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

A R T I C L E   I N F O

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

A B S T R A C T

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.

© 2008 Elsevier B.V. All rights reserved.

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:

```c
int g1, g2;
void f(int *p, int *q) {
  1: *p = &g1; 2: *q = &g2; 3: return *p;
}

main() {
  int i, j;
  4: f(&i, &i); 5: f(&j, &j); 6: 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, g1, g2, p, q, respectively. The assignment 2 kills the effect of assignment 1 if both p and q initially point to the same memory location, as in the cases of procedure calls at 4 and 5. In this case, the analyzer can safely discard the effect of assignment 1, which is called a strong update [11]. Otherwise, the analyzer needs to keep the side effect of 1 (e.g., the procedure call at 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.
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–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 effectively identifying relevant alias contexts, where the number of relevant alias contexts is bounded.

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 \(k\).
The summary computed for each control flow graph (CFG) node \( n \) of a procedure \( f \) are defined as a set of \textit{inout} elements, where each \textit{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, \( (p, (\{x\} \leftarrow \{j\})) \) computed on line 4 of Fig. 1 abstracts every memory state that is obtainable after line 4 has been executed, since \( B = 0 \) 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 \( AP-V \) to \( M \) while deleting the old side effect \( AP-V \) in \( M \). Note that we kill only the side effect on \( AP \) from \( M \) and do not kill the side effect on 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 \( AP_{l} \) with the memory \( AP_{l-1} \rightarrow V_{l-1} \ldots AP_{0} \rightarrow V_{0} \), alias relations between \( AP_{l} \) and \( AP_{l-1}, \ldots, AP_{0} \) are considered in the analysis. Note that alias relations between \( AP \) and access paths \( \{AP_{l-1}, \ldots, AP_{0}\} \) are not relevant because they do not satisfy condition 3. Alias relations between \( AP \) 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 \( (p, \{x\} \leftarrow \{j\}) \) to the initial memory \( \epsilon \) successively. This means postponing the decision about what side effect of the old update history is killed by an update operation if it depends on the calling context. As mentioned in Section 2.5.2, the expression \( \epsilon_{3} \) on line 5 may be interpreted differently depending on the input context of procedure \( f \). So, the analyzer introduces a new input context \( (q, \{y\} \leftarrow \{r\}) \) and computes the corresponding summary for this input context as well. The side effect for \( p, \{x\} \leftarrow \{j\} \) is killed because the \( \epsilon \)-value of this assignment is a singleton set \( \{p, \} \), and \( p, \) is a unique access path. Finally, procedure \( f \) is summarized with the \textit{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 \( \epsilon \) is satisfiable for this calling context. So, the analyzer applies corresponding side effects \( \{q, \{y\} \leftarrow \{r\} \} \) for this particular calling context with initial memory \( \epsilon \) of procedure \( g \). The underlining in the figure shows the sequence of updates performed for this application. The first line resolves the access path, in which access paths of the callee’s summary are substituted with those of the caller’s summary. The \( \star \) 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 \( \{q, \{y\} \leftarrow \{r\}\} \), since the successive side effect \( \{r, \{k\} \} \) 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
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 $(y_*, \neq z_*)$ and $\emptyset$ are satisfiable depending on the alias context of procedure $g$. So, the analyzer introduces a new input context $(y_*, \neq z_*)$ 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 analysis that was informally presented in Section 2 as a type system. The memory type system both input contexts.

3.1. The language

We will use the following notation: Let $X$ denote an arbitrary syntactic element in our type system, with $X$ used to abbreviate various syntactic enumerations (e.g., $(X, Y)$ abbreviates $(X_1, Y_1,…,X_n, Y_n)$).

The language used to model the pointer behaviors of the C program is defined in Fig. 2. A program $p$ is a sequence of the procedure declarations followed by expression $e$ which represents the body of the main procedure. Assignment statements such as $x := y$ and $s := r := z$ are represented as $&x := y(x)$ and $&((&x y)) := z(x y)$, respectively. Statements such as $x[1] := &y$ and $x[j] := &y$ are translated into one representative statement $x(x[j]) := y$, where the access path for the variable $x$ is safely regarded as the collapsed access path in the analysis. The condition of an $i,$ $t$ 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 $B$, the Inout type $P$, and the Summary type $D$ are explained in Section 2. The TypeEnv type $\Gamma$ is a finite mapping of procedure name to procedure summary.

3.3. Checking aliases between access paths

Judgements $B \vdash_{as} A$ and $B \vdash_{a} B'$ 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 $x_* \neq y_*$. In the given input context $B$, since both of $x_*$ and $y_*$ are unknown access paths. If $x_* \neq y_*$, this means that memory locations abstracted by $x_*$ and $y_*$ are not disjoint (may be aliased).

\footnote{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 $\tau$; 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.}

\footnote{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.

The definition of $P$ is extended to a tuple $(B, M, V)$, where value $V$ represents a return value.}
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 $\epsilon$ (rule [read-\(\varepsilon\)], this indicates that we are reading the value to which $AP$ initially points. This value is represented as abstract access path \(\alpha(AP_\varepsilon)\), 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($\alpha$, $AP$)" is satisfied if one of the following conditions is satisfied: (1) $l \in \text{prefix}(AP)$, where prefix $\in AP \rightarrow 2^k$ 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) $\exists AP, AP' \in \text{Dom}(\alpha) : AP_1 \neq AP_2 \wedge \alpha(AP_1) = \alpha(AP_2) = AP$. Otherwise, $AP$ is not a collapsed access path, which is denoted as "unique($\alpha$, $AP$)".

There are two cases if the examined current update history is $M(AP \rightarrow V_1)$: (1) if $AP$ is a unique access path, then the value of $AP$ is $V_1$, since $[AP \rightarrow V_1]$ must update the $AP$ (rule [read-\(\varepsilon\)]) 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 $[AP \rightarrow V_1]$ may update the $AP$ (rule [read-safe]).

There are also two cases if the examined current update history is $M(AP_1 \rightarrow V_1)$: (1) if $AP_1$ is disjoint with $AP$, then we read the value of $AP$ in memory $M$ since $[AP_1 \rightarrow V_1]$ 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 $[AP_1 \rightarrow V_1]$ may update the location represented by $AP$ (rule [read-safe])

Now, judgement $B, M \vdash_{A \Rightarrow V}$ is added to the type system that specifies the safe approximation of pointed access paths $V$ of the set of access paths $A$ with $M$ under $B$. Typing rule [read-\(\varepsilon\)] 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 $AP$, 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 $M_1 \sqsubseteq M_2$) 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 $[\sqsubseteq_M(2)]$), $M_1 \sqsubseteq_M M_2$ determines $M_1 \sqsubseteq_M M_2$,
[shift]

\[
\begin{align*}
\text{shift}(B, \epsilon, AP) &= [AP \mapsto \{\alpha(AP, \epsilon)\}] \\
\text{shift}(B, M[AP \mapsto V], AP) &= M[AP \mapsto V] \\
\text{shift}(B, M[AP' \mapsto V'], AP) &= \\
&= \begin{cases} 
\text{let shift}(B, M, AP) = M[AP \mapsto V] & \text{in} \\
\text{if } B \vdash AP \# AP' \lor \text{collapsed}(\alpha, AP) \text{ then } M'[AP' \mapsto V'][AP \mapsto V] \\
\text{else } M'[AP' \mapsto V'][AP \mapsto V \cup V']
\end{cases}
\end{align*}
\]

[Subtype relation for the memory type]

\[
\begin{align*}
[M(1)] & \quad \vdash M \sqsubseteq M \text{ shift}(B, M, AP) \\
[M(2)] & \quad \vdash \text{shift}(B, M_1, AP) \sqsubseteq M_2[AP \mapsto V] \\
[M(3)] & \quad \vdash B \vdash M \sqsubseteq M' V \subseteq V'
\end{align*}
\]

[Join operation for the memory type]

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

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 \( M_1 \). The \( \text{shift}(B, M_1, AP) \) 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 \( \text{shift}(B, M_1, AP) \) always contains those of \( M_1 \) under input context \( B \). In other words, \( \text{shift}(B, M_1, AP) \) computes an \( M_3 \) such that the last update history of \( M_3 \) is \( [AP \mapsto V'] \), and for every abstract access path \( AP \), of procedure \( f \), if \( B, M_1 \triangleright AP \theta \Rightarrow V_1 \) and \( B, M_1 \triangleright AP \theta \Rightarrow V_1 \), then \( V_1 \subseteq V_\theta \).

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

The typing rules for the memory join operation \(([\sqcup_M])\) are given in 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 \( \sim \) such that \( X_1 \sim X_2 \) if and only if \( X_1 \equiv X_2 \) and \( X_2 \equiv X_1 \). [18] Note that we omitted the definition of the greatest element \( \sqcup_X \) and the least element \( \sqcap_X \) for each type \( X \), where the corresponding extensions are trivial.

3.6. Memory updates

The judgement \( B, M \vdash \epsilon \Rightarrow M' \) 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 \((\text{update-s})\) is explained in Section 2.4, where \( M \triangleright \{AP\} \) refers to the memory after removing mapping \( [AP \mapsto V] \) (if it exists) from \( M \).
If we are updating a collapsed access path \( AP \) to \( V \) with \( M[\text{update-w}] \), we do not know for sure which concrete access path represented by \( AP \) is updated by this update operation. Therefore, the old side effect \( \frac{AP}{V} \) cannot be safely killed when the new side effect \( \frac{AP}{V} \) is added to \( M \). On the other hand, simply adding \( \frac{AP}{V} \) to \( M \) to keep the old side effect \( \frac{AP}{V} \) 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 \( \frac{M[\text{leave}] AP}{V} \), where the side effect for \( AP \) is safely 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 \( \frac{AP}{V} \) to \( M[\text{leave}] \). The update operation for the multiple access paths ([update-w]) is explained similarly.

### Typing rules for the program

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

**Judgement 1**

(1) \( p \vdash p \Rightarrow (M, V) \).
(2) \( f \vdash \text{fun} f \Rightarrow e \).
(3) \( B, M \vdash e \Rightarrow M', V \).

Judgement 1 is read as “The type of a program \( p \) is \( (M, V) \)”. This means that the safe environment \( M \) and \( V \) are obtained of program \( p \) is \( M \) and \( V \). Judgement 2 is read as “\( f \) types \( f \)”, which means that \( f \) 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 \( \Gamma \), which provides the safe summary of each procedure \( f_i \) invoked in \( e \). In the type system, we simply assume that such \( \Gamma \) for a program are given globally. The inference algorithm of Section 4 describes the algorithm to infer such \( \Gamma \). In [Fundec], we first prepare a set of input contexts \( B_1, \ldots, B_n \), where each \( B_i \) is consid-
If each \( B_i \) 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 \( \Gamma \) typed to type procedure \( f \) contains its own summary, which explains the typing for recursion and mutually recursive procedures.

\[ (B, M \vdash e \& x) \Rightarrow M, \{ x \} \]

\[ (B, M \vdash e \& V) \Rightarrow M_1, V \]

\[ (B, M \vdash e \& (c \cdot e)) \Rightarrow M_1, A \]

\[ (B, M \vdash e \Rightarrow M_1, V_1, B, M_1 \vdash e \Rightarrow V) \]

\[ (B, M \vdash e \Rightarrow M_1, V_1, B, M_1 \vdash \text{field}(V_1, fld) \Rightarrow V) \]

\[ (B, M \vdash e \Rightarrow M_1, V_1, B, M_1 \vdash e \Rightarrow V_1 \Rightarrow M) \]

\[ (B, M \vdash e \Rightarrow M_1, V_1, B, M_1 \vdash \text{field}(V_1, fld) \Rightarrow V) \]

\[ B, M \vdash e \Rightarrow M_1, V_1, B, M_1 \vdash e \Rightarrow M_2, V_2, B, M_2 \vdash e \Rightarrow V_1 \Rightarrow M_3 \]

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

\[ B, M \vdash e \Rightarrow M_1, V_1, B, M_1 \vdash e \Rightarrow M_2, V_2 \]

\[ B, M \vdash e \Rightarrow M_1, V_1 \Rightarrow M_2 \U V_2 \]

\[ B, M \vdash e \Rightarrow M_1, V_1 \Rightarrow M_2 \U V_2 \]

The purpose of \( B \cdot B[M_i \vdash e] \) is not to check any property for verification as in the typical type system, but to select an \( \text{inout} \) \((B_i, (M_i, V_i))\), 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 \( \psi \) (which means all calling contexts) exists in the summary. Finally, we iteratively perform each instantiated update \([A_0 \to V_i]_M \cdot S\) with the caller’s memory \( M \) and input context \( B \) (see the definition of \( M \cdot A[M_i \vdash e] \) and the substitution in Fig. 9).

\[ \text{New} \] can be simply regarded as a procedure application whose summary is \( \{ (\emptyset, (c, \{ i \})) \} \), where \( i \) 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 losing 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 \( \text{malloc} \ l \cdot (\text{sizeof}(\text{struct} \cdot \text{structlist})) \), where \( \text{list} \) is a structure that has two fields \( d \) and \( n \). We can use a refined \( \pi \) that distinguishes...
the fields of the structure list. We can also relate the call sequence to the allocation point using a tuple \((l, \sigma)\), where the type variable \(\sigma\) is instantiated with the caller's label \(l\) at the call site, which yields \((l, \{l\}, \sigma)\). In this case, we limit the size of the call sequence set to some constant \(k\).

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

### 3.8. Typing example

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

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

int *sp, int *sq, int **sr = NULL;

f1(sp, sq, sr) {
    if (?) {
        sp = malloc(sizeof(int));
        return;
    }
    *(sq++) = *(sr++);
}

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 \(\pi_{\pi_q} = \pi_{\pi_r}\) 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{t}_1] and [\text{k}_4] are the type derivation trees\footnote{We marked the used rule at the right-hand side of each type derivation tree, and the abbreviation \text{[\text{t}]} is used to denote the type derivation tree \text{[\text{t}]} } for the assignments I1 and I2 of procedure \text{f1}, respectively. Note that we simply assume that access path abstraction function \text{/put} and set of relevant input contexts \text{B}_1, \ldots, \text{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 \text{x} such that access paths \text{p.x} and \text{q.x} are unique access paths and the relevant input contexts of \text{f1} are \emptyset and \{p. \# q.x\}. 

\{\text{t}_2\} computes the join of \{p. \# \{\text{t}_1\}\} and \{q. \# \{\text{t}_1\}\} which are the typing results of each branch. Note that the result of \text{shift}([\text{B}.\{p. \# \{\text{t}_1\}\}, q.x]) which is used in the first depth of the type derivation tree is dependent on input context \text{B}. If \text{B} = \emptyset, then \text{V}_1 = \{q. \# \text{t}_1\}. If \text{B} = \text{I1}, then \text{V}_1 = \{q. \# \text{t}_1\}. \text{[\text{t}_3]} is the type derivation tree for the \text{let} expression, which is typed by the typing rule \text{[If]} with the type derivation trees \text{[\text{t}2], \text{[\text{t}4]}}, and \text{[\text{t}6]} is the type derivation tree for computing the summary of procedure \text{f1}. For each relevant input context \emptyset and \{p. \# \{\text{t}_4\}\}, the corresponding typing results of \text{[\text{t}_6]} are added to the summary.

\{\text{t}_4\} is the type derivation tree for the procedure call in procedure \text{f2}. Since the typing result of \text{g2} = \text{g1} is \text{g2}{\downarrow}\{g1.\#\}, access paths \text{p.x} and \text{q.x} of the summary of procedure \text{f1} are resolved as \text{g1.\#} and \text{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 \text{p}.

Proof. The full formal proof is available in our technical report \cite{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 \text{I} which is simply assumed to be given\footnote{The access path abstraction function \text{/put} discussed in Section 3.4 can be determined before computing the summary of each procedure using the \text{C}_\text{type} information of each variable. For example, the abstract access paths for a variable whose \text{C}_\text{type} is a recursive structure are statically determined in the manner discussed in Section 2.1.} in the type system. Note that [\text{Fundec}] recursively uses \text{I} to specify the conditions for \text{I} to safely approximate the behavior of procedure \text{f}, which means that it is simply a recursive equation that is usually solved by a fixpoint iteration algorithm \cite{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 \text{I}, 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{npinfer}_\text{I1} are recursively defined using the fixpoint iteration for \text{B} = \text{Bool}, \text{npinfer}_\text{I2} \in \text{B} \times \text{B} \rightarrow \text{Boo}, \text{npinfer}_\text{I3} \in \text{B} \times \text{M} \times \text{AP} \rightarrow \text{V}, \text{npinfer}_\text{I4} \in \text{B} \times \text{M} \times \text{A} \rightarrow \text{V}. Since we are interested in computing the summary of each procedure for the fixed input context \emptyset, which abstracts every input context, is trivially defined using the fixpoint iteration for \text{I} 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 partition(\text{B}, \text{M}, \text{A}) operation of Fig. 13 computes a set of input contexts \{\text{B}_1, \ldots, \text{B}_n\} partitioned from \text{B} (i.e., \text{B} \uplus \text{B}) that are identified as relevant for determining the set of pointed access paths of \text{A} when the sequence of side effects is \text{M}. The global variable \text{K} in this operation is used to improve the efficiency of the inference algorithm by avoiding duplicated partitioning, and the global variable \text{K} means the partitioning bound.

The partitioning read operation \text{npinfer} \in \text{B} \times \text{M} \times \text{A} \rightarrow \text{2}^{\text{AP}} \subset \text{V}, which uses the given \text{B}, \text{M}, and \text{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 \text{I} explained in Section 4.4.
4.3. Intraprocedural analysis stage: infer

The intraprocedural analysis stage of our inference algorithm is defined by operation infer in B × M × e → D as shown in Fig. 14. infer(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 can be considered as an abstract transfer function for node ne of a procedure f. infer computes the safe approximation D for the execution results of node ne when the side effect from the start of procedure f to the predecessor of ne 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, V_e))\}. If e behaves differently depending on B, which can be effectively identified by the partitioning operations of Section 4.2, we introduce such Bi for the analysis and compute the corresponding results as \{(B, (M, V_e)), (B_1, (M_1, V_{e_1})), ..., (B_n, (M_n, V_{e_n}))\}.

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

\[
\text{infer}(B, M, A) = \begin{cases} 
BS = \{B\}; & \text{for each } AP \in A \{ 
\begin{aligned}
&\text{assume } M = M'[AP \mapsto V][AP_1 \mapsto V_1] \cdots [AP_n \mapsto V_n] \\
&B_4 = B \cup \{AP \# AP_1 \mid AP_1 \neq \text{aliasable}(AP, \{AP_1, \ldots, AP_n\})\}; \\
&\text{if } (\exists B' \in BS : B' \subseteq B') \land |BS| < k \text{ then} \\
&BS = BS \cup \{B_4\}; BS_2 = BS_2 \cup \{B_4\}; k = k + 1;
\end{aligned}
\end{cases}
\]

\]

\[
\text{return (BS)}; \\
\text{infer}(B, M, A) = \text{return } \{(B_i, np\text{_{infer}}(B_i, M, A)) \mid B_i \in \text{partition}(B, M, A)\};
\]

Fig. 13. Partitioning operations.

Fig. 14. Intraprocedural analysis stage.
We added about 7500 lines of OCaml code to CIL.  

4.4. Interprocedural analysis stage: the overall input coverage of the procedure summary.

For \( f: A \rightarrow B\), the reachability of the callee \( B\) via the \( A\) path 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, (i.e., infer, is a function such that \( \text{infer}_e = E \circ F\), where \( F\) is monotonic and \( E\) is extensive), as in Fig. 14, infer, does not guarantee the monotonicity. In order to ensure that the fixpoint iteration of Fig. 15 always terminates with the postfixpoint (safe) solution of \( F\), we define the sequence \( D_0, \ldots, D_i\) of our fixpoint iteration based on Theorem 2.

**Theorem 2** (Termination with postfixpoint). If \( D\) is finite, \( F: D \rightarrow D\) is monotonic, \( E: D \rightarrow D\) is extensive, then the sequence \( D_0 = \bot_D\), \( D_{i+1} = D_{i} \cup_D (E \circ F)(D_i)\) is stationary and its limit is the postfixpoint of \( F\).

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

Now, the soundness of the inference algorithm is proved in Theorem 3. Note that we use the monotonic function \( F\) (infer, 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 \( \vdash p = (M, V)\), \( \text{infer}_e(p) = D\), and trim is not used, then \( \forall (B, (M, V)) \in D: B \supseteq M \cup M' \land \land V \subseteq V\).

**Proof.** The proof is trivial by simple induction on the structure of \( p\) and \( e\), since the typing result computed by infer, without the trim operation is the same as the memory type system.
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., \( x = \& 1 \)), (2) indirect assignments (e.g., \( \star x = \& 1 \)), 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. 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 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 immediately 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.

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, 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 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".

---

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

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., \( \star \star x \) comprises two dereferences).
<table>
<thead>
<tr>
<th>Program</th>
<th>$N(L)$</th>
<th>$F$</th>
<th>$S$</th>
<th>$C$</th>
<th>Analysis Performance</th>
<th>Precision</th>
<th>$S$-update %</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Kind $k$ Time Mem Avg Safe Indirect Inter</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mcf</td>
<td>13(1.5)</td>
<td>26</td>
<td>1</td>
<td>1</td>
<td>XXX 1 1 2 1.6 0 0 0 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OXX 1 1 2 1.4 21 0 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OOX 1 1 2 1.4 21 5 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OOX 1 2 1 3 1.4 22 5 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OOX 1 3 2 1.4 22 5 6</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OOO 1 2 1 3 1.4 23 5 6</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OOO 5 1 3 1.4 23 5 6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pr</td>
<td>17(2.7)</td>
<td>41</td>
<td>4</td>
<td>2</td>
<td>XXX 1 1 7 2.6 5 0 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OXX 1 1 7 2.4 37 0 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OOX 1 1 7 2.4 37 15 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OOX 2 3 8 1.3 46 15 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OOX 5 5 8 1.3 46 14 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OOO 1 3 8 2.4 46 15 21</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OOO 2 6 8 1.3 55 15 21</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OOO 5 5 8 1.3 55 15 22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sort</td>
<td>26(3.6)</td>
<td>45</td>
<td>1</td>
<td>2</td>
<td>XXX 1 2 10 1.7 11 0 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OXX 1 2 10 1.7 21 0 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OOX 1 2 10 1.7 21 24 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OOX 2 5 10 1.7 21 24 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OOX 5 8 13 1.5 21 24 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OOO 1 2 9 1.7 30 24 27</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OOO 2 4 10 1.7 30 24 27</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OOO 5 5 11 1.3 40 24 32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>gzip</td>
<td>45(5.7)</td>
<td>89</td>
<td>1</td>
<td>1</td>
<td>XXX 1 3 14 1.4 28 0 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OXX 1 3 13 1.3 54 0 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OOX 1 3 14 1.3 54 24 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OOX 2 8 15 1.3 54 24 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OOX 5 12 18 1.1 64 24 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OOO 1 3 13 1.2 60 24 24</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OOO 2 6 13 1.2 60 24 24</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OOO 5 10 15 1.0 70 24 24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bzip2</td>
<td>33(3.9)</td>
<td>74</td>
<td>1</td>
<td>1</td>
<td>XXX 1 1 6 1.3 60 0 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OXX 1 1 6 1.3 60 95 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OOX 1 2 6 1.3 60 95 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OOX 5 1 6 1.3 60 95 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OOO 1 1 6 1.2 68 95 92</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OOO 2 1 6 1.2 68 95 92</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OOO 5 1 6 1.2 68 95 92</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ammp</td>
<td>149(13.7)</td>
<td>179</td>
<td>2</td>
<td>33</td>
<td>XXX 1 96 50 10.9 0 0 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OXX 1 96 50 10.8 2 0 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OOX 1 95 50 10.8 2 6 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OOX 2 136 53 10.8 2 6 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OOX 5 195 56 10.8 2 6 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OOO 1 150 56 10.6 13 6 24</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OOO 2 188 59 10.5 13 6 24</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OOO 5 241 61 10.5 13 6 24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>vpr</td>
<td>151(17.1)</td>
<td>300</td>
<td>7</td>
<td>11</td>
<td>XXX 1 43 48 3.2 0 0 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OXX 1 43 48 3.2 1 0 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OOX 1 42 48 3.2 1 22 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OOX 2 55 49 3.2 1 22 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OOX 5 71 54 3.2 1 23 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OOO 1 364 67 2.7 38 22 35</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OOO 2 415 67 2.7 38 22 35</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OOO 5 489 75 2.6 38 23 36</td>
<td></td>
<td></td>
</tr>
<tr>
<td>crafty</td>
<td>192(22.0)</td>
<td>109</td>
<td>2</td>
<td>1</td>
<td>XXX 1 42 68 1.9 0 0 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OXX 1 42 68 1.6 43 0 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OOX 1 43 68 1.6 43 41 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OOX 2 49 68 1.6 43 41 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OOX 5 62 82 1.6 43 41 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OOO 1 73 103 1.6 54 41 36</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OOO 2 85 100 1.6 54 41 36</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OOO 5 97 103 1.6 54 41 36</td>
<td></td>
<td></td>
</tr>
<tr>
<td>twolf</td>
<td>252(24.9)</td>
<td>191</td>
<td>1</td>
<td>1</td>
<td>XXX 1 60 59 2.7 0 0 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OXX 1 60 59 2.7 2 0 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OOX 1 60 59 2.7 2 6 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OOX 2 82 67 2.7 2 6 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OOX 5 115 92 2.7 2 6 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OOO 1 213 77 2.4 33 6 32</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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 interprocedural 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 disadvantage 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:

```c
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 “OOX”, “OXX”, 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

Table 1 (continued)

<table>
<thead>
<tr>
<th>Program</th>
<th>N(L)</th>
<th>F</th>
<th>S</th>
<th>C</th>
<th>Analysis</th>
<th>Performance</th>
<th>Precision</th>
<th>S-update %</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Time</td>
<td>Mem</td>
<td>Indirect</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Avg</td>
<td>Safe</td>
<td></td>
</tr>
<tr>
<td>OOO</td>
<td>2</td>
<td>247</td>
<td>92</td>
<td>2.4</td>
<td>33</td>
<td></td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>OOO</td>
<td>5</td>
<td>320</td>
<td>150</td>
<td>2.4</td>
<td>33</td>
<td></td>
<td></td>
<td>6</td>
</tr>
</tbody>
</table>

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).
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 \(C_\text{a}\)), the return value of procedure \(f\) is conservatively approximated as \(\{g_1, g_2\}\) due to the potential alias between \(p.\) and \(q.\). Although we attempted to avoid such approximations by distinguishing the behaviors of a procedure for the relevant alias contexts, such kinds of approximations still remain since a procedure can involve an exponential potential number of relevant alias contexts, and we limit the number of alias contexts considered in the summary to a constant bound \(k\). On the contrary, top-down analyses need not introduce such kinds of approximations since the concrete calling context can be always determined before analyzing the behavior of the procedure.

6.5. 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 as the stack variable can be used as the recursive and the selectors may not involve 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 only accurately 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


