.

8. We use following abbreviations (we additionally abbreviate $B$ when $B = \emptyset$) in the proof.

\[
\begin{align*}
B, M \vdash_{AP} AP \Rightarrow V & \iff M_B(AP) = V \\
B, M \vdash_A A \Rightarrow V & \iff M_B(A) = V \\
B, M, V \vdash_a A \Rightarrow M' & \iff M \cdot_B [A \mapsto V] = M' \\
M \searrow M'[S] = M'' & \iff M \cdot_B M'[S] = M'' \\
B \vdash M_1 \cup_B M_2 \Rightarrow M_3 & \iff M_1 \cup_B M_2 = M_3 \\
B \vdash M_1 \subseteq_M M_2 & \iff M_1 \subseteq_B M_2
\end{align*}
\]

9. We assume that the type system is extended as follows for the proof: assume that we are analyzing $e_1; e_2$ with the initial memory $M$ and $AP \in \text{Dom}(M)$ is collapsed. If we have information such that only one particular concrete access path of $AP$ is accessed in $e_1$, we analyze $e_1$ with $M[AP \mapsto M(AP)]$ (introduce a duplicated update history for $AP$) regarding the newly introduced $[AP \mapsto M(AP)]$ as the side effect for the particular concrete access path (unique access path). In other words,

- the update operation on $AP$ in $e_1$ kills the recent update history for $AP$
- the read operation on $AP$ in $e_1$ ignores the valuation of old update history for $AP$
- the analysis result $M_1$ of $e_1$ have two update histories for $AP$ and old update history for $AP$ is not changed

When we analyze $e_2$ where other concrete access paths of $AP$ are accessed, we collapse two update histories for $AP$ into one. Such a situation happens when we call a procedure where $M$ is the caller’s memory state, $S$ is the substitution
(calling context), $AP'$ is proved to be unique inside the callee procedure $f$ (proved to access only one runtime location), and $AP'[S] = \{AP\}$.

In this thesis, we abbreviated the formalization of this idea since such an extension is only related to proving soundness of the current type system. Instead, we prove the typing result in the current type system is more conservative than the typing result of the extended type system.

### A.2 The Proof

**Lemma 1 [ Sound read ]**

If $m \preceq M$, $v \preceq V$, $m(v) = v'$ and $\emptyset, M \vdash_r V \Rightarrow V'$, then $v' \preceq V'$.

**Proof:** Trivial induction on the derivation of $\emptyset, M \vdash_r V \Rightarrow V'$ with the definition of $m \preceq M$ (i.e., if $m \preceq M$, $\alpha(ap) = AP$, $m(ap) = v$ and $\emptyset, M \vdash_{AP} AP \Rightarrow V$, then $v \preceq V$)

**Lemma 2 [ Sound update ]**

If (h1) $m \preceq M$, (h2) $v \preceq V$ and $v' \preceq V'$, (h3) $\text{update-mem}(m, v, v') = m'$, (h4) $\emptyset, M, V' \vdash_u V \Rightarrow M'$, then $m' \preceq M'$.

**Proof:**

- Case $v = \emptyset$:
  
  By assumption (h3), and definition of update-mem, we have (1) $m' = \bot_m$. By (1) and definition of $\preceq_m$, arbitrary $M'$ satisfy $m' \preceq M'$

- Case $v = \{ap\}$ and $\text{unique}(ap)$:
  
  By assumption and (h2)
  
  (1) $\alpha(ap) \in V$ i.e., $V = \{\alpha(ap), AP_1, \cdots, AP_n\}$
  
  By assumption and (h3)
  
  (2) $m' = \text{if unique}(ap) \text{ then } \{\text{mem}[ap \mapsto v] \mid \text{mem} \in m\} \text{ else } \{\text{mem}[ap \mapsto v \cup \text{mem}(ap)] \mid \text{mem} \in m\}$
  
  - Case $\neg \text{unique}(ap)$:
    
    By assumption and (h1), we have (3) $\text{collapsed}(\alpha, \alpha(ap))$. By (3), (1),

57
and rule \([\text{update-w}]\), we have
\[
M \cdot \left[ V \mapsto V' \right] = (M - V)[\alpha(ap) \mapsto V' \cup M(\alpha(ap))][AP_1 \mapsto M(AP_1 \cup V') \cdots [AP_n \mapsto M(AP_n \cup V')].
\]
By \(4\), \(2\), (h1), and definition 1, it is trivial to show \(m' \preceq M \cdot [V \mapsto V']\).

- Case unique\((ap)\):
  By assumption and (h1), we have \((5)\) unique\((\alpha, \alpha(ap))\) or collapsed\((\alpha, \alpha(ap))\).
  If unique\((\alpha, \alpha(ap))\) and \(V = \{\alpha(ap)\}\), then \((6-1)\)
  \[
  M \cdot \left[ V \mapsto V' \right] = (M - V)[\alpha(ap) \mapsto V'] \text{ by } (5), (1), \text{[update-s]}.
  \]
  Otherwise, we have \((6-2)\)
  \[
  M \cdot \left[ V \mapsto V' \right] = (M - V)[\alpha(ap) \mapsto V' \cup M(\alpha(ap))][AP_1 \mapsto M(AP_1 \cup V') \cdots [AP_n \mapsto M(AP_n \cup V')].
  \]
  By \((6-1)\), \((6-2)\), \((2)\), (h1), and definition 1, it is trivial to show \(m' \preceq M \cdot [V \mapsto V']\).

- Case \(v = \{ap_1, \cdots, ap_n\}\):
  similar induction with the case 2

**Lemma 3 [ Sound join ]**
If \((h1)\) \(m_1 \preceq M_1\) and \((h2)\) \(m_2 \preceq M_2\), then \(m_1 \cup m_2 \preceq M_1 \cup M_2\).

**Proof:** By \((h1)\), \((h2)\) and definition 1, we have \((1)\) known\((M_1)\) and known\((M_2)\).
By \((1)\) and definition of known, we have \((2)\)
\[
\forall AP_i, AP_j \in M_1, M_2: AP_i \neq AP_j \Rightarrow \emptyset \vdash AP_i \# AP_j.
\]
When \((2)\) (there is no alias between every distinct access paths), the join operation is independent from the order of update history. Therefore, it is trivial to show \((3)\)
\[
(M_1 \cup M_2)(AP) = M_1(AP) \cup M_2(AP).
\]
By \((3)\) and definition 1, it is obvious that \((m_1 \cup m_2)(ap) \preceq (M_1 \cup M_2)(\alpha(ap))\).

**Lemma 4 [ Smaller substitution preserves input context typing ]**
If \(B \vdash B'[S]\), Dom\((S') = Dom(S)\) and \(\forall AP : S'(AP) \subseteq S(AP)\), then \(B \vdash B'[S']\).

**Proof:** Trivial induction on the derivation of \(B \vdash B'[S]\)

**Lemma 5 [ Read rule is monotonic ]**
If \(B, M_1 \vdash_r A_1 \Rightarrow V_1, M_1 \sqsubseteq_B M_2, \text{ and } A_1 \subseteq A_2\), then \(B, M_2 \vdash_r A_2 \Rightarrow V_2 \text{ and } V_1 \subseteq V_2\).
Proof: The proof is trivial when \( M_1 = M_2 \) and \( A_1 \subseteq A_2 \). Therefore, we show the case when \( M_1 \neq M_2 \) and \( A_1 = A_2 = \{ AP \} \) where its extension for the case \( A_1 \subseteq A_2 \) is trivial. Induction on the derivation of \( \subseteq_M \).

- Case \([\subseteq_M (1)]\):
  By assumption, we have \((1)\) \( M_2 = \text{shift}(B, M_1, AP) \).
  
  - Case \( M_1 = \epsilon \):
    By assumption and \([\text{read-}\epsilon]\), we have \((2)\) \( B, M_1 \vdash_{AP} AP \Rightarrow \{ \alpha(\cdot AP,*) \} \).
    By \((1)\), assumption, definition of shift and rule \([\text{read-}\epsilon]\), we have \((3)\) \( B, \text{shift}(B, \epsilon, AP) \vdash_{AP} AP \Rightarrow \{ \alpha(\cdot AP,*) \} \).
    By \((2)\), \((3)\) and rule \([\text{read-A}]\), we have \( V_1 \subseteq V_2 \).
  
  - Case \( M_1 = M[AP \mapsto V] \) and unique(\(\alpha, AP\)):
    By assumption, and rule \([\text{read-}\Rightarrow]\), we have \((2)\) \( B, M_1 \vdash_{AP} AP \Rightarrow V \).
    By \((1)\), assumption, definition of shift and rule \([\text{read-}\Rightarrow]\), we have \((3)\) \( B, M_2 \vdash_{AP} AP \Rightarrow V \). By \((2)\), \((3)\) and rule \([\text{read-A}]\), we have \( V_1 \subseteq V_2 \).
  
  - Case \( M_1 = M[AP \mapsto V] \) and collapsed(\(\alpha, AP\)):
    By assumption and \([\text{read-safe}]\), we have \((2)\) \( B, M_1 \vdash_{AP} AP \Rightarrow V \cup M_B(AP) \).
    By \((1)\), assumption, definition of shift, and rule \([\text{read-safe}]\), we have \((3)\) \( B, M_2 \vdash_{AP} AP \Rightarrow V \cup M_B(AP) \).
    By \((2)\), \((3)\) and rule \([\text{read-A}]\), we have \( V_1 \subseteq V_2 \).
  
  - Case \( M_1 = M[AP' \mapsto V] \) and \( B \vdash AP#AP' \):
    By assumption and \([\text{read-}\#]\), we have \((2)\) \( B, M_1 \vdash_{AP} AP \Rightarrow M_B(AP) \).
    By \((1)\), assumption, definition of shift and rule \([\text{read-}\Rightarrow]\), \([\text{read-safe}]\), we have \((3)\) \( B, M_2 \vdash_{AP} AP \Rightarrow M_B(AP) \).
    By \((2)\), \((3)\) and rule \([\text{read-A}]\), we have \( V_1 \subseteq V_2 \).
  
  - Case \( M_1 = M[AP' \mapsto V] \) and not \( B \vdash AP#AP' \):
    By assumption and rule \([\text{read-safe}]\), we have \((2)\) \( B, M_1 \vdash_{AP} AP \Rightarrow V \cup M_B(AP) \).
    By \((1)\), assumption, definition of shift and rule \([\text{read-}\Rightarrow]\), \([\text{read-safe}]\), we have \((3)\) \( B, M_2 \vdash_{AP} AP \Rightarrow V \cup M_B(AP) \).
    By \((2)\), \((3)\) and rule \([\text{read-A}]\), we have \( V_1 \subseteq V_2 \).
Lemma 6 [Removing history is monotonic]
If \( B \vdash M \subseteq M' \), then \( B \vdash (M - A) \subseteq (M' - A) \).

Proof: Trivial induction on \( \subseteq_M \) rule

Lemma 7 [The shift operation is monotonic]
If \( M_1 \subseteq_B M_2 \), then \( \text{shift}(B, M_1, AP) \subseteq_B \text{shift}(B, M_2, AP) \).

Proof: Induction on the derivation of \( \subseteq_M \)

- Case \( \subseteq_M(1) \):
  By assumption, we have (1) \( \text{shift}(B, M_1, AP) = M_2 \). By (1) and rule \( \subseteq_M(1) \), we have \( \text{shift}(B, M_1, AP) = M_2 \subseteq_B \text{shift}(B, M_2, AP) \).

- Case \( \subseteq_M(2) \):
  By assumption, we have (1) \( \text{shift}(B, M_1, AP) \subseteq_B M_2 \). By rule \( \subseteq_M(1) \), we have (2) \( M_2 \subseteq \text{shift}(B, M_2, AP) \). By (1) and (2), we have \( \text{shift}(B, M_1, AP) \subseteq_B \text{shift}(B, M_2, AP) \).

- Case \( \subseteq_M(3) \):
  By assumption,
  (1) \( M_1 = M'_1[AP' \mapsto V_1] \) and \( M_2 = M'_2[AP' \mapsto V_2] \)
  (2) \( M'_1 \subseteq_B M'_2 \) and \( V_1 \subseteq V_2 \)

  - Case \( AP = AP' \):
    By assumption, we have (3) \( \text{shift}(B, M_1, AP) = M_1 \) and \( \text{shift}(B, M_2, AP) = M_2 \). By (3) and hypothesis, we have \( \text{shift}(B, M_1, AP) = M_1 \subseteq_B M_2 \subseteq_B \text{shift}(B, M_2, AP) \).

  - Case \( AP \neq AP' \):
    By (2) and I.H., we have (3) \( \text{shift}(B, M'_1, AP) = (M'_1 - AP)[AP \mapsto V] \subseteq_B \text{shift}(B, M'_2, AP) = (M'_2 - AP)[AP \mapsto V'] \). By (2), (3) and rule \( \subseteq_M(3) \), we have (4) \( M''_1 = (M'_1 - AP)[AP \mapsto V][AP' \mapsto V_1] \subseteq_B (M'_2 - AP)[AP \mapsto V'][AP' \mapsto V_2] = M''_2 \). By (4), I.H. and definition of shift,
we have (5) shift\((B, M''_1, AP) = (M'_1 - AP)[AP' \mapsto V_1][AP \mapsto V_3] \subseteq_B (M'_2 - AP)[AP' \mapsto V_2][AP \mapsto V_4] = \text{shift}(B, M''_2, AP)\) where \((V_3, V_4) = \text{if } B \vdash AP \# AP' \text{ or collapsed}(\alpha, AP) \text{ then } (V, V') \text{ else } (V \cup V_1, V' \cup V_2). \)

By (1), (3), (5) and definition of shift, \(\text{shift}(B, M_1, AP) = (M'_1 - AP)[AP' \mapsto V_1][AP \mapsto V_3] \subseteq_B (M'_2 - AP)[AP' \mapsto V_2][AP \mapsto V_4] = \text{shift}(B, M_2, AP)\) is satisfied.

**Lemma 8** [ Update rule is monotonic ]

If (h1) \(B, M_1, V_1 \vdash_u A_1 \Rightarrow M_{O1}, \) (h2) \(B, M_2, V_2 \vdash_u A_2 \Rightarrow M_{O2}, \) (h3) \(B \vdash M_1 \subseteq_M M_2, \) and (h4) \(V_1 \subseteq V_2, A_1 \subseteq A_2, \) then \(B \vdash M_{O1} \subseteq_M M_{O2}.\)

**Proof:** Induction on the derivation of \(\subseteq_M\)

- **Case** \(A_1 = \bot:\) obvious by rule [update-\(\bot\)]

- **Case** \(A_1 = \{AP\}, \) unique(\(\alpha, AP\)) and \(A_2 = \{AP\): By assumption and rule [update-s], we have (1) \(M_{O1} = (M_1 - \{AP\})[AP \mapsto V_1]\) and (2) \(M_{O2} = (M_2 - \{AP\})[AP \mapsto V_2].\) By (h3), lemma 7 and definition of shift, we have (3) \(\text{shift}(B, M_1, AP) = (M_1 - \{AP\})[AP \mapsto V'_1] \subseteq_B \text{shift}(B, M_2, AP) = (M_2 - \{AP\})[AP \mapsto V'_2].\) By (3) and \([\subseteq_M (3)]\), we have (4) \((M_1 - \{AP\}) \subseteq_B (M_2 - \{AP\}).\) By (4), (h4), (1), (2) and rule \([\subseteq_M (3)],\) we have \(M_{O1} \subseteq_B M_{O2}.\)

- **Case** \(A_1 = \{AP\}, \) unique(\(\alpha, AP\)) and \(A_2 = \{AP, AP_1, \ldots, AP_n\): By assumption and rule [update-s], we have (1) \(M_{O1} = (M_1 - \{AP\})[AP \mapsto V_1].\) By assumption and rule [update-w], we have (2) \(M_{O2} = (M_2 - \{AP, AP_1, \ldots, AP_n\})[AP_1 \mapsto V_2 \cup V_1' \ldots [AP_n \mapsto V_2 \cup V_n'][AP \mapsto V_2 \cup V']\) where \(\text{shift}(B, M_2, AP) = (M_2 - \{AP_i\})[AP_i \mapsto V_i'][A P \mapsto V_2][AP \mapsto V'],\) and \(\text{shift}(B, M_2, AP) = (M_2 - \{AP\})[AP \mapsto V].\) By (h3), lemma 7 and definition of shift, we have (3) \(\text{shift}(B, M_1, AP) = (M_1 - \{AP\})[AP \mapsto V'] \subseteq_B \text{shift}(B, M_2, AP) = (M_2 - \{AP\})[AP \mapsto V_2'].\) By \([\subseteq_M (1)]\), we have (4) \(\text{shift}(B, M_2, AP) \subseteq_B \text{shift}(B, M_2, \{AP_1, \ldots, AP_n, AP\}) \subseteq_B \text{shift}(B, M_2, \{AP_1, \ldots, AP_n, AP\})[AP_1 \mapsto V_1'] \ldots [AP_n \mapsto V_n'][AP \mapsto V_2'].\) By (3) and (4), we have (5) \((M_1 - \{AP\})[AP \mapsto V'_1] \subseteq_B (M_2 - \{AP_1, \ldots, AP_n, AP\})[AP_1 \mapsto V'_1] \ldots [AP_n \mapsto V_n'][AP \mapsto V_2'].\) By (5) and \([\subseteq_M (3)],\) we have (6) \((M_1 -
Lemma 9 [ Resolving is monotonic ]

If \( B \models M_1 \subseteq_B M_2 \), \( B, M_1 \models_{\text{res}} AP \models V_1 \), and \( B, M_2 \models_{\text{res}} AP \models V_2 \), then \( V_1 \subseteq V_2 \).

Proof: Trivial induction with lemma 5

Lemma 10 [ Memory join is monotonic ]

If \( (h1) \) \( B \models M_1 \subseteq_B M_2 \) and \( B \models M_3 \subseteq_B M_4 \), then \( M_1 \sqcup_B M_3 \subseteq B M_2 \sqcup M_4 \).

Proof: We prove the case such that \( M_3 = M_4 \), since the dual case can be proved in the same way. Therefore, the modified lemma is “If \( M_1 \subseteq_B M_2 \), then \( (M_1 \sqcup_B M_3) \subseteq (M_2 \sqcup_B M_3) \)”
• Case $M_1 \subseteq_B M_2$ by $\subseteq_M (1)$:

By assumption, we have (1) $M_2 = \text{shift}(B, M_1, AP)$.
By $\subseteq_M (1)$, we have (2) $M_1 \subseteq_B \text{shift}(B, M_1, AP)$.
By (2) and I.H., we have (3) $M_1 \cup_B M_3 \subseteq_B \text{shift}(B, M_1, AP) \cup_B M_3$.
By (3), and (1), we have $M_1 \cup_B M_3 \subseteq_B M_2 \cup_B M_3$.

• Case $M_1 \subseteq_B M_2$ by $\subseteq_M (2)$:

By assumption and definition of shift, we have (1) $M_2 = M'_2[AP \mapsto V]$ and (2) $\text{shift}(B, M_1, AP) = (M_1 - \{AP\})[AP \mapsto V'] \subseteq_B M'_2[AP \mapsto V] = M_2$.
By (2) and $\subseteq_M (3)$, we have (3) $V' \subseteq V$ and $(M_1 - \{AP\}) \subseteq_B M'_2$.

- Case $M_3 = \epsilon$:

By (2), assumption, rule $\sqcup_M (2)$ and definition of shift, we have (4) $\text{shift}(B, M_1, AP) \cup_B M_3 = (M_1 - AP)[AP \mapsto V'] \sqcup_B \epsilon = ((M_1 - AP) \cup_B \epsilon)[AP \mapsto V' \cup \{\alpha(AP.*)\}]$.
By (1), assumption, rule $\sqcup_M (2)$ and definition of shift, we have (5) $M_2 \cup_B M_3 = M'_2[AP \mapsto V] \sqcup_B \epsilon = (M'_2 \cup_B \epsilon)[AP \mapsto V \cup \{\alpha(AP.*)\}]$.
By (3) and I.H., we have (6) $(M_1 - AP) \sqcup_B \epsilon \subseteq_B (M'_2 \cup_B \epsilon)$.
By (3), we have (7) $V' \cup \{\alpha(AP.*)\} \subseteq V \cup \{\alpha(AP.*)\}$.
By (5), (6), (7), (4), (5), and $\subseteq_M (3)$, we have (8) $\text{shift}(B, M_1, AP) \cup_B M_3 \subseteq_B M_2 \cup_B M_3$.
By $\subseteq_M (1)$, we have (9) $M_1 \subseteq_B \text{shift}(B, M_1, AP)$.
By (9) and I.H., we have (10) $M_1 \cup_B M_3 \subseteq_B \text{shift}(B, M_1, AP) \cup_B M_3$.
By (10), and (8), we have $M_1 \cup_B M_3 \subseteq_B M_2 \cup_B M_3$.

- Case $M_3 = M'_3[AP \mapsto V'']$: similar proof with the first subcase ($M_3 = \epsilon$)

- Case $M_3 = M'_3[AP' \mapsto V'''$]

By (1), assumption, rule $\sqcup_M (3)$ and definition of shift, we have (4) $M_1 \cup_B M_3 = M_1 \cup_B M'_3[AP' \mapsto V''] = \text{shift}(B, M_1, AP') \cup_B M'_3[AP' \mapsto V''] = (M_1 - \{AP'\})[AP' \mapsto V_1] \sqcup_B M'_3[AP' \mapsto V''] = ((M_1 - \{AP'\}) \cup_B M'_3)[AP \mapsto V'' \cup V_1]$.
By (1), assumption, rule $\sqcup_M (3)$ and definition of shift, we have (5) $M_2 \cup_B M_3 = M_2 \cup_B M'_3[AP \mapsto V''] = \text{shift}(B, M_2, AP') \cup_B M'_3[AP' \mapsto V''] = \text{shift}(B, M'_2[AP \mapsto V], AP') \cup_B M'_3[AP' \mapsto V'''] = (M'_2[AP \mapsto V] - \{AP'\})[AP' \mapsto V_2] \sqcup M'_3[AP' \mapsto V''] = (((M'_2[AP \mapsto V] - \{AP'\}) \cup_B M'_3)[AP' \mapsto V'' \cup V_2]$.
By (4), (5), hypothesis $(M_1 \subseteq_B M_2)$ and lemma 7, we have (6) $\text{shift}(B, M_1, AP') =
(M_1 - \{ AP' \})[AP' \mapsto V_1] \subseteq_B \text{shift}(B, M_2, AP') = \text{shift}(B, M'_2[AP \mapsto V], AP') = (M'_2[AP \mapsto V] - \{ AP' \})[AP' \mapsto V_2]. \text{ By (6) and } [\subseteq_M (3)], \text{ we have (7) } (M_1 - \{ AP' \}) \subseteq_B (M'_2[AP \mapsto V] - \{ AP' \}) \text{ and (8) } V_1 \subseteq V_2. \text{ By (7) and I.H., (9) } ((M_1 - \{ AP' \}) \sqcup_B M'_3) \subseteq_B ((M'_2[AP \mapsto V] - \{ AP' \}) \sqcup_B M'_3). \text{ By (9), (8), (4), (5) and } [\subseteq_M (3)], \text{ we have } M_1 \sqcup_B M_3 \subseteq_B M_2 \sqcup_B M_3.

- Case $M_1 \subseteq_B M_2$ by $[\subseteq_M (3)]$:
  
  By assumption and definition of shift
  (1) $M_1 = M'_1[AP \mapsto V]$ and $M_2 = M'_2[AP \mapsto V']$
  (2) $V \subseteq V'$ and $M'_1 \subseteq_B M'_2$

  - Case $M_3 = \epsilon$:
    
    By (1), assumption, rule $[\sqcup_M (2)]$ and definition of shift, we have (3) $M_1 \sqcup_B M_3 = M'_1[AP \mapsto V] \sqcup_B \epsilon = M'_1[AP \mapsto V] \sqcup_B \text{shift}(B, \epsilon, AP) = (M'_1 \sqcup_B \epsilon)[AP \mapsto V \cup \{ \alpha(\text{AP}.* \})], \text{ and (4) } M_2 \sqcup_B M_3 = M'_2[AP \mapsto V'] \sqcup_B \epsilon = M'_2[AP \mapsto V'] \sqcup_B \text{shift}(B, \epsilon, AP) = (M'_2 \sqcup_B \epsilon)[AP \mapsto V' \cup \{ \alpha(\text{AP}.* \}).$

    By (2) and I.H., we have (5) $(M'_1 \sqcup_B \epsilon) \subseteq_B (M'_2 \sqcup_B \epsilon).$ By (2), we have (6) $V \sqcup \{ \alpha(\text{AP}.* \}) \subseteq V' \sqcup \{ \alpha(\text{AP}.* \}).$ By (5), (6), (3), (4) and $[\subseteq_M (3)],$ we have $M_1 \sqcup_B M_3 \subseteq_B M_2 \sqcup_B M_3.$

  - Case $M_3 = M'_3[AP \mapsto V'']$: similar proof with the first subcase ($M_3 = \epsilon$)

- Case $M_3 = M'_3[AP' \mapsto V'']$:
  
  By (1), assumption, rule $[\sqcup_M (3)]$ and definition of shift, we have (3) $M_1 \sqcup_B M_3 = M'_1[AP \mapsto V] \sqcup_B M'_3[AP' \mapsto V''] = \text{shift}(B, M_1, AP') \sqcup_B M'_3[AP' \mapsto V''] = (M'_1[AP \mapsto V] - \{ AP' \})[AP' \mapsto V_1] \sqcup_B M'_3[AP' \mapsto V''] = (((M'_1[AP \mapsto V] - \{ AP' \}) \sqcup_B M'_3)[AP \mapsto V''] \sqcup V_1), \text{ and (4) } M_2 \sqcup_B M_3 = M'_2[AP \mapsto V'] \sqcup_B M'_3[AP' \mapsto V''] = \text{shift}(B, M_2, AP') \sqcup_B M'_3[AP' \mapsto V''] = (M'_2[AP \mapsto V'] - \{ AP' \})[AP' \mapsto V_2] \sqcup_B M'_3[AP' \mapsto V''] = (((M'_2[AP \mapsto V'] - \{ AP' \}) \sqcup_B M'_3)[AP \mapsto V''] \sqcup V_2). \text{ By (3), (4), hypothesis } (M_1 \subseteq_B M_2) \text{ and lemma 7, we have (5) } \text{shift}(B, M_1, AP') = (M'_1[AP \mapsto V] - \{ AP' \})[AP' \mapsto V_1] \subseteq_B \text{shift}(B, M_2, AP') = \text{shift}(B, M'_2[AP \mapsto V], AP') = (M'_2[AP \mapsto V'] - \{ AP' \})[AP' \mapsto V_2]. \text{ By (5) and } [\subseteq_M (3)], \text{ we have (6) } (M'_1[AP \mapsto V] - \{ AP' \}) \subseteq_B (M'_2[AP \mapsto V'] - \{ AP' \}),$ and
Lemma 11 [Expression typing is monotonic]
If \( B \vdash M_1 \sqsubseteq M_2 \), \( B, M_1 \vdash e \Rightarrow (M_{O1}, V_1) \), and \( B, M_2 \vdash e \Rightarrow (M_{O2}, V_2) \), then \( B \vdash M_{O1} \sqsubseteq M_{O2} \) and \( V_1 \subseteq V_2 \).

Proof: Trivial induction with lemma 5, 8, 9, 10.

Lemma 12 [Stack variable]
If \( B, M \vdash e \Rightarrow M_o, V_o, B, M \vdash AP \xrightarrow{\bar{y}} V_i \), \( \forall AP \in M : \text{prefix}(AP) \neq x_i \) and \( \bar{y} \) new, then \( B, M[\bar{y} \mapsto V_x] \vdash e[\bar{y}/\bar{x}] \Rightarrow (M_o - \bar{x})[\bar{y} \mapsto V'_x], V_o[\bar{y}/\bar{x}] \).

Proof: (1) Because \( \bar{x} \) does not appear in \( e[\bar{y}/\bar{x}] \), its value does not change. (2) Because \( \bar{y} \) is new (no alias), its history position does not affect valuation. (3) Because \( x_i \) is changed to \( \bar{y} \), every update on \( x_i \) in the old expression decides \( \bar{y} \)'s value. (4) There is no way to directly access deep location like \( x.* \). And every indirect access using \( \bar{x} \) is changed to \( \bar{y} \). By (1), (2), (3), (4), it is trivial to prove this lemma.

Lemma 13 [Removing preserves join typing]
If \( M_1 \sqcup_B M_2 = M_3 \), then \( M_1 - A \sqcup_B M_2 - A = M_3 - A \).

Proof: Trivial induction on \( \sqcup_B \) rule.

Lemma 14 [Smaller input context preserves basic typing]
If \( B \vdash B' \) and \( B' \vdash B'' \), then \( B \vdash B'' \).

Proof: Trivial induction on the derivation of \( B \vdash B' \).

Lemma 15 [Substitution preserves input context typing]
If (h1) \( \text{known}(M) \), (h2) \( \forall AP \in \text{Ran}(\alpha) : S(AP) = \cup \{ V \mid \alpha(AP') = AP \} \), \( \emptyset, M \vdash_{res} AP'[\bar{y}/\bar{x}] \Rightarrow V \} \), (h3) \( \alpha \vdash S \), (h4) \( \emptyset \vdash B'[S] \), and (h5) \( B' \vdash B'' \), then \( \emptyset \vdash B''[S] \).

Proof: Trivial induction on the derivation of \( B' \vdash B'' \).
Lemma 16 [ Substitution preserves shift order ]

If (h1) known(M), (h2) \( \forall AP \in \text{Ran}(\alpha) : S(AP) = \cup \{V \mid \alpha(AP') = AP\} \), (h3) \( \alpha \vdash S \), and (h4) \( \emptyset \not\vdash B'[S] \), then \( M \cdot M_1[S] \sqsubseteq M \cdot \text{shift}(B', M_1, AP)[S] \).

Proof: Induction on the derivation of shift

- Case \( M_1 = \epsilon \):
  By assumption, we have (1) \( \text{shift}(B', M_1, AP) = [AP \mapsto \{\alpha(AP \ast)\}] \). By (1), we have (2) \( M \cdot \text{shift}(B', M_1, AP)[S] = M \cdot [AP[S] \mapsto \{\alpha(AP \ast)\}[S]] \). By (2) and (h2), we have (3) \( \forall AP' \) s.t. \( \alpha(AP') = AP : M(AP') \in \{\alpha(AP \ast)\}[S] \).
  By assumption, (3) and rules [update-s], [update-w], we have (4) \( M \cdot M_1[S] = M \sqsubseteq M \cdot [AP[S] \mapsto \{\alpha(AP \ast)\}[S]] \).
  By (4), and (2), we have \( M \cdot M_1[S] \sqsubseteq M \cdot \text{shift}(B', M_1, AP)[S] \).

- Case \( M_1 = M'_1[AP \mapsto V] \):
  By assumption, we have (1) \( \text{shift}(B', M_1, AP) = M'_1[AP \mapsto V] = M_1 \). By assumption and (1), we have \( M \cdot M_1[S] = M \cdot \text{shift}(B', M_1, AP)[S] \sqsubseteq M \cdot \text{shift}(B', M_1, AP)[S] \).

- Case \( M_1 = M'_1[AP \mapsto V'] \) and \( B' \vdash AP \neq AP' \):
  By assumption, we have (1) \( \text{shift}(B', M_1, AP) = M''_1[AP' \mapsto V'][AP \mapsto V''] \)
  where \( \text{shift}(B', M'_1, AP) = M''_1[AP \mapsto V''] \).
  By assumption and (h4), we have (2) \( AP[S] \cap AP'[S] = \emptyset \).
  Then we finally have \( M \cdot M_1[S] \sqsubseteq M \cdot \text{shift}(B', M'_1, AP)[S] \) as follows.

  \[ M \cdot M_1[S] \text{ by assumption and definition of substitution} \]
  \[ = M \cdot M'_1[S] \cdot [AP'[S] \mapsto V'[S]] \text{ by I.H.} \]
  \[ \subseteq M \cdot \text{shift}(B', M'_1, AP)[S] \cdot [AP'[S] \mapsto V'[S]] \text{ by (1)} \]
  \[ = M \cdot M''_1[S] \cdot [AP[S] \mapsto V''[S]] \cdot [AP'[S] \mapsto V'[S]] \text{ by (2), it is trivial to show} \]
  \[ \subseteq M \cdot M''_1[S] \cdot [AP'[S] \mapsto V'[S]] \cdot [AP[S] \mapsto V''[S]] = M \cdot \text{shift}(B', M_1, AP)[S] \]

- Case \( M_1 = M'_1[AP \mapsto V'] \) and collapsed(\( \alpha, AP \)): similar induction with the third case
• Case $M_1 = M_1'[AP' \rightarrow V']$, $\neg B' \vdash AP \# AP'$ and unique$(\alpha, AP)$:

By assumption, we have (1) shift$(B', M_1, AP) = M_1''[AP' \rightarrow V']$ where shift$(B', M_1, AP) = M_1''[AP \rightarrow V'']$ and $V'' = V' \cup V'''$. Then we finally have $M \cdot M_1[S] \sqsubseteq M \cdot \text{shift}(B', M_1, AP)[S]$ as follows.

$M \cdot M_1[S]$ by assumption and definition of substitution

$= M \cdot M_1'[S] \cdot [AP'[S] \rightarrow V'[S]]$ by I.H.

$\sqsubseteq M \cdot \text{shift}(B', M_1, AP)[S] \cdot [AP'[S] \rightarrow V'[S]]$ by (1)

$= M \cdot M_1''[S] \cdot [AP[S] \rightarrow V''[S]] \cdot [AP'[S] \rightarrow V'[S]]$ by (1), it is trivial to show

$\sqsubseteq M \cdot M_1'[S] \cdot [AP'[S] \rightarrow V'[S]] \cdot [AP[S] \rightarrow V''[S]] = M \cdot \text{shift}(B', M_1, AP)[S]$

Lemma 17 [ Substitution preserves $AP$ read typing ]

If (h1) known$(M)$, (h2) $\forall AP \in \text{Ran}(\alpha) : S(AP) = \cup\{V \mid \alpha(AP') = AP\}$, $\emptyset, M \vDash_{\text{res}} AP'[\bar{y}/\bar{x}] \Rightarrow V\}$, (h3) $\alpha \vdash S$, (h4) $\emptyset \vdash B'[S]$, and (h5) $B', M_1 \vdash_{AP} AP \Rightarrow V_o$, then $\emptyset, M \cdot M_1[S] \vdash_{AP} AP[S] \Rightarrow V_o[S]$.

Proof: Induction on the derivation of (h5)

• Case [read-⊥]: trivial

• Case [read-ε]:

By assumption and (h5), we have (1) $M_1 = \epsilon$ and (2) $V_o = \alpha(AP,*)$. By (1), we have (3) $M \cdot M_1[S] = M$. By definition of $\alpha$, we have (4) “If $\alpha(AP') = AP$, then $\alpha(AP',*) = \alpha(AP,*)$”. By (4) and (h2), we have (5) $\alpha(AP,*)[S] = \cup\{V' \mid \alpha(AP',*) = \alpha(AP,*)\}$, $\emptyset, M \vdash_r (AP,*)[\bar{y}/\bar{x}] \Rightarrow V'$. By (5) and rule [resolve(2)], we have (6) $\alpha(AP,*)[S] = \cup\{V' \mid \alpha(AP',*) = \alpha(AP,*)\}$, $\emptyset, M \vdash_r AP'[\bar{y}/\bar{x}] \Rightarrow V$, $\emptyset, M \vdash_r V \Rightarrow V'$. By (h2), (h3), (4), (6) and rule [read-A], we have $B, M \vdash_r AP[S] \Rightarrow \alpha(AP,*)[S]$.

• Case [read=]:

By assumption and (h5), we have (1) $M_1 = M_2[AP \rightarrow V_o]$ and (2) unique$(\alpha, AP)$. By (2) and (h3), we have (3) $AP[S] = \{AP'\}$.

– Case unique$(AP')$:

By assumption, (1), and rule [update-s], we have (4) $M \cdot M_1[S] = M \cdot$
\[M_2[S] \cdot [AP[S] \mapsto V_o[S]] = (M \cdot M_2[S] - \{AP'[S]\})[AP' \mapsto V_o[S]].\]

By (3), (4) and rule [read-\#], we have \(\emptyset, M \cdot M_1[S] \vdash_r AP[S] \Rightarrow V_o[S].\)

- Case \([AP']\):

By assumption, (1), (2), (3), and assumption 9 in Section A.1, we have (4) \(M \cdot M_1[S] = M \cdot M_2[S] \cdot [AP[S] \mapsto V_o[S]] = (\text{remove-recent}(M[AP' \mapsto M(AP') \cdot M_2[S], AP'])[AP' \mapsto V_o[S]).\)

By (3), (4) and assumption 9 in Section A.1, we have \(\emptyset, M \cdot M_1[S] \vdash_r AP[S] \Rightarrow V_o[S].\)

- Case [read-\#]:

By assumption and (h5), we have (1) \(M_1 = M_2[AP_1 \mapsto V_1],\) (2) \(B' \vdash AP\#AP_1,\) and (3) \(B', M_2 \vdash AP \Rightarrow V_o.\)

By (2) and (h4), we have (4) \(\emptyset \vdash AP[S]\#AP_1[S]\) (i.e., \(AP[S] \cap AP_1[S] = \emptyset).\)

By (1) and definition of \(\cdot,\) we have (5) \(M \cdot M_1[S] = M \cdot M_2[S] \cdot [AP_1[S] \mapsto V_1[S]].\)

By (5) and update rule, we have (6) \(M \cdot M_1[S] = (M \cdot M_2[S] - AP_1[S])[AP'_1 \mapsto V'_1] \cdots [AP'_n \mapsto V'_n]\) where \(AP_1[S] = \{AP'_1, \cdots, AP'_n\}.\)

By (6), (4) and rule [read-A], [read-\#], we have (7) \(\emptyset, M \cdot M_1[S] \vdash_r AP[S] \Rightarrow V'\) where \(\emptyset, M \cdot M_2[S] - AP_1[S] \vdash_r AP[S] \Rightarrow V'.\)

By (3) and I.H., we have (8) \(\emptyset, M \cdot M_2[S] \vdash_r AP[S] \Rightarrow V_o[S].\)

By (7), (8), (4), and (h1), we have (9) \(V_o[S] = V'.\) By (9), we have \(\emptyset, M \cdot M_1 \vdash_r AP[S] \Rightarrow V_o[S].\)

- Case [read-safe]:

By assumption and (h5), we have (1) \(M_1 = M_2[AP'[S] \mapsto V'[S]],\) (2) \(B', M_2 \vdash AP \Rightarrow V,\) and (3) \(V_o = V \cup V'.\)

By (2) and I.H., we have (4) \(\emptyset \vdash M \cdot M_2[S] \vdash_r AP[S] \Rightarrow V[S].\)

By (4), read rule, subsumption rule, we have (5) \(\emptyset \vdash M \cdot M_2[S] - AP'[S] \vdash_r AP[S] \Rightarrow V[S].\)

- Case \(AP'[S] = \{AP_1\}\) and unique(\(\alpha, AP_1\)): By assumption, (1) and rule [update-s], we have (6) \(M \cdot M_1[S] = M \cdot M_2[S] \cdot [\{AP_1\} \mapsto V'[S]] = (M \cdot M_2[S] - AP'[S])[AP_1 \mapsto V'[S]].\)

By (5), (6), rule [read-safe], we have (7) \(\emptyset, M \cdot M_1[S] \vdash_r AP[S] \Rightarrow V[S] \cup V'[S].\)

- Case \(AP'[S] = \{AP_1\},\) collapsed(\(\alpha, AP_1\)) and \(AP_1 \notin AP[S]:\)

By assumption, (1) and rule [update-w], we have (8) \(M \cdot M_1[S] = M \cdot M_2[S] \cdot [\{AP_1\} \mapsto V'[S]] = (M \cdot M_2[S] - AP'[S])[AP_1 \mapsto V'[S] \cup V''].\)

where shift(\(\emptyset, M \cdot M_2[S], AP_1\)) = \(M \cdot M_2[S] - \{AP_1\}[AP_1 \mapsto V'\). By assumption and known(\(M\) of (h1), we have (9) \(\emptyset \vdash AP[S]\#AP_1.\) By (8), (9), (5)
Lemma 18 [ Substitution preserves A read typing ]

If (h1) known(M), (h2) \( \forall AP \in \text{Ran}(\alpha) : S(AP) = \cup\{V \mid \alpha(AP') = AP\} \), \( \emptyset, M \vdash_{\text{res}} AP'[\overline{y}/\overline{x}] \Rightarrow V \}, (h3) \alpha \vdash S \), (h4) \( \emptyset \vdash B'[S] \), and (h5) \( B', M_1 \vdash A \Rightarrow V_o \), then \( \emptyset, M \cdot M_1[S] \vdash_{\text{r}} A[S] \Rightarrow V_o[S] \).

Proof:
Let (1) \( A[S] = \{AP_1, \ldots, AP_n\} \), (2) \( \forall AP_j \in A : B', M_1 \vdash_{\text{AP}} AP_j \Rightarrow V_j \) where \( V_o = \cup_j V_j \). By (2) and lemma 17, we have (3) \( \forall AP_j \in A : \emptyset, M \cdot M_1[S] \vdash_{\text{r}} AP_j[S] \Rightarrow V_j[S] \) where \( V_o[S] = V_1[S] \cup \cdots \cup V_n[S] \). By (3) and rule [read-A], we have (4) \( \forall AP_k \in AP_j[S] : \emptyset, M \cdot M_1[S] \vdash_{\text{AP}} AP_k[S] \Rightarrow V_{jk} \) where \( V_j[S] = V_{j1} \cup \cdots \cup V_{jm} \). By (4), (1), (3) and rule [read-A], we have \( \emptyset, M \cdot M_1[S] \vdash_{\text{r}} A[S] \Rightarrow V_o[S] \).

Lemma 19 [ Substitution preserves update typing ]

If (h1) known(M), (h2) \( \forall AP \in \text{Ran}(\alpha) : S(AP) = \cup\{V \mid \alpha(AP') = AP\} \), \( \emptyset, M \vdash_{\text{res}} AP'[\overline{y}/\overline{x}] \Rightarrow V \}, (h3) \alpha \vdash S \), (h4) \( \emptyset \vdash B'[S] \), and (h5) \( B', M_1, V_2 \vdash_{u} V_1 \Rightarrow M_o \), then \( \emptyset, M \cdot M_1[S], V_2[S] \vdash_{u} V_1[S] \Rightarrow M \cdot M_o[S] \).

Proof: Induction on the derivation of (h5)
• Case [update-w] and $V_1 = \{AP\}$:
  By assumption and (h5), we have (1) $V_1 = \{AP\}$ and unique($\alpha, AP$) and (2) $M_o = (M_1 - \{AP\})[AP \mapsto V_2]$. By (1), and (h3), we have (3) $AP[S] = \{AP\}$.

  - Case unique($\alpha, AP'$) and $[AP \mapsto V'] \notin M_1$:
    By assumption, and (2), we have $M \cdot M_o[S] = M \cdot (M_1 - \{AP\})[S] \cdot [AP[S] \mapsto V_2[S]] = M \cdot M_1[S] \cdot [AP[S] \mapsto V_2[S]]$.

  - Case unique($\alpha, AP'$) and $M_1 = M'_1[AP \mapsto V'][M'_1]$:
    By assumption, (2), (3) and rule [update-s], we have
    \[
    M \cdot M_o[S] = M \cdot (M_1 - \{AP\})[S] \cdot [\{AP\} \mapsto V_2[S]] = M \cdot M'_1[S] \cdot [\{AP\} \mapsto V_2[S]]
    \]
    since intermediate side effect on $AP'$ is killed by last update
    \[
    = M \cdot M'_1[S] \cdot [\{AP\} \mapsto V''[S]] \cdot M''_1[S] \cdot [\{AP\} \mapsto V_2[S]]
    \]
    \[
    = M \cdot M_1[S] \cdot [\{AP\} \mapsto V_2[S]]
    \]
    \[
    = (M \cdot M_1[S] - \{AP\})[AP' \mapsto V_2[S]]
    \]
    By (1), (3) and rule [update-s], we have (5) $(M \cdot M_1[S]) \cdot [V_1[S] \mapsto V_2[S]] = (M \cdot M_1[S] - \{AP\})[AP' \mapsto V_2[S]]$. By (4), and (5), we have
    $\emptyset, M \cdot M_1[S], V_2[S] \vdash_u V_1[S] \Rightarrow M \cdot M_o[S]$. 

  - Case collapsed($\alpha, AP'$) and $M_1 = M'_1[AP \mapsto V'][M'_1]$:
    $M \cdot M_1[S] \cdot [V_1[S] \mapsto V_2[S]] = M \cdot M_1[S] \cdot [AP' \mapsto V_2[S]]$
    by (1), (3), assumption and assumption 9 in Section A.1
    \[
    = M[AP' \mapsto M(AP')] \cdot M'_1[S] \cdot [\{AP\} \mapsto V''[S]] \cdot M''_1[S] \cdot [\{AP\} \mapsto V_2[S]]
    \]
    since intermediate side effect on $AP'$ is killed by last update
    \[
    = M \cdot M'_1[S] \cdot M''_1[S] \cdot [AP' \mapsto V_2[S]] \quad \text{by assumption}
    \]
    \[
    = M \cdot (M_1 - AP)[S] \cdot [AP' \mapsto V_2[S]] \quad \text{by (2), (3)}
    \]
    \[
    = M \cdot M_o[S]
    \]
    - Case collapsed($\alpha, AP'$) and $[AP \mapsto V'] \notin M_1$: similar proof with the third subcase.

• Case [update-w] and $V_1 = \{AP\}$:
  By assumption and (h5), we have (1) collapsed($\alpha, AP$), and (2) $M_o = (M_1 - \{AP\})[AP \mapsto V_2 \cup V']$ where shift($B', M_1, AP$) = $(M_1 - \{AP\})[AP \mapsto V']$. 

70
Lemma 20 [ Substitution preserves access path resolving ]

If (h1) known(M), (h2) ∀AP ∈ Ran(α) : S(AP) = ∪{V | α(AP) = AP},
0, M ⊨_{res} AP′[\bar{y}/\bar{x}] ⇒ V}, (h3) α ⊨ S, (h4) 0 ⊨ B′[S] , and (h5) B′, M ⊨_{res} AP ⇒ V₀,
then 0, M • M₁[S] ⊨_{AP} AP[S] ⇒ V₀[S].

Proof: Trivial induction with lemma 17, 18
Lemma 21 [Substitution preserves $\sqcup_M$ typing]

If (h1) known$(M)$, (h2) $\forall AP \in \text{Ran}(\alpha) : S(AP) = \cup \{ V \mid \alpha(AP') = AP \}$, $\emptyset, M \vdash_{res} AP'[\overline{\alpha}/\overline{x}] \Rightarrow V \}$, (h3) $\alpha \vdash S$, (h4) $\emptyset \vdash B'[S]$, and (h5) $B' \vdash M_1 \sqcup M_2 \Rightarrow M_o$, then $\emptyset \vdash M \cdot M_1[S] \sqcup_M M \cdot M_2[S] \Rightarrow M \cdot M_o[S]$.

Proof: Induction on the derivation of $B' \vdash M_1 \sqcup M_2 \Rightarrow M_o$

- Case (h5) is typed by $[\sqcup_M(1)]$:
  By assumption, we have (1) $M_1 = M'_1[AP \mapsto V]$ and $M_2 = M'_2[AP \mapsto V']$, (2) $B' \vdash M_1 \sqcup M M'_2 \Rightarrow M_2$, and (3) $M_o = M_{12}[AP \mapsto V \cup V']$. By (2) and I.H., we have (4) $\langle M \cdot M_1[S] \sqcup M \cdot M'_2[S] = M \cdot M_{12}[S]$. Let (5) $\langle AP[S] = A$. By (4), (5) and definition of $\sqcup$, it is trivial to show (6) $(\langle M \cdot M'_1[S] - A) \sqcup (\langle M \cdot M'_2[S] - A) = (\langle M \cdot M_{12}[S] - A)$.

- Case $A = \{ AP_1 \}$ and unique$(\alpha, AP_1)$:
  By (1) and rule [update-s], we have (7) $M \cdot M_1[S] = (M \cdot M'_1[S] - A)[AP_1 \mapsto V[S]]$, and (8) $M \cdot M_2[S] = (M \cdot M'_2[S] - A)[AP_1 \mapsto V'[S]]$.
  By (6), (7), (8) and rule $[\sqcup_M(1)]$, we have $M \cdot M_1[S] \sqcup M \cdot M_2[S] = (M \cdot M_{12}[S] - A)[AP_1 \mapsto V[S] \cup V'[S]] = M \cdot M_o[S]$ (by (3), rule [update-s]).

- Case $A = \{ AP_1 \}$ and collapsed$(\alpha, AP_1)$:
  By (1), assumption and rule [update-w], we have (7) $M \cdot M_1[S] = M \cdot M'_1[S] \cdot [AP_1 \mapsto V[S]] = (M \cdot M'_1[S] - AP_1)[AP_1 \mapsto V_1 \cup V[S]]$ where shift$(\emptyset, M \cdot M'_1[S], AP_1) = (M \cdot M'_1[S] - \{AP_1\})[AP_1 \mapsto V_1]$, and (8) $M \cdot M_2[S] = M \cdot M'_2[S] \cdot [AP_1 \mapsto V'[S]] = (M \cdot M'_2[S] - AP_1)[AP_1 \mapsto V_2 \cup V'[S]]$ where shift$(\emptyset, M \cdot M'_2[S], AP_1) = (M \cdot M'_2[S] - \{AP_1\})[AP_1 \mapsto V_2]$. By (3), assumption and rule [update-w], we have (9) $M \cdot M_o[S] = M \cdot M_{12}[S] \cdot [AP_1 \mapsto V[S] \cup V'[S]] = (M \cdot M_{12}[S] - \{AP_1\})[AP_1 \mapsto V[S] \cup V'[S] \cup V_3]$ where shift$(\emptyset, M \cdot M_{12}[S], AP_1) = (M \cdot M_{12}[S] - \{AP_1\})[AP_1 \mapsto V_3]$. By (4), (7), (8), (9) and lemma 7 and $[\sqcup_M (3)]$, we have (10) $V_1 \subseteq V_3 \wedge V_2 \subseteq V_3$. By (6), (7), (8), (9), (10) and $[\sqcup_M (1)]$ and subsumption rule, we have $M \cdot M_1[S] \sqcup M \cdot M_2[S] = (M \cdot M_{12}[S] - A)[AP_1 \mapsto V_1 \cup V_2 \cup V[S] \cup V'[S]] \sqsubseteq (M \cdot M_{12}[S] - A)[AP_1 \mapsto V_3 \cup V[S] \cup V'[S]] = M \cdot M_o[S]$.  

72
– Case \( A = \{ AP_1, \cdots, AP_n \} \):

When unique(\( \alpha, AP_i \)), the proof is similar with the first subcase.
When collapsed(\( \alpha, AP_i \)), the proof is similar with the second subcase.

- Case \( h5 \) is typed by \( \cup_M(2) \), \( AP[S] = \{ AP' \} \) and unique(\( \alpha, AP' \)):

  (1) \( M_0 = M_1 \cup_{B'} M_2 \) by assumption

  \[
  = M'_1[AP \mapsto V] \cup_{B'} M'_2[AP \mapsto V'] \quad \text{where shift}(B', M_2, AP) = M'_2[AP \mapsto V']
  = (M'_1 \cup_{B'} M'_2)[AP \mapsto V \cup V']
  \]

  By (1) and I.H., we have (2) \( M \cdot M'_1[S] \cup M \cdot M'_2[S] = M \cdot (M'_1 \cup_{B'} M'_2)[S] \).

  By (2) and lemma 13, we have (3) \( M \cdot M'_1[S] \cup M \cdot M'_2[S] = AP[S] \).

  By (1) and assumption, we have

  (4) \( M \cdot M_1[S] \cup M \cdot M_2[S] \)

  \[
  = M \cdot M'_1[S] : [AP[S] \mapsto V[S]] \cup M \cdot M'_2[S] : [AP[S] \mapsto V'[S]]
  \]

  by assumption and rule [update-s]

  \[
  = (M \cdot M'_1[S] - AP[S])[AP' \mapsto V[S]] \cup (M \cdot M'_2[S] - AP[S])[AP' \mapsto V'[S]]
  \]

  by (3) and \( \cup_M(1) \)

  \[
  = (M \cdot (M'_1 \cup_{B'} M'_2)[S] - AP[S])[AP' \mapsto V[S] \cup V'[S]]
  \]

  by assumption, (1) and rule [update-s]

  \[
  = M \cdot M_0[S]
  \]

- Case \( h5 \) is typed by \( \cup_M(2) \) and \( AP[S] = \{ AP_1, \cdots, AP_n \} \):

  By assumption and definition of shift and \( \cup_M(1) \), we have (1) \( M_1 \cup_{B'} M_2 = M'_1[AP \mapsto V] \cup_{B'} (shift(B', M_2, AP) = M'_2[AP \mapsto V] \cup_{B'} (M_2 - \{ AP \}))[AP \mapsto V'] = (M'_1 \cup_{B'} (M_2 - \{ AP \}))[AP \mapsto V \cup V'] = M_0 \).

  By (1) and I.H., we have (2) \( M \cdot M'_1[S] \cup M \cdot (M'_2 - \{ AP \})[S] = M \cdot (M'_1 \cup_{B'} (M_2 - \{ AP \}))[S] \).

  By assumption, (1), and rule [update-w], we have (3) \( M \cdot M_1[S] = M \cdot M'_1[S] : [AP[S] \mapsto V[S]] \cup M \cdot M'_2[S] : [AP[S] \mapsto V'[S]] \]

  \[
  = (M \cdot M'_1[S] - AP[S])[AP_1 \mapsto V[S] \cup V_1] \cdots [AP_n \mapsto V[S] \cup V_n] \quad \text{where shift}(\emptyset, M \cdot M'_1[S], AP_i) = M'_1[AP_i \mapsto V_i], \quad \text{and (4) M \cdot shift}(B', M_2, AP)[S] = M \cdot (M_2 - \{ AP \})[S] : [AP[S] \mapsto V'[S]] = (M \cdot (M_2 - \{ AP \}))[S] - AP[S])[AP_1 \mapsto V'[S] \cup V'_1] \cdots [AP_n \mapsto V'[S] \cup V'_n] \quad \text{where shift}(\emptyset, M \cdot (M_2 - \{ AP \}))[S], AP_i) = M''_i[AP_i \mapsto V'_i].

  By (2) and lemma 13, we have (5) \( (M \cdot M'_1[S] - AP[S]) \cup (M \cdot (M_2 - \{ AP \}))[S] = AP[S] \).

  By (5), (3), (4) and \( \cup_M(1) \), we have (6) \( M \cdot M_1[S] \cup M \cdot shift(B', M_2, AP)[S] =\)
((M \cdot M_1[S] - AP[S]) \sqcup (M \cdot (M_2 - \{AP\})[S] - AP[S]))[AP_1 \mapsto V_1'''] \cdots [AP_n \mapsto V_n'''] = (M \cdot (M_1 \sqcup B' (M_2 - \{AP\}))[S] - AP[S])[AP_1 \mapsto V_1'''] \cdots [AP_n \mapsto V_n''']  

where $V_i''' = V[S] \cup V'[S] \cup V_i' \cup V_i'$. By assumption, (1), and rule [update-w], we have (7) $M \cdot M_o[S] = M \cdot (M_1 \sqcup B' (M_2 - \{AP\}))[S] \cdot [AP[S] \mapsto V[S] \cup V'[S]] = (M \cdot (M_1 \sqcup B' (M_2 - \{AP\}))[S] - AP[S])[AP_1 \mapsto V_1'''] \cdots [AP_n \mapsto V_n''']$ where shift($\emptyset, (M \cdot (M_1 \sqcup B' (M_2 - \{AP\}))[S]), AP_i) = M_i'''[AP_i \mapsto V_i''' \wedge V_i''' = V[S] \cup V'[S] \cup V_i'$. By (2), we have (8-1) $M \cdot M_1[S] \sqsubseteq M \cdot (M_1 \sqcup B' (M_2 - \{AP\}))[S]$ and (8-2) $M \cdot (M_2 - \{AP\})[S] \sqsubseteq M \cdot (M_1 \sqcup B' (M_2 - \{AP\}))[S]$. By (8-1), (8-2) and lemma 7, we have (9-1) shift($\emptyset, M \cdot M_1[S], AP_i) = M_i'''[AP_i \mapsto V_i'] \sqsubseteq$ shift($\emptyset, M \cdot (M_1 \sqcup B' (M_2 - \{AP\}))[S], AP_i) = M_i''''[AP_i \mapsto V_i''], and (9-2) shift($\emptyset, M \cdot (M_2 - \{AP\})[S], AP_i) = M_i''''[AP_i \mapsto V_i'] \sqsubseteq$ shift($\emptyset, M \cdot (M_1 \sqcup B' (M_2 - \{AP\}))[S], AP_i) = M_i''''[AP_i \mapsto V_i''']$. By (9-1), (9-2), rule $[simplified_extension(3)]$, we have (10) $V_i \subseteq V_i''' \wedge V_i' \subseteq V_i'''$. By (10), (6), (7), rule $[simplified_extension(3)]$ and subsumption rule, we have (11) $M \cdot M_1[S] \sqcup M \cdot shift(B', M_2, AP)[S] = M \cdot M_o[S]$. By lemma 16, we have (12) $M \cdot M_2[S] \sqsubseteq M \cdot shift(B', M_2, AP)[S]$. By (12), (11) and lemma 10, we have (13) $M \cdot M_1[S] \sqcup M \cdot M_2[S] \sqsubseteq M \cdot (M_1 \sqcup M_2[S] \sqcup M \cdot shift(B', M_2, AP)[S] = M \cdot M_o[S]$. By (13) and subsumption rule, we have $M \cdot M_1[S] \sqcup M \cdot M_2[S] = M \cdot M_o[S]$.

- Case (h5) is typed by $[\sqcup_M(3)]$: dual case with $[\sqcup_M(2)]$

**Lemma 22** [ Postponed update is sound under the substitution ]

*If (h0) known(M), (h1) known($M \cdot M_1[S] \cdot M_2[S'']$) \& known($M \cdot (M_1 \sqcup B' (M_2[S'']) [S]$),
(h2) Dom($S''$) = Dom($S'$) \& AP[$S''$] = AP[$S'$][S], (h3) $\emptyset \vdash B'[S]$, and (h4) $\alpha \vdash S$,
then $M \cdot M_1[S] \cdot M_2[S''] \sqsubseteq_M M \cdot (M_1 \sqcup B' (M_2[S'']) [S]$.)

**Proof:**

(a1) We assume the size of $M_2$ is 1 ($M_2 = [AP \mapsto V]$) since the proof for the size $m$ is trivial by induction. By (h2), and (a1), we have (1) $AP[S''] = AP[S'][S] \sqcap V[S''] = V[S'][S]$. By (a1), and (1), we have (2) $M_1 = M \cdot M_1[S] \cdot M_2[S''] = M \cdot M_1[S] \cdot [AP[S'][S] \mapsto V[S'][S]]$.

- Case $AP[S'] = \{AP_i\}$ and unique($\alpha, AP_i$):

  By assumption and rule [update-s], we have
Lemma 23 [Substitution preserves expression typing]

If (h1) known(M), (h2) ∀AP ∈ Ran(α) : S(AP) = ∪{V | α(AP') = AP},
∅, M ⊨_{res} AP'[y/x] ⇒ V}, (h3) α ⊨ S, (h4) ∅ ⊨ B'[S], and (h5) B', M_1 ⊨_e e ⇒ M_2, V,
then ∅, M · M_1[S] ⊨_e e[y/x] ⇒ (M · M_2[S], V[S]).

Proof: Induction on e
• Case $e = f \overline{w}$:

By (h5) and rule [App], we have

(1) $\Gamma(f) = \forall \overline{w} : \{(B, (M, V))\}$

(2) $\forall AP \in \Gamma(f) : S'(AP) = \cup\{V' \mid \alpha(AP') = AP \land B', M \vdash_{res} AP'[\overline{w}/\overline{x}] \Rightarrow V'\}$

(3) $B' \vdash_{B_j} S'$

(4) $M_2 = M_1 B'_j \triangleleft M_j[S']$

(5) $V = V_j[S']$

Let

(6) $\forall AP \in \Gamma(f) : S''(AP) = \cup\{V'' \mid \alpha(AP') = AP \land \emptyset, M \cdot M_1[S] \vdash_{res} AP'[\overline{w}/\overline{x}] \Rightarrow V''\}$

By (h1), (h2), (h4), (h3) and lemma 20, whenever $B', M_1 \vdash_r AP'[\overline{w}/\overline{x}] \Rightarrow V'$

for $S'$ is satisfied, $\emptyset, M \cdot M_1[S] \vdash_r AP'[\overline{w}/\overline{x}] \Rightarrow V''[S]$ is satisfied. In addition, $Dom(S') = Dom(S'')$. As a result, by (6)

(7) $\forall AP \in Dom(S') : AP[S'] = V \Rightarrow V[S] = AP[S''] \ i.e., \ AP[S'][S] = AP[S'']$.

By (h1), (h2), (h3), (h4), (3) and lemma 15

(8) $\emptyset \vdash_{B_j} S'[S]$.

By (7), (8) and lemma 4, we have

(9) $\emptyset \vdash_{B_j} S''[S]$.

By (1), (6), (9) and rule [App], we have

(10) $\emptyset, M \cdot M_1[S] \vdash_{e} f \overline{w}[\overline{y}/\overline{x}] \Rightarrow (M \cdot M_1[S] \cdot M_j[S''], V_j[S''])$

By (h1), (h2), (2), (7), (h3), (h4) and lemma 22, we have

(11) $M \cdot M_1[S] \cdot M_j[S''] \sqsubseteq M \cdot (M_1 B'_j \triangleleft M_j[S''])[S]$

By (10), (11), (7) and subsumption rule, we have

(12) $\emptyset, M \cdot M_1[S] \vdash_{e} f \overline{w}[\overline{y}/\overline{x}] \Rightarrow (M \cdot (M_1 B'_j \triangleleft M_j[S''])[S], V_j[S'][S])$

By (12), (4), and (5), we have

$\emptyset, M \cdot M_1[S] \vdash_{e} f \overline{w}[\overline{y}/\overline{x}] \Rightarrow (M \cdot M_2[S], V[S])$

• Case $e = \&w$:

By (h5) and rule [Addr-id], we have (1) $M_2 = M_1$, and (2) $V_o = \{w\}$. Now we have two cases of $w[\overline{y}/\overline{x}]$: When $w = x_i$, by (1) and rule [Addr-id], we have (3-1) $\emptyset, M \cdot M_1[S] \vdash_{e} y_i \Rightarrow (M \cdot M_2[S], \{y_i\})$. By assumption, (h2), (2), and (3-1), we have (3-2) $V_o[S] = \{w\}[S] = \{y_i\}$. When $w \neq x_i$, by (1)
and rule [Addr-id], we have (4-1) \( \emptyset, M \cdot M_1[S] \vdash_e w \Rightarrow (M \cdot M_2[S], \{w\}) \). By assumption, (h2), (2), and (4-1), we have (4-2) \( V_e[S] = \{w\}[S] = \{w\} \).

- Case \( e = \&(*e) \):
  By (h5) and rule [Addr-*], we have (1) \( B', M_1 \vdash_e e \Rightarrow (M_2, V) \). By (h1), (h2), (h3), (h4), (1) and I.H., we have (2) \( \emptyset, M \cdot M_1[S] \vdash_e e[e/\overline{x}] \Rightarrow (M \cdot M_2[S], V[S]) \).
  By (2) and rule [Addr-*], we have \( \emptyset, M \cdot M_1[S] \vdash_e \&(*e)[e/\overline{x}] \Rightarrow (M \cdot M_2[S], V[S]) \).

- Case \( e = *e \):
  By (h5) and rule [Deref*], we have (1) \( B', M_1 \vdash_e e \Rightarrow M_2, V_1 \), and (2) \( B', M_2 \vdash_e V_1 \Rightarrow V \).
  By (h1), (h2), (h3), (h4), (1) and I.H., we have (3) \( \emptyset, M \cdot M_1[S] \vdash_e e[e/\overline{x}] \Rightarrow (M \cdot M_2[S], V_1[S]) \).
  By (h1), (h2), (h3), (h4), (2) and lemma 18, we have (4) \( \emptyset, M \cdot M_2[S] \vdash_e V_1[S] \Rightarrow V[S] \).
  By (3), (4) and rule [Deref*], we have \( \emptyset, M \cdot M_1[S] \vdash_e *e[e/\overline{x}] \Rightarrow (M \cdot M_2[S], V[S]) \).

- Case \( e = e.fld \): similar induction with \&(*e)

- Case \( e = e1:=e2 \):
  By (h5) and rule [Asgn], we have (1) \( B', M_1 \vdash_e e_1 \Rightarrow M', V_1 \), (2) \( B', M' \vdash_e e_2 \Rightarrow M'', V_2 \), (3) \( B', M'', V_2 \vdash_a V_1 \Rightarrow M_2 \), and (4) \( V = \emptyset \). By (1), (2) and I.H., we have (5) \( \emptyset, M \cdot M_1[S] \vdash_e e_1[e/\overline{x}] \Rightarrow M \cdot M'[S], V_1[S] \), and (6) \( \emptyset, M \cdot M'[S] \vdash_e e_2[e/\overline{x}] \Rightarrow M \cdot M''[S], V_2[S] \).
  By (h1), (h2), (h3), (h4), (3) and lemma 19, we have (7) \( \emptyset, M \cdot M''[S], V_2[S] \vdash_a V_1[S] \Rightarrow M \cdot M_2[S] \).
  By (5), (6), (7), (4) and rule [Asgn*], we have \( \emptyset, M \cdot M_1[S] \vdash_e (e_1:=e_2)[e/\overline{x}] \Rightarrow (M \cdot M_2[S], V[S]) \).

- Case \( e = e1; e2 \): similar induction with \emph{e1:=e2}

- Case \( e = \texttt{new}_l \):
  By (h5) and rule [New], we have (1) \( M_2 = M_1 \), and (2) \( V = \{l\} \). By (1), (h2) and rule [New], we have \( \emptyset, M \cdot M_1[S] \vdash_e \texttt{new}_l[e/\overline{x}] \Rightarrow (M \cdot M_2[S], V[S]) \).

- Case \( e = \texttt{if}(e_1,e_2) \):
  By (h5) and rule [If], we have (1) \( B', M_1 \vdash_e e_1 \Rightarrow M', V_1 \), (2) \( B', M_1 \vdash_e e_2 \Rightarrow M'', V_2 \), (3) \( B' \vdash M' \cup M'' \Rightarrow M_2 \), and (4) \( V = V_1 \cup V_2 \). By (1), (2) and I.H., we have (5) \( \emptyset, M \cdot M_1[S] \vdash_e e_1[e/\overline{x}] \Rightarrow (M \cdot M'[S], V_1[S]) \), and (6) \( \emptyset, M \cdot M_1[S] \vdash_e e_2[e/\overline{x}] \Rightarrow (M \cdot M''[S], V_2[S]) \).
  By (3) and lemma 21, we have (7) \( \emptyset \vdash M \cdot M'[S] \cup M \cdot M''[S] \Rightarrow M \cdot M_2[S] \).
By (4) and definition of \([S]\), we have (8) \(V[S] = (V_1 \cup V_2)[S] = V_1[S] \cup V_2[S]\). By (5), (6), (7), (8) and rule [If], we have \(\emptyset, M \cdot M_1[S] \vdash_e (\text{if}(e_1, e_2))[\overline{y/x}] \Rightarrow M \cdot M_2[S], V[S]\).

**Lemma 24 [ Substitution ]**

If (h1) known\((M)\), (h2) \(\forall AP \in \text{Ran}(\alpha) : S(AP) = \cup\{V \mid \alpha(AP') = AP \land \emptyset, M \vdash_{res} AP'[\overline{y/x}] \Rightarrow V\}\), (h3) \(\emptyset \vdash B'[S]\), and (h4) \(B', \epsilon \vdash_e e \Rightarrow M_f, V\), then \(\emptyset, M \vdash_e e[\overline{y/x}] \Rightarrow (M \cdot M_f[S], V[S])\).

**Proof:**

(A) The main goal of this lemma is to prove that the execution result of a function call is safely approximated by the summary application rule [App]. However, the presented type system is not precise enough to prove this lemma since it loses information on execution paths. Consider following example code:

```c
f(int *p) {
    l1: *p = 3; l2: *p = 4;
}
main() {
    int * x; int i, j;
    if(?) x = &i; else x = &j;
    l3: f(x);
}
```

The formal semantics kills side effect of \(l1\) since it keeps the path information by collecting all possible execution results as a set (e.g. the memory \(m\) computed by the formal semantics at the start of \(l3\) is \(\{[x \mapsto \{i\}], [x \mapsto \{j\}]\}\)). To the contrary, the memory \(M\) computed by the type system at the start of \(l3\) is \([x \mapsto \{i, j\}]\) which lose the information on execution paths. So, if we type the function call by inlining the function body of \(f\), the side effect of \(l1\) cannot be killed since \(p.*\) is resolved as \(\{i, j\}\) (update on multiple access paths). However, \(p.*\) is strongly-updated when we summarize the function by our type system since \(p.*\) can be locally concluded as a unique access path (it represents the same runtime object throughout that procedure execution for a particular call). To summarize, we can prove this lemma only when we keep those path information in the type system by extending the definition of memory as a set.
In this thesis, we informally overview the path sensitive extension considered for the proof of this lemma. Then, we show the soundness of the current type system by relating this extension to the current type system. Notice that such an extension is not the matter of the goal analysis which is path-insensitive but the matter of the proving soundness.

We first extend the formal semantics as follows. (1) It does not have the representative name for array or address of the heap allocated memory. (2) One transition in this formal semantics collects every possible execution results (trace) of each \( \text{mem} \in m \). Then, we define abstraction functions \( \text{abstract-amem} : \text{mem} \rightarrow M \) and \( \text{abstract-mem} : m \rightarrow MS \) as follows:

\[
\text{abstract-amem}(\text{mem}) = [ap_1 \mapsto mem(ap_1)] \cdots [ap_n \mapsto mem(ap_n)]
\]

where \( ap_i \in \text{Dom}(\text{mem}) \)

\[
\text{abstract-mem}(m) = \{\text{abstract-amem}(\text{mem}) | \text{mem} \in m\}
\]

Finally, this lemma is modified such that “if \( B', \epsilon \vdash e \Rightarrow M_f \) and \( \emptyset \vdash B'[S] \) then \( \forall M_i \in \text{abstract-mem}(m) : \emptyset, M_i \vdash e[S/\overline{y}] \Rightarrow M_i \cdot M_f[S_i] \)”.

In this thesis, we assume above extension for the proof. Then, we prove that \( M \cdot M_f[S] \) which is the result of summary application in the current type system, safely approximates \( \sqcup_i(M_i \cdot M_f[S_i]) \) which is the typing result of summary application in this extended type system.

By (A), every concretization \( M_i \) of \( M \) satisfies
(1) \( \alpha \vdash S_i \) where \( S_i \) is substitution for \( M_i \) computed by the rule [App].

By (A), (h2), and (1), we have (2) \( \forall AP : S_i(AP) \subseteq S(AP) \).

By (2), (h3) and lemma 4, we have (3) \( \emptyset \vdash B'[S_i] \).

By (h1), (A), (1), (3), (h4) and lemma 23, we have (4) \( \emptyset, M_i \vdash e[S/\overline{y}] \Rightarrow (M_i \cdot M_f[S_i], V[S_i]) \).

By (A) (i.e., \( M_i \subseteq M \)) and monotonicity of update operation (lemma 8), it is trivial to show that (5) \( M_i \cdot M_f[S_i] \subseteq M \cdot M_f[S] \), and (6) \( V[S_i] \subseteq V[S] \).
By (5), (6), known($M_i$), and known($M$), we have $(\cup_i (M_i \cdot M_f[S_i])) \subseteq M \cdot M_f[S]$, and $(\cup_i (V[S_i])) \subseteq V[S]$.

**Lemma 25 [ Type preservation (subject reduction) ]**

If $(m, e, K) \rightarrow C'$ and $\vdash (m, e, K) : (M_c, V_c)$, then $\vdash C' : (M_c, V_c)$.

**Proof:** Induction on the derivation of configuration $C$

- Case $C = (m, f \bar{y}, K)$:
  
  By assumption, we have

  1-1) $(m, f \bar{y}, K) \rightarrow (\text{update-mem}(m, \{z\}, \overline{m(y)}), e[\overline{\bar{z}}], \text{ret}(\bar{z}).K)$

  1-2) $\gamma(f) = \langle \bar{x}, e \rangle$ and $\bar{z}$ new

  2) $\vdash (m, f \bar{y}, K) : (M_c, V_c)$

  By (2) and rule [C-e], we have

  3) $m \preceq M$

  4) $\emptyset, M \vdash e f \bar{y} \Rightarrow M_r, V_r$

  5) $\emptyset, M_r, V_r \vdash K : (M_c, V_c)$

  By (4) and rule [App], we have

  6) $\Gamma(f) = \forall \bar{x} : \{(B_1, (M_1, V_1)), \cdots, (B_n, (M_n, V_n))\}$

  7) $\forall AP \in \Gamma(f) : S(AP) = \cup \{V \mid \alpha(AP') = AP, B, M \vdash \text{res} AP'[\bar{y}/\bar{x}] \Rightarrow V\}$

  8) $\emptyset \vdash B_i[S]$

  9) $M_r = M \cdot M_i[S]$

  10) $V_r = V_i[S]$

  Since this lemma is used by Theorem 1, $\gamma$ of formal semantics and $\Gamma$ of type system satisfy following condition.

  11) If $\gamma(f) = \langle \bar{x}, e \rangle$, then $\Gamma \vdash_{fdec} \text{fun } f(\bar{x}) = e$

  By (11), (1-2), (6) and rule [Fundec], we have

  12-1) $B_1, \epsilon \vdash e \Rightarrow M_1', V_1$

  ...  

  12-n) $B_n, \epsilon \vdash e \Rightarrow M_n', V_n$

  13-i) $M_i' - \bar{x} = M_i$

  We assume formal parameters of each function are different for brevity (instantiation issue). Then, by this assumption, (3), (7), (8), (12-i) and lemma 24
By (14), (1-2) and lemma 12, we have

\[ (M \cdot M'[S] - \overline{y}) \]

since each \( y_i \) is not aliased with any other

By (5) and rule [K-return], we have

\[ (M \cdot M'[S] - \overline{y}) \Rightarrow M(\overline{y}) \]

By assumption, we have (1) \((m, new, K) : (M_c, V_c)\)

By (3) and definition 1, we have

\[ (\text{update-mem}(m, \overline{e}), e[\overline{x}/\overline{y}], \overline{m}()) \Rightarrow M[\overline{x} \mapsto \overline{y}]\]

By (18), (16), (17) and rule [C-e], we have

\[ (\text{update-mem}(m, \overline{e}), e[\overline{x}/\overline{y}], \overline{m}()) \Rightarrow M[\overline{x} \mapsto \overline{y}] \]

By assumption, we have (1) \((m, &x, K) : (M_c, V_c)\)

By (2), rule [C-e] and [Addr-id], we have (3) \( m \subseteq M \), (4) \( \emptyset, M \vdash e \Rightarrow M, \{x\} \)

and (5) \( M, \{x\} \vdash K : M_c, V_c \).

By (3), (5) and rule [Conf-v], we have \( \vdash (m, \{x\}, K) : M_c, V_c \).

By assumption, we have (1) \((m, &(e), K) : (M_c, V_c)\)

By (2), rule [C-e] and [Addr-*], we have (3) \( m \subseteq M \), (4) \( \emptyset, M \vdash e \Rightarrow M_1, V \)

and (5) \( M_1, V \vdash K : M_c, V_c \).

By (3), (4), (5) and rule [C-e], we have
\[ \vdash (m, e, K) : (M_c, V_c). \]

- Case \( C = (m, \& (e.fld), K) \):

  By assumption, we have (1) \( (m, \& (e.fld), K) \rightarrow (m, \& e, \text{field}(fld).K) \), and (2) \( \vdash (m, \& (e.fld), K) : (M_c, V_c) \). By (2) and rule [C-e], we have (3) \( m \preceq M \), (4) \( \emptyset, M \vdash e \& (e.fld) \Rightarrow M_1, V_1 \), and (5) \( M_1, V_1 \vdash K : M_c, V_c \). By (4) and rule [Addr-fld], we have (6) \( \emptyset, M \vdash e \& e \Rightarrow M_1, V_2 \), and (7) \( V_1 = \text{field}(V_2, fld) \). By (7), (5) and rule [K-field], we have (8) \( M_1, V_2 \vdash \text{field}(fld).K : (M_c, V_c) \). By (3), (6), (8) and rule [C-e], we have \( \vdash (m, \& e, \text{field}(fld).K) : (M_c, V_c) \).

- Case \( C = (m, \ast e, K) \):

  By assumption, we have (1) \( (m, \ast e, K) \rightarrow (m, e, \text{deref}.K) \), and (2) \( \vdash (m, \ast e, K) : (M_c, V_c) \). By (2) and rule [C-e], we have (3) \( m \preceq M \), (4) \( \emptyset, M \vdash e \ast e \Rightarrow M_1, V_1 \), and (5) \( M_1, V_1 \vdash K : M_c, V_c \). By (4) and rule [Deref-\ast], we have (6) \( \emptyset, M \vdash e \Rightarrow M_1, V_2 \), and (7) \( \emptyset, M_1 \vdash_\ast V_2 \Rightarrow V_1 \). By (7), (5) and rule [K-deref], we have (8) \( M_1, V_2 \vdash \text{deref}.K : (M_c, V_c) \). By (3), (6), (8) and rule [C-e], we have \( \vdash (m, e, \text{deref}.K) : (M_c, V_c) \).

- Case \( C = (m, e.fld, K) \): similar induction with \( (m, \ast e, K) \)

- Case \( C = (m, e_1:=e_2, K) \):

  By assumption, we have (1) \( (m, e_1:=e_2, K) \rightarrow (m, e_1, \text{assignl}(e_2).K) \), and (2) \( \vdash (m, e_1:=e_2, K) : (M_c, V_c) \). By (2) and rule [C-e], we have (3) \( m \preceq M \), (4) \( \emptyset, M \vdash e_1:=e_2 \Rightarrow M', V' \), and (5) \( M', V' \vdash K : M_c, V_c \). By (4) and rule [Asgn], we have (6) \( \emptyset, M \vdash e_1 \Rightarrow M_1, V_1 \), (7) \( \emptyset, M_1 \vdash e_2 \Rightarrow M_2, V_2 \), (8) \( M_2, V_2 \vdash_\ast V_1 \Rightarrow M' \), and (9) \( V' = \emptyset \). By (7), (8), (9), (5) and rule [K-assignl], we have (10) \( M_1, V_1 \vdash \text{assignl}(e_2).K : M_c, V_c \). By (3), (6), (10) and rule [C-e], we have \( \vdash (m, e_1, \text{assignl}(e_2).K) : (M_c, V_c) \).

- Case \( C = (m, e_1; e_2, K) \): similar induction with \( (m, e_1:=e_2, K) \)

- Case \( C = (m, \text{if}(e_1, e_2), K) \):

  By assumption, we have (1) \( (m, \text{if}(e_1, e_2), K) \rightarrow (m, e_1, \text{jinto}(e_2, m).K) \), and (2) \( \vdash (m, \text{if}(e_1, e_2), K) : (M_c, V_c) \). By (2) and rule [C-e], we have (3) \( m \preceq M \), (4) \( \emptyset, M \vdash e \text{if}(e_1, e_2) \Rightarrow M', V' \), and (5) \( M', V' \vdash K : M_c, V_c \). By (4)

82
and rule [If], we have (6) \( \emptyset, M \vdash e_1 \Rightarrow M_1, V_1 \), (7) \( \emptyset, M \vdash e_2 \Rightarrow M_2, V_2 \), (8) \( \emptyset \vdash M_1 \sqcup M_2 \Rightarrow M' \), and (9) \( V' = V_1 \sqcup V_2 \). By (3), (7), (8), (9), (5) and rule [K-jointo], we have (10) \( M_1, V_1 \vdash \text{jointo}(e_2, m).K : (M_c, V_c) \). By (3), (6), (10) and rule [C-e], we have \( \vdash (m, e_1, \text{jointo}(e_2, m).K) : (M_c, V_c) \).

- Case \( C = (m, v, \text{ret}(\pi)).K \):
  
  By assumption, we have (1) \( (m, v, \text{ret}(\pi)).K \rightarrow (m - \pi, v, K) \), and (2) \( \vdash (m, v, \text{ret}(\pi)).K : (M_c, V_c) \). By (2) and rule [C-v], we have (3) \( m \preceq M \), (4) \( v \preceq V \), and (5) \( M, V \vdash \text{ret}(\pi).K : (M_c, V_c) \). By (5) and rule [K-return], we have (6) \( M - \pi, V \vdash K : M_c, V_c \). By definition 1, we have (7) \( m - \pi \preceq M - \pi \). By (7), (4), (6) and rule [C-v], we have \( \vdash (m - \pi, v, K) : (M_c, V_c) \).

- Case \( C = (m, v, \text{deref}).K \):
  
  By assumption, we have (1) \( (m, v, \text{deref}).K \rightarrow (m, m(v), K) \), and (2) \( \vdash (m, v, \text{deref}).K : (M_c, V_c) \). By (2) and rule [C-v], we have (3) \( m \preceq M \), (4) \( v \preceq V \), and (5) \( M, V \vdash \text{deref}.K : (M_c, V_c) \). By (5) and rule [K-deref], we have (6) \( \emptyset, M \vdash_{\pi} V \Rightarrow V_1 \), and (7) \( M, V_1 \vdash K : (M_c, V_c) \). By (3), (4), (6) and lemma 1 (sound read), we have (9) \( m(v) \preceq V_1 \). By (3), (9), (7) and rule [C-v], we have \( \vdash (m, m(v), K) : (M_c, V_c) \).

- Case \( C = (m, v, \text{field}(fld)).K \):
  
  By assumption, we have (1) \( (m, v, \text{field}(fld)).K \rightarrow (m, \{ap.fld | ap \in v\}, K) \), and (2) \( \vdash (m, v, \text{field}(fld)).K : (M_c, V_c) \). By (2) and rule [C-v], we have (3) \( m \preceq M \), (4) \( v \preceq V \), and (5) \( M, V \vdash \text{field}(fld).K : (M_c, V_c) \). By (5) and rule [K-field], we have (6) \( M, \text{field}(V, fld) \vdash K : (M_c, V_c) \). By (4), definition of “field” operation and \( \preceq \), we have (7) \( \{ap.fld | ap \in v\} \preceq \text{field}(V, fld) \). By (3), (7), (6) and rule [C-v], we have \( \vdash (m, \{ap.fld | ap \in v\}, K) : (M_c, V_c) \).

- Case \( C = (m, v, \text{assignl}(e)).K \):
  
  By assumption, we have (1) \( (m, v, \text{assignl}(e)).K \rightarrow (m, e, \text{assign}(v).K) \), and (2) \( \vdash (m, v, \text{assignl}(e).K) : (M_c, V_c) \). By (2) and rule [C-v], we have (3) \( m \preceq M \), (4) \( v \preceq V \), and (5) \( M, V \vdash \text{assignl}(e).K : (M_c, V_c) \). By (5) and rule
[K-assignl], we have (6) $\emptyset, M \vdash e \Rightarrow M_1, V_1$, (7) $\emptyset, M_1, V_1 \vdash u \Rightarrow M_2$, and (8) $M_2, \emptyset \vdash K : (M_c, V_c)$. By (4), (6), (7), (8) and rule [K-assignl], we have (9) $M_1, V_1 \vdash \text{assign}(v).K : (M_c, V_c)$. By (3), (6), (9) and rule [C-e], we have $\vdash (m, e, \text{assign}(v).K) : (M_c, V_c)$.

- **Case C = $(m, v, \text{assign}(v').K)$:**

  By assumption, we have (1) $(m, v, \text{assign}(v').K) \rightarrow (\text{update-mem}(m, v', v), \emptyset, K)$, and (2) $\vdash (m, v, \text{assign}(v').K) : (M_c, V_c)$. By (2) and rule [C-v], we have (3) $m \preceq M$, (4) $v \preceq V$, and (5) $M, V \vdash \text{assign}(v').K : (M_c, V_c)$. By (5) and rule [K-assignl], we have (6) $v' \preceq V'$, (7) $\emptyset, M, V \vdash u' \Rightarrow M_1$, and (8) $M_1, \emptyset \vdash K : (M_c, V_c)$. By (3), (4), (6), (7) and lemma 2 (sound update), we have (9) update-mem$(m, v', v) \preceq M_1$. By (9), (8) and rule [C-v], we have $\vdash (\text{update-mem}(m, v', v), \emptyset, K) : (M_c, V_c)$.

- **Case C = $(m, v, \text{jointo}(e, m_{old}).K)$:**

  By assumption, we have (1) $(m, v, \text{jointo}(e, m_{old}).K) \rightarrow (m_{old}, e, \text{join}(v, m).K)$, and (2) $\vdash (m, v, \text{jointo}(e, m_{old}).K) : (M_c, V_c)$. By (2) and rule [C-v], we have (3) $m \preceq M$, (4) $v \preceq V$, and (5) $M, V \vdash \text{jointo}(e, m_{old}).K : (M_c, V_c)$. By (5) and rule [K-jointo], we have (6) $m_{old} \preceq M_{old}$, (7) $\emptyset, M_{old} \vdash e \Rightarrow M_2, V_2$, (8) $B \vdash M \cup M_2 \Rightarrow M'$, (9) $V' = V \cup V_2$, and (10) $M', V' \vdash K : (M_c, V_c)$. By (4), (3), (9), (8), (10) and rule [K-jointo], we have (11) $M_2, V_2 \vdash \text{join}(v, m).K : (M_c, V_c)$. By (6), (7), (11) and rule [C-v], we have $\vdash (m_{old}, e, \text{join}(v, m).K) : (M_c, V_c)$.

- **Case C = $(m, v, \text{join}(v', m').K)$:**

  By assumption, we have (1) $(m, v, \text{join}(v', m').K) \rightarrow (m \cup m', v \cup v', K)$, and (2) $\vdash (m, v, \text{join}(v', m').K) : (M_c, V_c)$. By (2) and rule [C-v], we have (3) $m \preceq M$, (4) $v \preceq V$, and (5) $M, V \vdash \text{join}(v', m').K : (M_c, V_c)$. By (5) and rule [K-join]

  (6) $v' \preceq V'$, (7) $m' \preceq M'$, (8) $V'' = V' \cup V$, (9) $\emptyset \vdash M' \cup M \Rightarrow M''$, and (10) $M'', V'' \vdash K : (M_c, V_c)$. By (3), (7) and lemma 3 (sound join), we have (12) $(m \cup m') \preceq M''$. By (4), (6), (8) and definition 1, we have (13) $(v \cup v') \preceq V''$. By (12), (13), (10) and rule [C-v], we have $\vdash (m \cup m', v \cup v', K) : (M_c, V_c)$. 

84
요 약 문

업데이트 기록에 기반한 상향방식 포인트 분석

C나 Java 프로그램과 같이 포인터에 관련된 수행이 있는 프로그램을 효과적으로 분석하기 위해서는 포인터 분석이 중요하다. 대부분의 포인터 분석 방법들은 호출자로부터 피호출자의 순서로 분석을 수행하는 하향방식 분석으로 제안되었다. 하지만 이러한 방식의 분석들은 전체 프로그램이 주어졌을 때에도 작동 할 수 있기 때문에, 각각의 소프트웨어 컴포넌트들에 따로 작성되어 차후 순차적으로 합쳐진
는 방식으로 개발되는 모듈단위 개발 과정에서 활용될 수 없는 단점이 있다. 상향 방식의 포인터 분석도 프로그램을 피호출자로부터 호출자의 순서로 분석함으로
써 이러한 문제점을 해결할 수 있다. 하지만, 일반적으로 호출자에 대한 정보 없이
피호출자를 분석해야 하는 상향방식의 포인터 분석들은 호출 문맥의 차이에 따른
피호출자의 실행의 차이를 정확히 추적하기 어려운 문제가 있다.

본 논문에서는 업데이트 기록이라 이를 지어진 새로운 상향방식의 포인터 분석
공간을 기반으로, 프로그램 호름과 호출 문맥에 민감한 상향방식의 포인터 분석 알고리즘을 제안한다. 업데이트 기록은 함수의 호출 문맥에 독립적으로 메모리 상태를 요약할 수 있을 뿐만 아니라, 메모리 반응이 일어난 순서에 관한 정보를 유지할
수 있다. 업데이트 기록의 이러한 특성은 상향방식의 포인터 분석을 고안할 수 있
게 해줄 뿐만 아니라, 분석의 정확도를 높이기 위해 효과적으로 활용될 수 있는 정
보인 측은 메모리 반응 또는 관련된 별칭 문맥을 알아내는 데에도 효과적으로 사용
될 수 있다. 본 논문에서는 업데이트 기록을 이용하여 프로그램의 포인터 관련 수
행을 요약해주는 타입시스템을 정형화 하고, 이에 대한 추론 알고리즘으로서 상향 방식의 포인터 분석을 정형화 하였다. 그리고, 타입시스템 이론의 잘 정리된 증명
방법들을 이용하여, 기존의 연구에서는 충분히 다투지 못했던, 제안된 상향방식의
포인터 분석의 안전성에 대한 정형화된 증명을 제시하였다.

실험은 C 프로그램에 대한 포인터 분석을 제안된 방법과 기존의 대표적인 상향
방식의 포인터 분석들을 OCaml언어로 구현하여 비교하였다. 실험 결과 제안된 방
법이 기존의 방법들에 비해 성능을 많이 향상하지 않으면서도 정확도를 높일 수 있
음을 확인하였다. 또한, 제안된 상황방식의 포인터 분석 결과를 사용하여 초기화 되지 않은 포인터나 빈 포인터 참조 에러들을 추적해본 결과, 기존의 방법들을 사용한 실험에 비하여, 최대 37% 더 많은 포인터 참조가 안전함을 보일 수 있었다.
References


[8] Dino Distefano, Peter W. O’Hearn, and Hongseok Yang. A local shape analysis based on separation logic. In Holger Hermanns and Jens Palsberg, editors,


감사의 글

짧지 않았던 박사학위 기간 동안 저를 지도해 주시고, 무사히 학위 과정을 마칠 수 있도록 따뜻하게 배려해 주신 제 지도교수님에선 한대숙 교수님께 감사 드립니다. 언제나 열정적인 모습으로 연구하는 사람의 본을 보여주셨으며, 소중한 가르침으로 제 연구의 초석을 닦아주신 이광근 교수님께 감사 드립니다. 배우는 즐거움을 알게워 주시고, 연구에 대한 많은 동기부여를 해주셨으며, 언제나 따뜻한 배려로 저에게 연구의 길을 열어주시는 도경구 교수님께 감사 드립니다. 바쁜 시간에도 불구하고, 제 연구에 관심을 기울여 주시고 격려와 함께 논문을 심사해 주신 최광무 교수님, 김문주 교수님께도 깊은 감사를 드립니다.

오랜 기간을 함께 생활했던 카이스트 프로그래밍 연구실의 모든 선, 후배, 동료들 (현일씨, 이현준 박사, 서선애 박사, 최석우 박사, 최윤서 박사, 최완재 박사, 유희, 정석, 진우, 고영, 건의, 현익, 장영, 조범, 신하, 신원, 향수, 최교, 최광무, 정석, 진우, 주호) 에게 감사의 마음을 전하고 싶습니다. 제전 카이스트 로파스와 한양대 연구실의 선, 후배, 동료들 (인주선 박사님, 김익순 박사님, 이욱세 박사님, 류석영 박사님, 양홍석 박사님, 정성준 박사, 재균, 정택씨, 김상진 박사님, 형철형, 기상, 노건, 기형, 태형, 건의, 경호) 에게도 감사의 마음을 전합니다. 이 모든 분들은 저와 짧지 않은 기간을 함께하며, 함께 공부하고 토론하며 서로의 발전에 도움을 주었으며, 같은 자리에서 같은 고민과 기쁨을 함께 하셨던 분들입니다. 한국전 자동차연구소를 그만두고 박사과정을 다시 시작할 때, 격려해 주시고 성원해주신 백종명 팀장님과 선배님들, 동료님들 (손주찬 선배님, 박화규 선배님, 박성진 박사님, 장민수 선배님, 박지현씨) 에게도 감사의 말씀을 드리고 싶습니다. 어떤 인연도 소중하지 않은 인연이 없었으며 다시 한번 이 모든 분들에게 감사 드리고 싶습니다.

각자의 자리에서 바쁜 와중에도 서로에게 여유가 되어주신 제 총마교우들 (현선 석, 선희, 성훈, 주영, 현배, 대현, 기국, 창열, 식호, 향준, 형진, 문수, 창영, 민규, 호진, 주영, 기원, 정태) 에게도 감사의 마음을 전합니다.

만약으로 공간하는 아들을 끌어없는 사랑과 희생으로 묵묵히 보살피시고 버팀목이 되어 주신 저의 어머니, 아버지의 노고를 생각하면 저는 저절로 고개가 숙여집니다. 단지 감사하다는 말씀은 너무나도 부족한 제 부모님의 은혜에 고개 숙여
감사 드리고 사랑하는 마음을 전합니다. 듬직한 동생 현욱이, 제수씨, 가족들에게 감사 합니다.

마지막으로 제 인생의 과도기에 저와 결혼하여, 어려움을 함께 하고 믿어주고 격려하며 든든히 제 길을 지켜준, 저의 아내 김윤수에게 특별한 사랑의 마음을 전합니다. 제게 큰 축복과 힘이 되어준 저의 아들 병민이와 어머님, 아버님, 형님들과 가족들에게 진심으로 감사 드리고 사랑하는 마음을 전합니다.

그 외에도 제가 지금 기억 못하는 보이지 않는 곳에서 저를 도와주신 많은 분들께 감사하는 마음을 간직하며 글을 맺겠습니다.
Curriculum Vitae

Name : Hyun-Goo Kang
Date of Birth : October 22, 1971
Birthplace : Seoul, Korea
Address : Division of Computer Science, 373-1, KAIST, Guseong-dong, Yuseong-gu, Daejeon, South Korea
E-mail : hgkang91@gmail.com, hgkang@pllab.kaist.ac.kr

Educations

Career
1997. 3. – 1999. 2. Research Assistant, Hanyang University
1999. 3. – 2000. 6. Member of Engineering Staff, ETRI (Electronics and Telecommunications Research Institute)
2000.11. – 2002. 2. Commissioned Engineering Staff, ETRI
2001. 3. – 2003. 2. Teaching Assistant, KAIST
Publications

International Journals


International Conferences


**Technical Reports**


**Domestic Journals**


4. 강현구, 도경구. 액션의 온라인 부분계산. 정보과학회논문지, 26(12):1531-1543, 1999년 12월.
Domestic Conferences


2. 강현구, 이광근. nML을 이용한 기술용어 사전 시스템 구축. 정보과학회 추계학술대회, 28(2), 340-342, 2001년 4월.

3. 강현구, 박성진, 백종명, 박상봉. 비즈니스 모델을 기반으로 하는 컴퓨터 조립 형 응용프로그램 구조. 한국CALS/EC학회 추계학술대회, 창원대학교, 1999년 11월.

4. 강현구, 백종명, 손주찬, 박성진. ERP 개발도구의 설계 및 구현. 정보처리학회 추계학술대회, 숭실대학교, 1999년 10월.

5. 강현구, 도경구. 온라인 액션 부분계산. 정보과학회 프로그래밍 언어 연구회 워크숍, 부산, 1998년 9월.


기타

1. 강현구. nML 언더그라운드 콘서트. 마이크로소프트웨어, 2002년 10월.

2. 강현구. 온라인 액션 부분계산기의 설계 및 구현. 석사학위논문, 한양대학교, 1999년 2월.