Information and Software Technology 51 (2009) 691–707

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

A bottom-up pointer analysis using the update history

Hyun-Goo Kang [*], Taisook Han

Department of Computer Science, Korea Advanced Institute of Science and Technology, 373-1 Kusong-dong Yusong-gu, Taejon 305-701, Republic of Korea

ARTICLE INFO

Article history:
Received 28 February 2008
Received in revised form 29 August 2008
Accepted 15 November 2008
Available online 3 January 2009

Keywords:
Flow- and context-sensitive pointer analysis
Modular analysis
Strong update
Type system

ABSTRACT

Pointer analysis is an important part for the source code analysis of C programs. In this paper, we propose a bottom-up and flow- and context-sensitive pointer analysis algorithm, where *bottom-up* refers to the ability to perform the analysis from callee modules to caller modules. Our approach is based on a new modular pointer analysis domain named the *update history* that can abstract memory states of a procedure independently of the information on aliases between memory locations and keep the information on the order of side effects performed. Such a memory representation not only enables the analysis to be formalized as a bottom-up analysis, but also helps the analysis to effectively identify killed side effects and relevant alias contexts. The experiments performed on a pilot implementation of the method shows that our approach is effective for improving the precision of a client analysis.

1. Introduction

A *bottom-up program analysis* performs the analysis 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 C program analyzer computes the *summary transfer function* for each procedure without utilizing any information on the 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, this scheme 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 the behavior of procedure without callers. Thus, the analysis can be applied in 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 since the whole-program and analysis results need not to be in memory simultaneously during the analysis, and the reuse of the 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 independent of this modification means that only the dependent parts of the code need to be reanalyzed.

Pointer analysis is essential to the effective source code analysis of C programs. Our goal was to design and implement an effective bottom-up pointer analysis that could be used in a modular software development environment. Although several bottom-up approaches [1–10] for analyzing the pointer behavior of C programs have been proposed, several problems remain that need to be addressed, two of which form the focus of this paper.

Consider the following example code, which provides fundamental insight into these problems:

```
int g1, g2;
f(int **p, int **q) {
  l1: *p = &g1; l2: *q = &g2; l3: return *p;
}
main() {
  int *i, *j;
  l4: f(&i, &i); l5: f(&j, &j); l6: f(&i, &j);
}
```

Each memory location represented by expressions &p, &q, &g1, &g2, p, q at the start of procedure *f* are designated as *p*, *q*, *g*1, *g*2, *p*.∗, and *q*.∗, respectively.

The assignment *l*2 kills the effect of assignment *l*1 if both p and q initially point to the same memory location, as in the cases of procedure calls at *l*4 and *l*5. In this case, the analyzer can safely discard the effect of assignment *l*1, which is called a *strong update* [11]. Otherwise, the analyzer needs to keep the side effect of *l*1 (e.g., the procedure call at *l*6). 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, none of the existing bottom-up pointer analyses [7–10] that are flow- and context-sensitive can identify interprocedurally killed side effects of this type.

* Corresponding author. Tel.: +82 42 350 3573.
E-mail address: hgkang@ropas.kaist.ac.kr (H.-G. Kang).

0950-5849/$ - see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2008.11.003

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 algorithm needs to identify only relevant alias contexts[2] that change the behavior of a procedure.

Our approach is based on the insight that both the killed side effects and relevant alias contexts discussed above 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 $p.*$ is updated before $q.*$ is updated, we can kill the side effect of $p.*$ when the analyzer becomes aware that $p.*$ and $q.*$ 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.
(2) We illustrate the effectiveness of our approach with empirical results obtained from experiments performed on a pilot implementation of the method.
(3) 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.

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

[1] We call the memory locations that are decided at the procedure invocation, such as $p.*$ and $q.*$, *unknown locations*.

[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 paper, 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. Overview

2.1. Memory location abstraction: access path

We name memory locations using the notion of the *symbolic access path* [12], 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:

$$s \in Selector \quad ::= fld \mid *$$
$$AP \in AccessPath ::= x \mid l \mid AP.s$$

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 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->next`, `x->next->next`, ... at the start of a procedure are abstracted into a single abstract access path $x.*.next$, when the field `next` means the recursive access, and $k = 1$.

When $k = 1$, we say that the access path $x.*.next$ has *collapsed* since it represents several memory locations at run-time, while a *unique* access path (e.g., $p.*$) represents only one memory location. The value of a unique access path can be destructively updated because it represents the same run-time object throughout the execution of a procedure for a particular call [13,14].

2.2. Memory state abstraction: update history

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

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

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 Fig. 1 abstracts the memory state after update operations $p.*$ to $\{i\}$, $q.*$ to $\{j\}$, and $r.*$ to $\{k\}$ 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 ($\sqsubseteq_M$) and join operation ($\sqcup_M$) are not trivially defined. To simplify the overview of this section, 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.

```
    int i, j, k;
1 : f(int **p, int **q, int **r) {     { (B, ε) }    where B = ∅
2 :   *p = &i;                         { (B, [p.* ↦ {i}]) }
3 :   *q = &j;                         { (B, [p.* ↦ {i}][q.* ↦ {j}]) }
4 :   *r = &k;                         { (B, [p.* ↦ {i}][q.* ↦ {j}][r.* ↦ {k}]) }
5 :   *p = *q;                         { (B ∪ {q. * #r.*},  [q.* ↦ {j}][r.* ↦ {k}][p.* ↦ {j}]),
                                         (B,                [q.* ↦ {j}][r.* ↦ {k}][p.* ↦ {j, k}]) }
    }

6 : g(int **x, int **y, int **z) {     { (B, ε) }    where B = ∅
7 :   f(x, y, y);                      { (B  , [q.* ↦ {j}][r.* ↦ {k}][p.* ↦ {j, k}]
                                               = [{y.*}↦{j}][{y.*}↦{k}][{x.*}↦{j, k}]
                                               = [y.* ↦ {k}][x.* ↦ {j, k}]) }
8 :   f(x, y, z);                      { (B ∪ {y. * #z.*}  , [y.* ↦ {k}][x.* ↦ {j, k}][q.* ↦ {j}][r.* ↦ {k}][p.* ↦ {j}]
                                                             = [y.* ↦ {k}][x.* ↦ {j, k}][{y.*}↦{j}][{z.*}↦{k}][{x.*}↦{j}]
                                                             = [y.* ↦ {j}][z.* ↦ {k}][x.* ↦ {j}] ),
                                         (B                  , [y.* ↦ {k}][x.* ↦ {j, k}][q.* ↦ {j}][r.* ↦ {k}][p.* ↦ {j, k}]
                                                             = [y.* ↦ {j}][z.* ↦ {k}][x.* ↦ {j, k}]) }
    }
```

Fig. 1. Analysis example.

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 Fig. 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 $[AP_1\mapsto V_1]\ldots[AP_n\mapsto V_n]$, alias relations between AP_i and $\{AP_{i+1},\ldots,AP_n\}$ are considered in the analysis. Note that alias relations between AP_i and access paths $\{AP_1,\ldots,AP_{i-1}\}$ 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 Fig. 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 $[p.*\mapsto\{i\}]$, $[q.*\mapsto\{j\}]$, and $[r.*\mapsto\{k\}]$ to the initial memory ϵ 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 $\{q.*\#r.*\}$ and computes the corresponding summary for this input context as well. The side effect for $p.*$ in the old update history can be killed since the l-value of this assignment is a singleton set $\{p.*\}$, and $p.*$ 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 $p.*$, $q.*$, and $r.*$ in the summary for procedure f are resolved as $x.*$, $y.*$, and $y.*$, 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 $[q.*\mapsto\{j\}][r.*\mapsto\{k\}][p.*\mapsto\{j,k\}]$ for this particular calling context with initial memory ϵ 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 $\underline{\mapsto}$ 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 $[q.*\mapsto\{j\}]$ since the successive side effect $[r.*\mapsto\{k\}]$ is determined to be a side effect on the same unique access path $y.*$ under this particular calling context on line 7. Note that performing this strong update when we summarized procedure f is unsafe because

$$\begin{array}{rcl}
p \in program & ::= & \overline{fundec}; e \\
fundec \in fundec & ::= & \texttt{fun}\, f(\overline{x}){=}e \\
e \in expression & ::= & \&x \mid \&(e.fld) \mid \&(*e) \mid *e \mid e.fld \mid \texttt{new}_l \mid \\
& & e_1{:=}e_2 \mid f\ \overline{y} \mid e_1; e_2 \mid \texttt{if}(e_1, e_2) \\
x, fld, f \in Identifier & &
\end{array}$$

Fig. 2. The language.

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

Fig. 3. Types.

the B does not guarantee that $q.*$ and $r.*$ are identical. On line 8, $p.*$, $q.*$, and $r.*$ are resolved as $x.*$, $y.*$, and $z.*$, respectively. In this case, both $\{y.*\#z.*\}$ and $\emptyset$ are satisfiable depending on the alias context of procedure g. So, the analyzer introduces a new input context $\{y.*\#z.*\}$ and computes the corresponding summary for both input contexts.

3. The memory type system

A type system [15] can be regarded as an equation that specifies the type that safely approximates the execution result of a program. In this section, we formalize the bottom-up analysis pointer analysis that was informally presented in Section 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[3] 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 Section 4.

3.1. The language

We will use the following notation: Let X denote an arbitrary syntactic element in our type system, with $\overline{X}$ used to abbreviate various syntactic enumerations (e.g., $(\overline{X, Y})$ 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 Fig. 2. 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 [2], which is a bottom-up and flow-insensitive pointer analysis.

We refer the formal semantics of the language to our technical report [17] since it is only related to the formal proof whose details are also referred to the technical report.

3.2. Types

The types (analysis domains) in our memory type system are defined in Fig. 3. The *Value* type A (or V) is a set of access paths. The meanings of the *Memory* type M, the *InputContext* type B,[4] the *Inout* type P,[5] and the *Summary* type D are explained in Section 2. The *TypeEnv* type Γ 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 Fig. 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 access paths. If $x.*\#y.* \notin B$, this means that memory locations abstracted by $x.*$ and $y.*$ are not disjoint (may be aliased).

[3] Our type system is a non-standard 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 τ; and hence, we do not need the property "progress" [16] 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.

[4] The extension of the type system for the other alias relations such as "must be aliased" is direct, and is not considered in this paper for brevity.

[5] The definition of P is extended to a tuple $(B, (M, V))$, where value V represents a return value.

$$[\text{hypoth}] \quad \frac{as \in B}{B \vdash_{as} as}$$

$$[AP\#AP] \quad \frac{\text{known}(AP_1) \quad \text{known}(AP_2) \quad AP_1 \neq AP_2}{B \vdash_{as} AP_1\#AP_2}$$

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

Fig. 4. Typing rules: input context.

$$[\text{read-}\epsilon] \quad B, \epsilon \vdash_{AP} AP \Rightarrow \{\alpha(AP.*)\}$$

$$[\text{read-}=] \quad \frac{\text{unique}(\alpha, AP)}{B, M[AP \mapsto V_1] \vdash_{AP} AP \Rightarrow V_1}$$

$$[\text{read-}\#] \quad \frac{B \vdash AP\#AP_1 \quad B, M \vdash_{AP} AP \Rightarrow V_2}{B, M[AP_1 \mapsto V_1] \vdash_{AP} AP \Rightarrow V_2}$$

$$[\text{read-safe}] \quad \frac{B, M \vdash_{AP} AP \Rightarrow V_2}{B, M[AP_1 \mapsto V_1] \vdash_{AP} AP \Rightarrow V_1 \cup V_2}$$

$$[\text{read-}A] \quad \frac{B, M \vdash_{AP} AP_1 \Rightarrow V_1 \quad \cdots \quad B, M \vdash_{AP} AP_n \Rightarrow V_n}{B, M \vdash_{r} \{AP_1, \cdots, AP_n\} \Rightarrow V_1 \cup \cdots \cup V_n}$$

Fig. 5. Typing rules: memory reads.

3.4. Points-to set of M: read operation

The judgement $B, M \vdash_{AP} AP \Rightarrow V$ specifies the safe approximation of pointed access paths V of 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 Fig. 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 ϵ (rule [read-ϵ]), this indicates that we are reading the value to which AP initially points. This value is represented as abstract access path $\alpha(AP.*)$, where $\alpha \in AP \rightarrow AP$ is a *well-known* [9,2] 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 "collapsed(α, 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) $\exists AP_1, AP_2 \in Dom(\alpha) : AP_1 \neq AP_2 \wedge \alpha(AP_1) = \alpha(AP_2) = AP$. Otherwise, AP is not a collapsed access path, which is denoted as "unique(α, 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 [read-=]) 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 [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 [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][6]).

Now, judgement $B, M \vdash_r A \Rightarrow V$ 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 Fig. 5 computes the pointed access paths of $\{AP_1, \ldots, AP_n\}$ 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 Fig. 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 $M_1 \sqsubseteq_M M_2$,

[6] 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 Section 4.

[shift]

$$
\begin{array}{l}
\text{shift}(B, \epsilon, AP) = [AP \mapsto \{\alpha(AP.*)\}] \\
\text{shift}(B, M[AP \mapsto V], AP) = M[AP \mapsto V] \\
\text{shift}(B, M[AP' \mapsto V'], AP) = \\
\quad \text{let shift}(B, M, AP) = M'[AP \mapsto V] \ \text{ in} \\
\quad \text{if } B \vdash AP\#AP' \vee \text{collapsed}(\alpha, AP) \text{ then } M'[AP' \mapsto V'][AP \mapsto V] \\
\quad \text{else } M'[AP' \mapsto V'][AP \mapsto V \cup V']
\end{array}
$$

[Subtype relation for the memory type]

$$
[\sqsubseteq_M(1)] \qquad B \vdash M \sqsubseteq_M \text{shift}(B, M, AP)
$$

$$
[\sqsubseteq_M(2)] \quad \frac{B \vdash \text{shift}(B, M_1, AP) \sqsubseteq_M M_2[AP \mapsto V]}{B \vdash M_1 \sqsubseteq_M M_2[AP \mapsto V]}
$$

$$
[\sqsubseteq_M(3)] \quad \frac{B \vdash M \sqsubseteq_M M' \quad V \subseteq V'}{B \vdash M[AP \mapsto V] \sqsubseteq_M M'[AP \mapsto V']}
$$

[Join operation for the memory type]

$$
[\sqcup_M(1)] \quad \frac{B \vdash M_1 \sqcup_M M_2 \Rightarrow M_3}{B \vdash M_1[AP \mapsto V] \sqcup_M M_2[AP \mapsto V'] \Rightarrow M_3[AP \mapsto V \cup V']}
$$

$$
[\sqcup_M(2)] \quad \frac{B \vdash M_1[AP \mapsto V] \sqcup_M \text{shift}(B, M_2, AP) \Rightarrow M_3}{B \vdash M_1[AP \mapsto V] \sqcup_M M_2 \Rightarrow M_3}
$$

$$
[\sqcup_M(3)] \quad \frac{B \vdash \text{shift}(B, M_1, AP) \sqcup_M M_2[AP \mapsto V] \Rightarrow M_3}{B \vdash M_1 \sqcup_M M_2[AP \mapsto V] \Rightarrow M_3}
$$

$$
[\sqcup_M(4)] \qquad B \vdash \epsilon \sqcup_M \epsilon \Rightarrow \epsilon
$$

Fig. 6. Typing rules: subtype relation (order between memory types).

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 Fig. 6 is devised to find such an M_3 by moving the side effect on AP in M_1 to the rightmost position such that the set of memory states abstracted by shift(B, M_1, AP) always contains those of M_1 under input context B. In other words, 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 $B, M_1 \vdash_{AP} AP_i \Rightarrow V_i$ and $B, M_3 \vdash_{AP} AP_i \Rightarrow V'_i$, then $V_i \subseteq V'_i$.

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

The typing rules for the memory join operation $(\sqcup_M)$ are given in Fig. 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 Section 4 is trivially derived from these rules. For example, rule $[\sqcup_M(3)]$ can have precedence when rule $[\sqcup_M(2)]$ is also applicable.

The orders and deterministic join operations between types other than the memory type, are defined in the standard manner as shown in Fig. 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 $X_2 \sqsubseteq_X X_1$ [18]. 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

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 Fig. 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.

$$
\begin{array}{ll}
B_1 \sqsubseteq_B B_2 & \text{iff } B_2 \subseteq B_1 \\
(B_1, (M_1, V_1)) \sqsubseteq_P (B_2, (M_2, V_2)) & \text{iff } B_2 \sqsubseteq_B B_1 \ \wedge \ B_2 \vdash M_1 \sqsubseteq_M M_2 \ \wedge \ V_1 \subseteq V_2 \\
D \sqsubseteq_D D' & \text{iff } \forall P' \in D' : \exists P \in D : P \sqsubseteq_P P' \\
D_1 \sqcup_D D_2 = \{(B_{ij}, (M_{ij}, V_i \cup V_j)) \mid & \begin{array}{l} (B_i, (M_i, V_i)) \in D_1, \ (B_j, (M_j, V_j)) \in D_2, \\ B_{ij} = B_i \cup B_j, \ B_{ij} \vdash M_i \sqcup_M M_j \Rightarrow M_{ij} \end{array} \}
\end{array}
$$

Fig. 7. Orders between types.

$$[\text{update-s}]\quad \frac{\text{unique}(\alpha, AP) \quad M' = M - \{AP\}}{B, M, V \vdash_u \{AP\} \Rightarrow M'[AP \mapsto V]}$$

$$[\text{update-w}]\quad \frac{\begin{array}{c} A = \{AP_1, \cdots, AP_n\} \quad \text{shift}(B, M, AP_i) = M''[AP_i \mapsto V_i] \\ M' = (M - A)[AP_1 \mapsto V \cup V_1] \cdots [AP_n \mapsto V \cup V_n] \end{array}}{B, M, V \vdash_u A \Rightarrow M'}$$

Fig. 8. Typing rules: memory updates.

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 $[AP \mapsto V_{old}] \in M$ 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 $[AP \mapsto V_{old}] \in M$ 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 Fig. 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) $\Gamma \vdash_{fdec} \texttt{fun}\, f(\bar{x}) = e$,
(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 "Γ types f", which means that Γ 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 Fig. 10, and the auxiliary operations used in those rules are given in Fig. 9.

The [Program] rule computes the safe approximation of execution results M_p and V_p of given program p using type environment Γ, which provides the safe summary of each procedure f_i invoked in e. In the type system, we simply assume that such Γ for a program are given globally. The inference algorithm of Section 4 describes the algorithm to infer such Γ. In [Fundec], we first prepare a set of input contexts $B_1, \ldots, B_n$, where each B_i is consid-

[**Operations**]

$$\begin{array}{rcl} M \overset{B}{\triangleleft} M'[S] & = & \text{the memory after we iteratively perform each instantiated} \\ & & \text{update } [A_i \mapsto V_i] \text{ of } M'[S] \text{ to } M \text{ under } B \\ \text{field}(A, fld) & = & \{AP' \mid AP \in A \wedge \alpha(AP.fld) = AP'\} \\ AP[\overline{y}/\overline{x}] & = & \text{if } AP = x_i.X \text{ then } y_i.X \text{ else } AP \end{array}$$

[**Substitution**] $S : AP \overset{\text{fin}}{\to} A$

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

[**Access path resolving**]

$$[\text{resolve(1)}]\quad \frac{(AP = x \ \vee \ AP = l)}{B, M \vdash_{res} AP \Rightarrow \{AP\}}$$

$$[\text{resolve(2)}]\quad \frac{AP = AP'.* \quad B, M \vdash_{res} AP' \Rightarrow V' \quad B, M \vdash_r V' \Rightarrow V}{B, M \vdash_{res} AP \Rightarrow V}$$

$$[\text{resolve(3)}]\quad \frac{AP = AP'.fld \quad B, M \vdash_{res} AP' \Rightarrow V' \quad \text{field}(V', fld) = V}{B, M \vdash_{res} AP \Rightarrow V}$$

Fig. 9. Auxiliary operations.

$$[\text{Program}]\quad \frac{\Gamma \vdash_{fdec} fundec_1 \quad \cdots \quad \Gamma \vdash_{fdec} fundec_n \quad \emptyset, \epsilon \vdash_e e \Rightarrow M_p, V_p}{\vdash_p \overline{fundec}; e \Rightarrow M_p, V_p}$$

$$[\text{Fundec}]\quad \frac{\begin{array}{c}\Gamma(f) = \{(\overline{B, (M, V)})\} \\ B_1, \epsilon \vdash_e e \Rightarrow M'_1, V_1 \quad \cdots \quad B_n, \epsilon \vdash_e e \Rightarrow M'_n, V_n \\ M'_i - \overline{x} = M_i\end{array}}{\Gamma \vdash_{fdec} \texttt{fun}\, f(\overline{x}) {=} e}$$

$$[\text{Addr-id}]\quad B, M \vdash_e \&x \Rightarrow M, \{x\}$$

$$[\text{Addr-}*]\quad \frac{B, M \vdash_e e \Rightarrow M_1, V}{B, M \vdash_e \&(*e) \Rightarrow M_1, V}$$

$$[\text{Addr-fld}]\quad \frac{B, M \vdash_e \&e \Rightarrow M_1, V \quad A = \text{field}(V, fld)}{B, M \vdash_e \&(e.fld) \Rightarrow M_1, A}$$

$$[\text{Deref-}*]\quad \frac{B, M \vdash_e e \Rightarrow M_1, V_1 \quad B, M_1 \vdash_r V_1 \Rightarrow V}{B, M \vdash_e *e \Rightarrow M_1, V}$$

$$[\text{Deref-fld}]\quad \frac{B, M \vdash_e \&e \Rightarrow M_1, V_1 \quad B, M_1 \vdash_r \text{field}(V_1, fld) \Rightarrow V}{B, M \vdash_e e.fld \Rightarrow M_1, V}$$

$$[\text{Asgn}]\quad \frac{B, M \vdash_e e_1 \Rightarrow M_1, V_1 \quad B, M_1 \vdash_e e_2 \Rightarrow M_2, V_2 \quad B, M_2, V_2 \vdash_u V_1 \Rightarrow M_3}{B, M \vdash_e e_1{:=}e_2 \Rightarrow M_3, \emptyset}$$

$$[\text{App}]\quad \frac{\begin{array}{c}\Gamma(f) = \{(\overline{B, (M, V)})\} \\ \forall AP \in \Gamma(f) : S(AP) = \cup\{V \mid \alpha(AP') = AP,\ B, M \vdash_{res} AP'[\overline{y}/\overline{x}] \Rightarrow V\} \\ B \vdash B_i[S]\end{array}}{B, M \vdash_e f\, \overline{y} \Rightarrow M \overset{B}{\lhd} M_i[S], V_i[S]}$$

$$[\text{New}]\quad B, M \vdash_e \texttt{new}_l \Rightarrow M, \{l\}$$

$$[\text{Seq}]\quad \frac{B, M \vdash_e e_1 \Rightarrow M_1, V_1 \quad B, M_1 \vdash_e e_2 \Rightarrow M_2, V_2}{B, M \vdash_e e_1; e_2 \Rightarrow M_2, V_2}$$

$$[\text{If}]\quad \frac{B, M \vdash_e e_1 \Rightarrow M_1, V_1 \quad B, M \vdash_e e_2 \Rightarrow M_2, V_2 \quad B \vdash M_1 \sqcup_M M_2 \Rightarrow M'}{B, M \vdash_e \texttt{if}(e_1, e_2) \Rightarrow M', V_1 \cup V_2}$$

Fig. 10. Typing rules: program.

ered to be relevant. If each (B_i, ϵ) types e as (M'_i, V_i), the summary of procedure f is $\{(B_1, (M_1, V_1)), \ldots, (B_n, (M_n, V_n))\}$. Note that Γ used to type procedure f contains its own summary, which explains the typing for recursion and mutually recursive procedures.

[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 Fig. 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 Fig. 8.

In [App], we first compute the substitution S that represents the calling context using the access path abstraction function α (explained in Section 3.4) used to type callee procedure f. $AP'[\bar{y}/\bar{x}]$ represents all concrete access paths of the caller that are abstracted into abstract access path AP of procedure f, whose formal parameters are $\bar{x}$. So, the caller's access paths for AP are computed by iteratively reading the sequence of access path selectors of $AP'[\bar{y}/\bar{x}]$ with the caller's memory M according to the access path resolving rules of Fig. 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 $inout(B_i, (M_i, V_i))$, 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 $M \overset{B}{\lhd} M_i[S]$ and the substitution in Fig. 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 $\texttt{malloc}_l(\texttt{sizeof(structlist)})$, where `list` is a structure that has two fields d and $next$. We can use a refined α that distinguishes

$$[t_1]\quad \dfrac{\dfrac{}{B,\epsilon\vdash_e \&p\Rightarrow\epsilon,\{p\}}\,\text{Addr-id}\qquad \dfrac{\dfrac{}{B,\epsilon\vdash_{AP} p\Rightarrow\{\alpha(p.*)\}}\,\text{read-}\epsilon}{B,\epsilon\vdash_r\{p\}\Rightarrow\{p.*\}}\,\text{read-}A}{B,\epsilon\vdash_e *(\&\mathtt{p})\Rightarrow\epsilon,\{p.*\}}\,\text{Deref-}*$$

$$[t_2]\quad \dfrac{\dfrac{\cdots}{\cdots}[t_1]\qquad \dfrac{}{B,\epsilon\vdash_e \mathtt{new_1}\Rightarrow\epsilon,\{l_1\}}\,\text{New}\qquad \dfrac{\text{unique}(\alpha,p.*)}{B,\epsilon,\{l_1\}\vdash_u\{p.*\}\Rightarrow[p.*\mapsto\{l_1\}]}\,\text{update-s}}{B,\epsilon\vdash_e *(\&\mathtt{p}):=\mathtt{new_1}\Rightarrow[p.*\mapsto\{l_1\}],\emptyset}\,\text{Asgn}$$

$$[t_3]\quad \dfrac{\dfrac{\dfrac{}{B,\epsilon\vdash_e \&r\Rightarrow\epsilon,\{r\}}\,\text{Addr-id}\quad \dfrac{\dfrac{}{B,\epsilon\vdash_{AP} r\Rightarrow\{\alpha(r.*)\}}\,\text{read-}\epsilon}{B,\epsilon\vdash_r\{r\}\Rightarrow\{r.*\}}\,\text{read-}A}{B,\epsilon\vdash_e *(\&\mathtt{r})\Rightarrow\epsilon,\{r.*\}}\,\text{Deref-}*\quad \dfrac{\dfrac{}{B,\epsilon\vdash_{AP} r.*\Rightarrow\alpha(r.*.*)}\,\text{read-}\epsilon}{B,\epsilon\vdash_r\{r.*\}\Rightarrow\{r.*.*\}}\,\text{read-}A}{B,\epsilon\vdash_e *(*(\&\mathtt{r}))\Rightarrow\epsilon,\{r.*.*\}}\,\text{Deref-}*$$

$$[t_4]\quad \dfrac{\dfrac{\text{similar with }[t_1]}{B,\epsilon\vdash_e *(\&q)\Rightarrow\epsilon,\{q.*\}}\,\text{Deref-}*\qquad \dfrac{\cdots}{\cdots}[t_3]\qquad \dfrac{\text{unique}(\alpha,q.*)}{B,\epsilon,\{r.*.*\}\vdash_u\{q.*\}\Rightarrow[q.*\mapsto\{r.*.*\}]}\,\text{update-s}}{B,\epsilon\vdash_e *(\&\mathtt{q}):=*(*(\&\mathtt{r}))\Rightarrow[q.*\mapsto\{r.*.*\}],\emptyset}\,\text{Asgn}$$

$$[t_5]\quad \dfrac{\dfrac{\dfrac{\dfrac{}{B\vdash[p.*\mapsto\{l_1\}]\sqcup_M\text{shift}(B,\epsilon,p.*)\Rightarrow[p.*\mapsto\{l_1,p.*.*\}]}\,\sqcup_M(1)}{B\vdash[p.*\mapsto\{l_1\}]\sqcup_M\epsilon\Rightarrow[p.*\mapsto\{l_1,p.*.*\}]}\,\sqcup_M(2)}{B\vdash\text{shift}(B,[p.*\mapsto\{l_1\}],q.*)\sqcup_M[q.*\mapsto\{r.*.*\}]\Rightarrow[p.*\mapsto\{l_1,p.*.*\}][q.*\mapsto V_1\cup\{r.*.*\}]}\,\sqcup_M(1)}{B\vdash[p.*\mapsto\{l_1\}]\sqcup[q.*\mapsto\{r.*.*\}]\Rightarrow[p.*\mapsto\{l_1,p.*.*\}][q.*\mapsto V_1\cup\{r.*.*\}]}\,\sqcup_M(3)$$

$$\text{where, shift}(B,[p.*\mapsto\{l_1\}],q.*)=[p.*\mapsto\{l_1\}][q.*\mapsto V_1]\ \wedge\ V_1=\begin{cases}\{q.*.*\} & \text{if } p.*\#q.*\in B\\ \{l_1,q.*.*\} & \text{otherwise}\end{cases}$$

$$[t_6]\quad \dfrac{\dfrac{\cdots}{\cdots}[t_2]\qquad\dfrac{\cdots}{\cdots}[t_4]\qquad\dfrac{\cdots}{\cdots}[t_5]}{B,\epsilon\vdash_e \mathtt{if}(*(\&\mathtt{p}):=\mathtt{new_1},\ *(\&\mathtt{q}):=*(*(\&\mathtt{r})))\Rightarrow[p.*\mapsto\{l_1,p.*.*\}][q.*\mapsto V_1\cup\{r.*.*\}],\emptyset}\,\text{If}$$

$$[t_7]\quad \dfrac{\begin{array}{c}\Gamma(f1)=\left\{\begin{array}{ll}(\emptyset, & ([p.*\mapsto\{l_1,p.*.*\}][q.*\mapsto\{l_1,r.*.*,q.*.*\}],\emptyset)),\\ (\{p.*\#q.*\}, & ([p.*\mapsto\{l_1,p.*.*\}][q.*\mapsto\{r.*.*,q.*.*\}],\emptyset))\end{array}\right\}\\ [t_6]\text{ where }B=\emptyset\qquad\qquad [t_6]\text{ where }B=\{p.*\#q.*\}\end{array}}{\Gamma\vdash_{fdec}\mathtt{fun}\,f1(\mathtt{p},\mathtt{q},\mathtt{r})\ =\ \mathtt{if}(*(\&\mathtt{p}):=\mathtt{new_1},\ *(\&\mathtt{q}):=*(*(\&\mathtt{r})))}\,\text{Fundec}$$

$$[t_8]\quad \dfrac{\begin{array}{c}\Gamma(f1)=\cdots\\ S=\{(p.*,\{g1.*\}),(p.*.*,\{g1.*.*\}),(q.*,\{g1.*\}),(q.*.*,\{g1.*.*\}),(r.*.*,\{g3.*.*\})\}\\ \emptyset\vdash_B\emptyset[S]\\ \begin{array}{l}[g2\mapsto\{g1.*\}]\overset{\emptyset}{\triangleleft}(([p.*\mapsto\{l_1,p.*.*\}][q.*\mapsto\{l_1,r.*.*,q.*.*\}])[S])\\ =[g2\mapsto\{g1.*\}]\overset{\emptyset}{\triangleleft}[\{g1.*\}\mapsto\{l_1,g1.*.*\}][\{g1.*\}\mapsto\{l_1,g3.*.*,g1.*.*\}]\\ =[g2\mapsto\{g1.*\}][g1.*\mapsto\{l_1,g3.*.*,g1.*.*\}]\end{array}\end{array}}{\emptyset,[g2\mapsto\{g1.*\}]\vdash_e \mathtt{f1}(\mathtt{g1},\mathtt{g2},\mathtt{g3})\Rightarrow[g2\mapsto\{g1\}][g1.*\mapsto\{l_1,g1.*.*,g3.*.*\}],\emptyset}\,\text{App}$$

Fig. 11. Typing example.

the fields of the structure `list`. We can also relate the call sequence σ to the allocation point l using a tuple $(\{l\},\sigma)$, where the type variable σ 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

Fig. 11 shows the example typing (type derivation trees) for the following C code:

```
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. For example, `*q=*r` is translated into `*(&q) := **(&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.

```
$\text{np\_infer}_r(B, M, \{AP_1, \cdots, AP_n\}) =$
  return $(\text{np\_infer}_{AP}(B, M, AP_1) \cup \cdots \cup \text{np\_infer}_{AP}(B, M, AP_n))$;
$\text{np\_infer}_{AP}(B, \epsilon, AP) =$ return $(\{\alpha(AP.*)\})$
$\text{np\_infer}_{AP}(B, M[AP \mapsto V_1], AP) =$
   if $\text{unique}(\alpha, AP)$ then return $(V_1)$;
   else return $(V_1 \cup \text{np\_infer}_{AP}(B, M, AP))$;
$\text{np\_infer}_{AP}(B, M[AP_1 \mapsto V_1], AP) =$
   if $\text{infer}_{as}(B, AP_1\#AP)$ then return $(\text{np\_infer}_{AP}(B, M, AP))$;
   else return $(V_1 \cup \text{np\_infer}_{AP}(B, M, AP))$;
```

Fig. 12. Non-partitioning operations.

$[t_2]$ and $[t_4]$ are the type derivation trees[7] for the assignments $l1$ and $l2$ of procedure $f1$, respectively. Note that we simply assume that access path abstraction function α 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 α such that access paths $p.*$ and $q.*$ are unique access paths and the relevant input contexts of $f1$ 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 [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 $f1$. 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 $f2$. Since the typing result of `g2 = g1` is $[g2\mapsto\{g1.*\}]$, access paths $p.*$ and $q.*$ of the summary of procedure $f1$ are resolved as $g1.*$ and $g1.*$, respectively. Therefore, the output for the input context $\emptyset$ is applied for this calling context.

3.9. Correctness of the type system

Theorem 1 (Soundness of the memory type system). *The typing results of our memory type system safely approximates the execution result of program p.*

Proof. The *full formal* proof is available in our technical report [17]. □

4. The analysis algorithm

In this section, we formulate the bottom-up pointer analysis algorithm as a type inference algorithm for the memory type system in Section 3. The primary goal of this inference algorithm is computing a safe Γ which is simply assumed to be given[8] in the type system. Note that [Fundec] recursively uses Γ to specify the conditions for Γ 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 [19]. 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 Γ, is the secondary goal of our inference algorithm.

4.1. Type inference without partitioning

Since the memory type system does not have any declarative deduction rules such as a subsumption rule, the non-partitioning operations (abstract transfer functions) $\text{np\_infer}_{as} \in B \times as \to Bool$, $\text{np\_infer}_B \in B \times B \to Bool$, $\text{np\_infer}_{AP} \in B \times M \times AP \to V$, $\text{np\_infer}_r \in B \times M \times A \to V$, $\text{np\_infer}_{\sqsubseteq_M} \in B \times M \times M \to Bool$, $\text{np\_infer}_{\sqcup_M} \in B \times M \times M \to M$, $\text{np\_infer}_u \in B \times M \times V \times A \to M$, $\text{np\_infer}_{res} \in B \times M \times AP \to V$, and $\text{np\_infer}_e \in B \times M \times e \to M \times V$ that, respectively, compute the safe typing results for the given input context B of judgements $B \vdash_{as} as$, $B \vdash_B B'$, $B, M \vdash_{AP} AP \Rightarrow V$, $B, M \vdash_r A \Rightarrow V$, $B \vdash M_1 \sqsubseteq_M M_2$, $B \vdash M_1 \sqcup_M M_2 \Rightarrow M_3$, $B, M, V \vdash_u A \Rightarrow M'$, $B, M \vdash_{res} AP \Rightarrow V$, and $B, M \vdash_e e \Rightarrow M', V$, are trivially defined. For example, the non-partitioning read operation $\text{np\_infer}_r \in B \times M \times A \to V$ which computes the typing result V for the judgement $B, M \vdash_r A \Rightarrow V$ using the given B, M, and A, can be trivially defined as shown in Fig. 12. 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

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 $\text{partition}(B, M, A)$ operation of Fig. 13 computes a set of input contexts $\{B_1, \ldots, B_n\}$ partitioned from B (i.e., $B_i \sqsubseteq_B B$) 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 $\text{infer}_r \in B \times M \times A \to 2^{(B,V)}$ of Fig. 13, which introduces the relevant contexts B_is by need and computes the corresponding read values V_is, 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 ($\text{np\_infer}_r \in B \times M \times A \to V$) operation of Section 4.1.

Partitioning operations for other non-partitioning operations are defined in a similar manner to infer_r.

[7] We marked the used rule at the right-hand side of each type derivation tree, and the abbreviation $\frac{\vdots}{\vdots}\,[t_1]$ is used to denote the type derivation tree $[t_1]$.

[8] The access path abstraction function α 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.

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

Fig. 13. Partitioning operations.

4.3. Intraprocedural analysis stage: infer$_e$

The intraprocedural analysis stage of our inference algorithm is defined by operation infer$_e \in B \times M \times e \to D$ as shown in Fig. 14. infer$_e(B, M, e) = D$ summarizes the behavior of expression e of procedure f for a given B, M, and Γ (Γ are given globally from the interprocedural stage). In terms of data flow analysis, infer$_e$ can be considered as an abstract transfer function for node n_e of a procedure f. 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 $B_i \sqsubseteq B$, which can be effectively identified by the partitioning operations of Section 4.2, we introduce such B_is into the analysis and compute the corresponding results as $\{(B, (M_e, V_e)),\ (B_1, (M_1, V_1)), \ldots, (B_n, (M_n, V_n))\}$.

For $e_1 := e_2$, we first summarize the behavior of e_1 partitioning B to B_is if they are relevant to type (determine the behavior of) e_1. Then for each partitioned B_i of infer$_e(B, M, e_1)$, we summarize the

```
infer$_e(B, M, e_s)$ =
$D_B = \emptyset$;
{
match $e_s$ with
 | $e_1 := e_2$ →
   for each $(B_1, (M_1, V_1)) \in \text{infer}_e(B, M, e_1)$     (* $B_1 \sqsubseteq B$ *)
     for each $(B_2, (M_2, V_2)) \in \text{infer}_e(B_1, M_1, e_2)$     (* $B_2 \sqsubseteq B_1$ *)
       for each $(B_3, M_3) \in \text{infer}_u(B_2, M_2, V_2, V_1)$     (* $B_3 \sqsubseteq B_2$ *)
         $D_B = D_B \cup \{(B_3, (M_3, \bot))\}$;
 | $\&x \mid \&(e.fld) \mid \&(*e) \mid *e \mid e.fld \mid \texttt{new}_l \mid e_1; e_2$ →
   similar partitioning with $e_1 := e_2$
 | if$(e_1, e_2)$ →
   $D_B = \text{trim}(\text{infer}_e(B, M, e_1) \sqcup_D \text{infer}_e(B, M, e_2))$
 | $f\ \overline{y}$ →
   $D = \Gamma(f)$;
   $B\_S = \{(B', S) \in \text{reach}(B, M, A[\overline{y}/\overline{x}]) \mid A \text{ appears in } D\}$;
   for each $(B', S) \in B\_S$ {
     (* select one satisfied $P \in D$ for the given calling context $(B', S)$ *)
     $D' = \{(B_i, (M_i, V_i)) \in D \mid B' \vdash B_i[S]\}$;
     $P = (B_i, (M_i, V_i))$; where $(B_i, (M_i, V_i)) \in D' \wedge \neg \exists P' \in D' : P \sqsubseteq_P P'$
     $D_B = D_B \cup \{(B, (M, V_i[S])) \mid (B, M) \in \text{iupdate}(B', M, M_i[S])\}$;
     (* partitioning if needs *)
     for each $(B_i, (M_i, V_i)) \in (D - D')$ {
       if $|BS_g| < k \wedge \neg(\exists B'' \in BS_g : B' \cup B_i[S] \subseteq B'')$ {
         $BS_g = BS_g \cup \{(B' \cup B_i[S])\}$;
         $D_B = D_B \cup \{(B, (M, V_i[S])) \mid (B, M) \in \text{iupdate}(B' \cup B_i[S], M, M_i[S])\}$;
       else skip;
     }
   }
}
return $(D_B)$;
```

Fig. 14. Intraprocedural analysis stage.

```
infer_p($\overline{fundec}$; $e_{main}$) =
  assume $\Gamma$ initially contains summaries of all library procedures
  for each $SCC \in p$ in the bottom-up order {
    initialize the summary of each procedure $f \in SCC$ to the bottom type
      (i.e., $\Gamma(f) = \{(\top_B, (\bot_M, \bot_V))\}$)
    do {
      for each fun $f(\overline{x})$=$e \in SCC$ {
        $BS_g = \{\top_B\}$;   $\Gamma = \Gamma[f \mapsto (\Gamma(f) \sqcup_D \text{infer}_e(\top_B, \epsilon, e))]$;
      }
    } while ($\Gamma$ is changed in the previous iteration)
  }
  return (infer_e($\emptyset, \epsilon, e_{main}$));
```

Fig. 15. Interprocedural analysis stage.

behavior of e_2, partitioning each B_i to B_js again if they are relevant to type e_2. Finally, we update V_1 of infer$_e(B, M, e_1)$ to V_2 of infer$_e(B_i, M_1, e_2)$ with memory M_2 obtained after executing e_2, partitioning B_j to B_ks again if they are identified as relevant.

For 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 Fig. 7. In terms of data flow analysis, 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 $\in D \to D$ operation which safely removes some *inout* elements from the given *summary* (i.e., $D \sqsubseteq_D \text{trim}(D)$). Note that an *inout* $(B_i, (M_i, V_i))$ of D is meaningless if $(B_j, (M_j, V_j)) \in D, B_i \sqsubseteq_B B_j, B_i \vdash M_j \sqsubseteq_M M_i$, 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 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 $\{B_1, \ldots, B_n\}$, 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 iupdate$(B, M, [A_1 \mapsto V_1] \ldots [A_n \mapsto V_n])$ 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 Fig. 15, 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 strongly connected components in the call graph is given.[9] The summary of each procedure is computed using the fixpoint iteration whose termination is discussed in Section 4.5. Note that Γ, 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 infer$_e$ of Section 4.3, the abstract transfer function infer$_e$ is *monotonic* since this case is identical to the memory type system where the [Fundec] rule considers only one input context $\top_B$ (i.e., $\emptyset$). The formal proof of the monotonicity of our memory type system is available in our technical report [17]. It is trivial to show that infer$_e$ is still monotonic even in the presence of the partition operation, since this simply introduces finite input contexts B_is, and the corresponding output for each input context B_i is computed in the same manner as in the memory type system (which is monotonic). However, when we composite the *extensive* function trim to infer$_e$ (i.e., infer$_e$ is a function such that infer$_e = E \circ F$, where F is monotonic and E is extensive), as in Fig. 14, infer$_e$ does not guarantee the monotonicity. In order to ensure that the fixpoint iteration of Fig. 15 always terminates with the postfixpoint (safe) solution of F [18], 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, $F : D \to D$ is monotonic, $E : D \to D$ is extensive, then the sequence*

$$\begin{aligned} D_0 &= \bot_D \\ D_{i+1} &= D_i \sqcup_D (E \circ F)(D_i) \end{aligned}$$

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$_e$ 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 $\forall (B, (M, V)) \in D : B \vdash M' \sqsubseteq_M M \wedge V' \subseteq V$.*

Proof. The proof is trivial by simple induction on the structure of p and e, since the typing result computed by infer$_e$ without the trim operation is the same as the memory type system. □

5. Experiments

5.1. Implementation

To assess the effects of using the update history, we implemented[10] various flow- and context-sensitive bottom-up pointer

[9] We refer to existing work [2] 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 Section 6 without loss of generality.

[10] We added about 7500 lines of OCaml code to CIL [20].

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 [9] 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 [7,8] 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 Section 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 [2]: (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.[11] 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 Fig. 13, 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 paper, 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 s was used in our experiments, which gave us the same precision as in experiments without the timeout feature, while it ran up to 6.29 times faster.

[11] Details of the safe field-sensitive analysis in the presence of type casting can be found elsewhere [2,21,22].

5.2. Experimental results

Table 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,[12] *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 1 lists the average number of targets for a pointer dereference[13] 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 Chatterjee et al. [9]. 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-value or integer-value (e.g., null-pointer) dereferences using the results of the pointer analysis. Column 11 (*safe*) in Table 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 Fig. 16. 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 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 *vpr*) of pointer dereferences were safe, even though this was possible with our analysis "OOO".

[12] We applied a criterion of scalability such that analysis time and memory are limited to 1 h and 1 GB, respectively.

[13] Multilevel dereferencing was counted separately (e.g., $**x$ comprises two dereferences).

Table 1
Experimental results.

Program	$N(L)$	F	S	C	Analysis		Performance		Precision		S-update %	
					Kind	k	Time	Mem	Avg	Safe	Indirect	Inter
mcf	13(1.5)	26	1	1	XXX	1	1	2	1.6	0	0	0
					OXX	1	1	2	1.4	21	0	0
					OOX	1	1	2	1.4	21	5	0
					OOX	2	1	3	1.4	22	5	0
					OOX	5	1	3	1.4	23	5	0
					OOO	1	1	2	1.4	22	5	6
					OOO	2	1	3	1.4	23	5	6
					OOO	5	1	3	1.4	23	5	6
pr	17(2.7)	41	4	2	XXX	1	1	7	2.6	5	0	0
					OXX	1	1	7	2.4	37	0	0
					OOX	1	1	7	2.4	37	15	0
					OOX	2	3	8	1.3	46	15	0
					OOX	5	5	8	1.3	46	14	0
					OOO	1	3	8	2.4	46	15	21
					OOO	2	6	8	1.3	55	15	21
					OOO	5	5	8	1.3	55	15	22
sort	26(3.6)	45	1	2	XXX	1	2	10	1.7	11	0	0
					OXX	1	2	10	1.7	21	0	0
					OOX	1	2	10	1.7	21	24	0
					OOX	2	5	10	1.7	21	24	0
					OOX	5	8	13	1.5	21	24	0
					OOO	1	2	9	1.7	30	24	27
					OOO	2	4	10	1.7	30	24	27
					OOO	5	5	11	1.3	40	24	32
gzip	45(5.7)	89	1	1	XXX	1	3	14	1.4	28	0	0
					OXX	1	3	13	1.3	54	0	0
					OOX	1	3	14	1.3	54	24	0
					OOX	2	8	15	1.3	54	24	0
					OOX	5	12	18	1.1	64	24	0
					OOO	1	3	13	1.2	60	24	24
					OOO	2	6	13	1.2	60	24	23
					OOO	5	10	15	1.0	70	24	24
bzip2	33(3.9)	74	1	1	XXX	1	1	6	1.3	60	0	0
					OXX	1	1	6	1.3	60	0	0
					OOX	1	1	6	1.3	60	95	0
					OOX	2	1	6	1.3	60	95	0
					OOX	5	1	6	1.3	60	95	0
					OOO	1	1	6	1.2	68	95	92
					OOO	2	1	6	1.2	68	95	92
					OOO	5	1	6	1.2	68	95	92
ammp	149(13.7)	179	2	33	XXX	1	96	50	10.9	0	0	0
					OXX	1	96	50	10.8	2	0	0
					OOX	1	95	50	10.8	2	6	0
					OOX	2	136	53	10.8	2	6	0
					OOX	5	195	56	10.8	2	6	0
					OOO	1	150	56	10.6	13	6	24
					OOO	2	188	59	10.5	13	6	24
					OOO	5	241	61	10.5	13	6	24
vpr	151(17.1)	300	7	11	XXX	1	43	48	3.2	0	0	0
					OXX	1	43	48	3.2	1	0	0
					OOX	1	42	48	3.2	1	22	0
					OOX	2	55	49	3.2	1	22	0
					OOX	5	71	54	3.2	1	23	0
					OOO	1	364	67	2.7	38	22	35
					OOO	2	415	67	2.7	38	22	35
					OOO	5	499	75	2.6	38	23	36
crafty	192(22.0)	109	2	1	XXX	1	42	68	1.9	0	0	0
					OXX	1	42	68	1.6	43	0	0
					OOX	1	43	68	1.6	43	41	0
					OOX	2	49	68	1.6	43	41	0
					OOX	5	62	82	1.6	43	41	0
					OOO	1	73	103	1.6	54	41	36
					OOO	2	85	100	1.6	54	41	36
					OOO	5	97	103	1.6	54	41	36
twolf	252(24.9)	191	1	1	XXX	1	60	59	2.7	0	0	0
					OXX	1	60	59	2.7	2	0	0
					OOX	1	60	59	2.7	2	6	0
					OOX	2	82	67	2.7	2	6	0
					OOX	5	115	92	2.7	2	6	0
					OOO	1	213	77	2.4	33	6	32

Table 1 (*continued*)

Program	N(L)	F	S	C	Analysis		Performance		Precision		S-update %	
					Kind	k	Time	Mem	Avg	Safe	Indirect	Inter
					OOO	2	247	92	2.4	33	6	32
					OOO	5	320	150	2.4	33	6	32

N: number of CFG nodes (1 = 1000), *L*: lines of code excluding comments and blank lines (1 = 1000), *F*: 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), and mem: maximum size of the major heap (in megabytes).

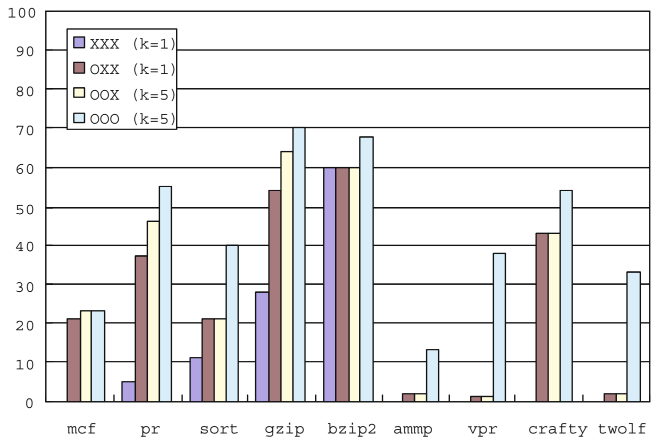

Fig. 16. Effects of using different approaches on a client analysis.

6. Related work

6.1. Bottom-up and flow-sensitive pointer analysis

There are several bottom-up pointer analysis algorithms [7–10] that are flow- and context-sensitive on which we are focusing in this paper.

Harrold and Rothermel [7] and Rountev et al. [8] extend the work of Landi and Ryder [23], 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 [7] and Rountev et al. [8] follow the method of Landi and Ryder [23] 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 [7] 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. [8] 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 [10] 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) [9] 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. [8] 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 [24–26], whereas our technique is based on the subject reduction lemma [16] of type system framework [15].

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 [1–6]. 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 *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 [27,28] 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 [23,12,29,13,14,28,30,31]. 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.

6.4. 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 *unknown* access paths into account which potentially causes the conservative approximations in the bottom-up analysis. Consider the example program in Section 1. 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 $\{g1, g2\}$ 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.

6.5. 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 [13,14] 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. [30] 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. [32] 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 paper.

6.6. Top-down analyses utilizing the reusable summary

There are various forms of pointer-related top-down analyses [13,14,33–36] 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 summary normally being used to improve the performance. Although these approaches 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 [13,14] 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 [13,14]) 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 [13,14], 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 and (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 [37].
- 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.

Acknowledgements

This research was supported by the MKE (Ministry of Knowledge Economy), Korea, under the ITRC (Information Technology Research Center) support program supervised by the institute of information technology advancement (IITA-2008-C1090-0801-0020).

References

[1] M. Fähndrich, J. Rehof, M. Das, Scalable context-sensitive flow analysis using instantiation constraints, in: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, 2000, pp. 253–263.

[2] B.-C. Cheng, W. Mei, W. Hwu, Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation, in: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, 2000, pp. 57–69.

[3] J.S. Foster, M. Fähndrich, A. Aiken, Polymorphic versus monomorphic flow-insensitive points-to analysis for C, in: Proceedings of the Annual International Static Analysis Symposium, 2000, pp. 175–198.
[4] E. Ruf, Effective synchronization removal for Java, in: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, 2000, pp. 208–218.
[5] N. Heintze, O. Tardieu, Ultra-fast aliasing analysis using CLA: a million lines of C code in a second, in: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, 2001, pp. 254–263.
[6] A. Rountev, B.G. Ryder, Points-to and side-effect analyses for programs built with precompiled libraries, in: Proceedings of the International Conference on Compiler Construction, 2002, pp. 20–36.
[7] M.J. Harrold, G. Rothermel, Separate computation of alias information for reuse, IEEE Transactions on Software Engineering (1996) 442–460.
[8] A. Rountev, B.G. Ryder, W. Landi, Data-flow analysis of program fragments, in: ACM SIGPLAN-SIGSOFT Symposium on the Foundations of Software Engineering, 1999, pp. 235–252.
[9] R. Chatterjee, B.G. Ryder, W.A. Landi, Relevant context inference, in: Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 1999, pp. 133–146.
[10] J. Whaley, M.C. Rinard, Compositional pointer and escape analysis for Java programs, in: OOPSLA, 1999, pp. 187–206.
[11] D.R. Chase, M. Wegman, F.K. Zadeck, Analysis of pointers and structures, in: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, 1990, pp. 296–310.
[12] A. Deutsch, Interprocedural may-alias analysis for pointers: beyond *k*-limiting, in: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, 1994, pp. 230–241.
[13] R.P. Wilson, M.S. Lam, Efficient context-sensitive pointer analysis for C programs, in: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, 1995, pp. 1–12.
[14] R.P. Wilson, Efficient, Context-Sensitive Pointer Analysis for C Programs, Ph.D. thesis, Stanford University, December 1997.
[15] B.C. Pierce, Types and Programming Languages, Kluwer Academic Publisher, The MIT Press, 2002.
[16] A.K. Wright, M. Felleisen, A syntactic approach to type soundness, Information and Computation 115 (1) (1994) 38–94.
[17] H.-G. Kang, T. Han, A bottom-up pointer analysis using the update history, Technical Report of the Department of Computer Science at KAIST. <http://cs.kaist.ac.kr/research/technical/Archive/CS-TR-2008-288.pdf>.
[18] P. Cousot, R. Cousot, Abstract interpretation frameworks, Journal of Logic and Computation 2 (4) (1992) 511–547.
[19] F. Nielson, H.R. Nielson, C. Hankin, Principles of Program Analysis, Springer-Verlag, 1999.
[20] G.C. Necula, S. McPeak, S.P. Rahul, W. Weimer, CIL: intermediate languages and tools for analysis and transformation of C programs, in: Proceedings of the International Conference on Compiler Construction, 2002, pp. 213–228.
[21] S.H. Yong, S. Horwitz, T.W. Reps, Pointer analysis for programs with structures and casting, in: PLDI, 1999, pp. 91–103.
[22] D.J. Pearce, P.H.J. Kelly, C. Hankin, Efficient field-sensitive pointer analysis of C, ACM Transactions on Programming Language and Systems 30 (1) (2007).
[23] W. Landi, B.G. Ryder, A safe approximate algorithm for interprocedural pointer aliasing, in: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, 1992, pp. 235–248.
[24] M. Sharir, A. Pnueli, Two approaches to interprocedural data flow analysis, in: Program Flow Analysis: Theory and Application, Kluwer Academic Publisher, Prentice-Hall, 1981, pp. 189–234.
[25] P. Cousot, R. Cousot, Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints, in: Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 1977, pp. 238–252.
[26] P. Cousot, R. Cousot, Modular static program analysis, in: Proceedings of the International Conference on Compiler Construction, 2002, pp. 159–178.
[27] M. Hind, Pointer analysis: haven't we solved this problem yet?, in: ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, ACM, 2001, pp. 54–61.
[28] M. Hind, M. Burke, P. Carini, J.-D. Choi, Interprocedural pointer alias analysis, ACM Transactions on Programming Language and Systems (1999) 848–894.
[29] M. Emami, R. Ghiya, L.J. Hendren, Context-sensitive interprocedural points-to analysis in the presence of function pointers, in: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, 1994, pp. 242–256.
[30] M. Sagiv, T. Reps, R. Wilhelm, Parametric shape analysis via 3-valued logic, ACM Transactions on Programming Language and Systems (2002) 217–298.
[31] J. Zhu, Towards scalable flow and context sensitive pointer analysis, in: W.H.J. Jr., G. Martin, A.B. Kahng (Eds.), DAC, ACM, 2005, pp. 831–836.
[32] S.H. Yong, S. Horwitz, Pointer-range analysis, in: Proceedings of the Annual International Static Analysis Symposium, 2004, pp. 133–148.
[33] N. Rinetzky, M. Sagiv, E. Yahav, Interprocedural shape analysis for cutpoint-free programs, in: C. Hankin, I. Siveroni (Eds.), SAS, Lecture Notes in Computer Science, vol. 3672, Springer, 2005, pp. 284–302.
[34] N. Rinetzky, J. Bauer, T.W. Reps, S. Sagiv, R. Wilhelm, A semantics for procedure local heaps and its abstractions, in: J. Palsberg, M. Abadi (Eds.), POPL, ACM, 2005, pp. 296–309.
[35] A. Gotsman, J. Berdine, B. Cook, Interprocedural shape analysis with separated heap abstractions, in: K. Yi (Ed.), SAS, Lecture Notes in Computer Science, vol. 4134, Springer, 2006, pp. 240–260.
[36] D. Distefano, P.W. O'Hearn, H. Yang, A local shape analysis based on separation logic, in: H. Hermanns, J. Palsberg (Eds.), TACAS, Lecture Notes in Computer Science, vol. 3920, Springer, 2006, pp. 287–302.
[37] H.-G. Kang, Y. Kim, T. Han, H. Han, A path sensitive type system for resource usage verification of C like languages, in: K. Yi (Ed.), APLAS, Lecture Notes in Computer Science, vol. 3780, Springer, 2005, pp. 264–280.