Precise and Compact Modular Procedure Summaries for Heap Manipulating Programs [*]

Isil Dillig
Department of Computer Science
Stanford University
isil@cs.stanford.edu

Thomas Dillig
Department of Computer Science
Stanford University
tdillig@cs.stanford.edu

Alex Aiken
Department of Computer Science
Stanford University
aiken@cs.stanford.edu

Mooly Sagiv
Department of Computer Science
Tel Aviv University
msagiv@post.tau.ac.il

Abstract

We present a strictly bottom-up, summary-based, and precise heap analysis targeted for program verification that performs strong updates to heap locations at call sites. We first present a theory of heap decompositions that forms the basis of our approach; we then describe a full analysis algorithm that is fully symbolic and efficient. We demonstrate the precision and scalability of our approach for verification of real C and C++ programs.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verication

General Terms Languages, Verification, Experimentation

1. Introduction

It is well-known that precise static reasoning about the heap is a key requirement for successful verification of real-world software. In standard imperative languages, such as Java, C, and C++, much of the interesting computation happens as values flow in and out of the heap, making it crucial to use a precise, context- and flow-sensitive heap analysis in program verification tools. Flow-sensitivity, in particular, enables *strong updates*. Informally, when analyzing an assignment `a := b`, a strong update replaces the analysis information for `a` with the analysis information for `b`. This natural rule is unsound if `a` is a *summary location*, meaning `a` may represent more than one concrete location. In previous work there is an apparent tension between scalability and precision in heap analysis:

- For scalability, it is desirable to analyze the program in pieces, for example, one function at a time. Many of the most scalable analyses in the literature are modular [1, 2].
- For precision, a large body of empirical evidence shows it is necessary to perform strong updates wherever possible [3, 4].

It is not obvious, however, how to perform strong updates in a modular heap analysis. Consider a function `h(x, y){e}`. When analyzing `h` in isolation, we do not know how many, or which, locations `x` and `y` may point to at a call site of `h`—it may be many (if either `x` or `y` is a summary location), two, or even one (if `x` and `y` are aliases). Without this information, we cannot safely apply strong updates to `x` and `y` in `e`. Thus, while there is a large body of existing work on flow- and context-sensitive heap analysis, most algorithms for this purpose either perform a whole-program analysis or perform strong updates under very restrictive conditions.

In this paper, we present a modular, strictly bottom-up, flow- and context-sensitive heap analysis that uses summaries to apply strong updates to heap locations at call sites. As corraborated by our experiments, strong updates are crucial for the level of precision required for successful verification. Furthermore, we are interested in a modular, summary-based analysis because it offers the following key advantages over a whole program analysis:

- *Reuse of analysis results:* A major problem with whole-program analysis is that results for a particular program component cannot be reused, since functions are analyzed in a particular context. For instance, adding a single caller to a library may require complete re-analysis of the entire library. In contrast, modular analyses allow complete reuse of analysis results because procedure summaries are valid in any context.
- *Analysis scalability:* Function summaries express a function's behavior in terms of its input/output interface, abstracting away its internal details. We show experimentally that our function summaries do not grow with program size; thus, an implementation strategy that analyzes a single function at a time, requiring only one function and its callee's summaries to be in memory, should scale to arbitrarily large programs.
- *Parallelizability:* In modular analysis, any two functions that do not have a caller/callee relationship can be analyzed in parallel. Thus, such analyses naturally exploit multi-core machines.

[*] This work was supported by grants from NSF and DARPA (CCF-0430378, CCF-0702681).

PLDI'11, June 4–8, 2011, San Jose, California, USA.
Copyright © 2011 ACM 978-1-4503-0663-8/11/06...

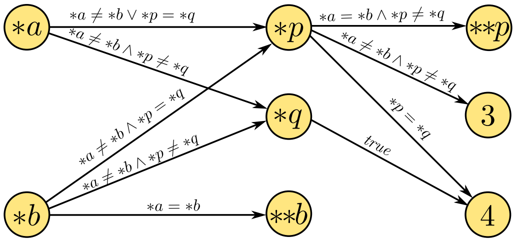

$*a$
$*a \neq *b \vee *p = *q$
$*p$
$*a = *b \wedge *p \neq *q$
$**p$
$*a \neq *b \wedge *p \neq *q$
$*a \neq *b \wedge *p \neq *q$
$*a \neq *b \wedge *p = *q$
$*q$
3
$*p = *q$
$*a \neq *b \wedge *p \neq *q$
true
$*b$
$*a = *b$
$**b$
4

Figure 1. Summary associated with function `f`

To illustrate our approach, consider the following simple function `f` along with its three callers `g1`, `g2`, and `g3`:

```
void f(int** a, int** b, int* p, int* q) {
  *a = p; *b = q; **a = 3; **b = 4; }
```

```
void g1() {             void g2() {             void g3() {
  int** a, int** b;       int** a, int** b;       int** a, int** b;
  a = new int*;           a = new int*;           a = new int*;
  b = new int*;           b = new int*;           b = a;
  int p = 0, q=0;         int p = 0;              int p = 0, q=0;
  f(a, b, &p, &q);        f(a, b, &p, &p);        f(a, b, &p, &q);
  assert(p == 3); }       assert(p == 4); }       assert(p == 0); }
```

Here, although the body of `f` is conditional- and loop-free, the value of `*p` after the call to `f` may be either 3, 4, or remain its initial value. In particular, in contexts where `p` and `q` are aliases (e.g., `g2`), `*p` is set to 4; in contexts where neither `a` and `b` nor `p` and `q` are aliases (e.g., `g1`), `*p` is set to 3, and in contexts where `a` and `b` are aliases but `p` and `q` are not (e.g., `g3`), the value of `*p` is unchanged after a call to `f`. Furthermore, to discharge the assertions in `g1`, `g2`, and `g3`, we need to perform strong updates to all the memory locations.

To give the reader a flavor of our technique, the function summary of `f` computed by our analysis is shown in Figure 1, which shows the points-to graph on exit from `f` (i.e., the heap when `f` returns). Here, points-to edges between locations are qualified by constraints, indicating the condition under which this points-to relation holds. The meaning of a constraint such as $*p = *q$ is that the location pointed to by `p` and the location pointed to by `q` are the same, i.e., `p` and `q` are aliases. Observe that Figure 1 encodes all possible updates to `*p` precisely: In particular, this summary indicates that `*p` has value 3 under constraint $*a \neq *b \wedge *p \neq *q$ (i.e., neither `a` and `b` nor `p` and `q` are aliases); `*p` has value 4 if `p` and `q` are aliases, and `*p` retains its initial value (`**p`) otherwise.

There are three main insights underlying our approach:

- First, we observe that a heap abstraction H at any call site of `f` can be overapproximated as the finite union of some structurally distinct *skeletal points-to graphs* $\hat{H}_1, \dots \hat{H}_m$ where each abstract location points-to *at most one* location. This observation yields a naive, but sound, way of performing summary-based analysis where the heap state after a call to function `f` is conditioned upon the skeletal graph at the call site.
- Second, we symbolically encode all possible skeletal heaps on entry to `f` in a *single* symbolic heap where points-to edges are qualified by constraints. This insight allows us to obtain a single, *polymorphic* heap summary valid at any call site.
- Third, we observe that using summaries to apply strong updates at call sites requires a negation operation on constraints. Since these constraints may be approximations, simultaneous reasoning about may and must information on points-to relations is necessary for applying strong updates when safe. To solve this difficulty, we use *bracketing constraints* [4].

The first insight, developed in Section 2, forms the basic framework for reasoning about the correctness and precision of our approach. The second and third insights, exploited in Section 4, yield a symbolic and efficient encoding of the basic approach. To summarize, this paper makes the following contributions:

- We develop a theory of abstract heap decompositions that elucidates the basic principle underlying modular heap analyses. This theory shows that a summary-based analysis must lose extra precision over a non-summary based analysis in some circumstances and also sheds light on the correctness of earlier work on modular alias analyses, such as [5–7].
- We present a full algorithm for performing modular heap analysis in a symbolic and efficient way. While our algorithm builds on the work of [7] in predicating summaries on aliasing patterns, our approach is much more precise and is capable of performing strong updates to heap locations at call sites.
- We demonstrate experimentally that our approach is both scalable and precise for verifying properties about real C and C++ applications up to 100,000 lines of code.

2. Foundations of Modular Heap Analysis

As mentioned in Section 1, our goal is to analyze a function f independently of its callers and generate a summary valid in any context. The main difficulty for such an analysis is that f's *heap fragment* (the portion of the program's heap reachable through f's arguments and global variables on entry to f) is unknown and may be arbitrarily complex, but a modular analysis must model this unknown heap fragment in a conservative way.

Our technique models f's heap fragment using abstractions $H_1, \dots, H_k$ such that (i) in each H_i, every location points to *exactly one location variable* representing the unknown points-to targets of that location on function entry, (ii) each H_i represents a distinct aliasing pattern that may arise in some calling context, and (iii) the heap fragment reachable in f at any call site is overapproximated by combining a subset of the heaps in $H_1, \dots, H_k$.

As the above discussion illustrates, our approach requires representing the heap abstraction at any call site as the finite union of heap abstractions where each pointer location has exactly one target. We observe that every known modular heap analysis, including ours, has this this *one-target* property. In principle, one could allow the unknown locations in a function's initial heap fragment to point to 2, 3, or any number of other unknown heap locations, but it is unclear how to pick the number or take advantage of the potential extra precision.

In this section, we present *canonical decompositions*, through which the heap is decomposed into a set of heaps with the one-target property, and *structural decompositions*, which coalesce isomorphic canonical heaps. We then show how these decompositions can be used for summary-based heap analysis.

2.1 Preliminaries

We describe the basic ideas on a standard *may points-to* graph, which we usually call a *heap* for brevity. A labeled node A represents one or more concrete memory locations $\zeta(A)$.

DEFINITION 1. **(Summary Location)** *An abstract location that may represent multiple concrete locations is a* summary location *(e.g., modeling elements in an array/list). An abstract location representing exactly one concrete location is a* non-summary location.

For any two distinct abstract locations A and A', we require $\zeta(A) \cap \zeta(A') = \emptyset$, and that $|\zeta(A)| = 1$ if A is a non-summary node. An edge (A, B) in the points-to graph denotes a partial function $\zeta_{(A,B)}$ from pointer locations in $\zeta(A)$ to locations in

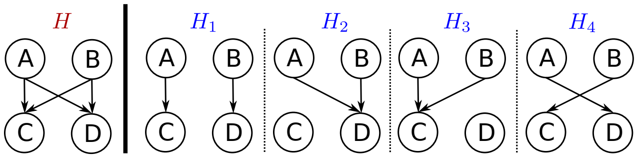


Figure 2. A heap H and its canonical decomposition $H_1, \ldots, H_4$

$\zeta(B)$, with the requirement that for every pointer location $l \in \zeta(A)$ there is exactly one node B such that $\zeta_{(A,B)}(l)$ is defined (i.e., each pointer location has a unique target in a concrete heap). Finally, each possible choice of ζ and compatible edge functions $\zeta_{(A,B)}$ for each edge (A, B) maps a points-to graph H to one concrete heap. We write $\gamma(H)$ for the set of all such possible concrete heaps for the points-to graph H. We also write $H_1 \sqsupseteq H_2$ if $\gamma(H_1) \supseteq \gamma(H_2)$, and $H_1 \sqcup H_2$ for the heap that is the union of all nodes and edges in H_1 and H_2. We define a semantic judgment $H \models S : H'$ as:

$$H \models S : H' \Leftrightarrow \forall h \in \gamma(H).\ \exists h' \in \gamma(H').\ eval(h, S) = h'$$

where $eval(h, S)$ is the result of executing code fragment S starting with concrete heap h. Now, we write $H \vdash_a S : H'$ to indicate that, given a points-to graph H and a program fragment S, H' is the new heap obtained after analyzing S using pointer analysis a. The pointer analysis a is sound if for all program fragments S:

$$H \vdash_a S : H' \ \Rightarrow\ H \models S : H'$$

2.2 Canonical Decomposition

In this section, we describe how to express a points-to graph H as the union of a set of points-to graphs $H_1, \ldots, H_k$ where in each H_i, every abstract location points to *at most one* location.

DEFINITION 2. **(Canonical points-to graph)** *We say a points-to graph is* canonical *if every abstract memory location has an edge to at most one abstract memory location.*

DEFINITION 3. **(Canonical decomposition)** *The canonical decomposition of heap H is obtained by applying these steps in order:*

1. *If a summary node A points to multiple locations $T_1, \ldots, T_k$, replace $T_1, \ldots, T_k$ with a single summary node T such that any edge to/from any T_i is replaced with an edge to/from T.*
2. *Let B be a location with multiple edges to $T_1, \ldots, T_k$. Split the heap into $H_1, \ldots, H_k$ where in each H_i, B has exactly one edge to T_i, and recursively apply this rule to each H_i.*

LEMMA 1. *Let $H_1, \ldots, H_k$ be the canonical decomposition of H. Then $(H_1 \sqcup \ldots \sqcup H_k) \sqsupseteq H$.*

PROOF 1. *Let H' be the heap obtained from step 1 of Definition 3. To show $H' \sqsupseteq H$ we must show $\gamma(H') \supseteq \gamma(H)$. Let $h \in \gamma(H)$ and let ζ_H by the corresponding mapping. We choose $\zeta^{H'}(T) = \zeta^H(T_1) \cup ... \cup \zeta^H(T_k)$ and $\zeta^{H'}(X) = \zeta^H(X)$ otherwise, and construct the edge mappings $\zeta^H_{(A,B)}$ from $\zeta^{H'}_{(A,B)}$ analogously. Thus, $h \in \gamma(H')$ and we have $\gamma(H') \supseteq \gamma(H)$. In step 2, observe that any location B with multiple edges to $T_1, \ldots, T_k$ must be a non-summary location. Hence, the only concrete location represented by B must point to exactly one T_i in any execution. Thus, in this step, $(H_1 \sqcup \ldots \sqcup H_k) = H' \sqsupseteq H$.* □

EXAMPLE 1. *Figure 2 shows a heap H with only non-summary locations. The canonical decomposition of H is H_1, H_2, H_3, H_4, representing four different concrete heaps encoded by H.*

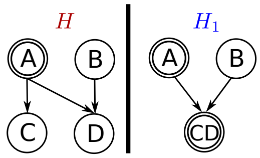


Figure 3. A heap H and its canonical decomposition H_1

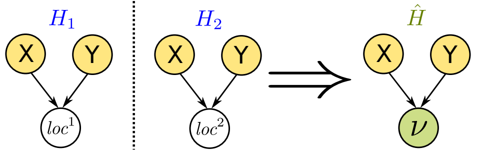


Figure 4. Two isomorphic canonical heaps and their skeleton

EXAMPLE 2. *Figure 3 shows another heap H with summary node A (indicated by double circles) and its canonical decomposition H_1. Heap H_1 is obtained from H by collapsing locations C and D into a summary location CD. Observe that we cannot split H into two heaps H_1 and H_2 where A points to C in H_1 and to D in H_2: Such a decomposition would incorrectly state that all elements in A must point to the same location, whereas H allows distinct concrete elements in A to point to distinct locations.*

COROLLARY 1. *If H has no summary nodes with multiple edges, then its canonical decomposition is exact, i.e., $\bigsqcup_{1 \le i \le k} H_i = H$.*

PROOF 2. *This follows immediately from the proof of Lemma 1.* □

LEMMA 2. *Consider a sound pointer analysis "a" and a heap H with canonical decomposition $H_1, \ldots, H_k$ such that:*

$$H_1 \vdash_a S : H'_1 \quad \ldots \quad H_k \vdash_a S : H'_k$$

Then, $H \models S : H'_1 \sqcup \ldots \sqcup H'_k$.

PROOF 3. *This follows directly from Lemma 1.* □

According to this lemma, we can conservatively analyze a program fragment S by first decomposing a heap H into canonical heaps $H_1, \ldots, H_k$, then analyzing S using each initial heap H_i, and finally combining the resulting heaps $H'_1, \ldots, H'_k$.

Recall that in a modular heap analysis, we require each node in a function f's initial heap abstraction to have the single-target property. Corollary 1 implies that if a call site of f has no summary nodes with multiple targets, then this assumption results in no loss of information, because we can use multiple distinct heaps for f that, in combination, are an exact representation of the call site's heap. However, if a summary location has multiple targets and there is aliasing involving that summary node, as illustrated in Figure 3, the modular analysis may strictly overapproximate the heap after a call to f. In this case, the requirement that f's initial heap have the single-target property means that f can only represent the call-site's heap (shown on the left of Figure 3) by an overapproximating heap that merges the target nodes (shown on the right of Figure 3).

2.3 Structural Decomposition

Consider the result of analyzing a program fragment S starting with initial canonical heaps H_1 and H_2 shown in Figure 4. Here, nodes labeled X and Y represent memory locations of x and y, which are the only variables in scope in S. Since the only difference between H_1 and H_2 is the label of the node pointed to by x and y, the heaps H'_1 and H'_2 obtained after analyzing S will be identical except

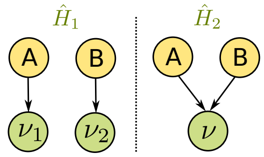


Figure 5. Structural decomposition of heap H from Figure 2

for the label of a single node. Thus, S can be analyzed only once starting with heap $\hat{H}$ in Figure 4, and H_1' and H_2' can be obtained from the resulting heap by renaming ν to *loc*^{1} and *loc*^{2} respectively. The rest of this section makes this discussion precise.

DEFINITION 4. **(Skeleton)** *Given a set of nodes N, let $\xi_N(H)$ be the heap obtained by erasing the labels of all nodes in H except for those in N. Now ξ_N defines an equivalence relation $H \equiv_N H'$ if $\xi_N(H) = \xi_N(H')$. We select one heap in each equivalence class of $\equiv_N$ as the class' unique* skeleton.

Note that nodes of skeletons are labeled—label erasure is only used to determine equivalence class membership.

EXAMPLE 3. *In Figure 4, H_1 and H_2 have the same skeleton $\hat{H}$.*

In other words, if heaps $H_1, \ldots, H_k$ have the same aliasing patterns with respect to a set of root locations N, then $\hat{H}$ is a unique points-to graph which represents their common aliasing structure. Skeletons are useful because, if N represents formals and globals in a function f, all possible aliasing patterns at call sites of f can be expressed using a finite number of skeletons.

DEFINITION 5. **(Π)** *Let H be a heap and let $\hat{H}$ be its skeleton w.r.t. nodes N. The mapping $\Pi_{H,\hat{H}}$ maps every node label in $\hat{H}$ to the label of the corresponding node in H and any other node to itself.*

DEFINITION 6. **(Structural Decomposition)** *Given heap H and nodes N, the* structural decomposition *of H w.r.t. N is a set of heaps D such that for every H_i in the canonical decomposition of H, the sketelon $\hat{H}$ of H_i w.r.t. N is in D.*

Observe that the cardinality of the structural decomposition of H is never larger than the cardinality of H's canonical decomposition.

DEFINITION 7. **(Instances of skeleton)** *Let $\hat{H}$ be a skeleton in the structural decomposition of H. The* instances *of $\hat{H}$, written $\mathcal{I}_H(\hat{H})$, are the canonical heaps of H with skeleton $\hat{H}$.*

EXAMPLE 4. *Consider heap H from Figure 2 and the root set $\{A, B\}$. The structural decomposition $\hat{H}_1, \hat{H}_2$ of H is shown in Figure 5. Observe that canonical heaps H_1 and H_4 from Figure 2 have the same skeleton $\hat{H}_1$, and H_2 and H_3 have skeleton $\hat{H}_2$. Thus, $\mathcal{I}_H(\hat{H}_1) = \{H_1, H_4\}$ and $\mathcal{I}_H(\hat{H}_2) = \{H_2, H_3\}$. Also:*

$$\begin{array}{ll} \Pi_{H_1,\hat{H}_1} = [\nu_1 \mapsto C, \nu_2 \mapsto D] & \Pi_{H_4,\hat{H}_1} = [\nu_1 \mapsto D, \nu_2, \mapsto C] \\ \Pi_{H_2,\hat{H}_2} = [\nu \mapsto D] & \Pi_{H_3,\hat{H}_2} = [\nu \mapsto C] \end{array}$$

LEMMA 3. *Consider program fragment S and nodes N representing variables in scope at S. Let H_N be the heap fragment reachable through N before analyzing S and let $\hat{H}_1, \ldots, \hat{H}_m$ be the structural decomposition of H_N w.r.t. N. If*

$$\hat{H}_1 \vdash_a S : \hat{H}_1' \quad \ldots \quad \hat{H}_m \vdash_a S : \hat{H}_m'$$

and if $\hat{H}_N'$ is the heap defined as:

$$\hat{H}_N' = \bigsqcup_{1 \le i \le m} \left(\bigsqcup_{H_{ij} \in \mathcal{I}_{H_N}(\hat{H}_i)} \Pi_{H_{ij},\hat{H}_i}(\hat{H}_i') \right)$$

then $H_N \models S : \hat{H}_N'$.

PROOF 4. *First, by Definitions 4 and 2, we have:*

$$\bigsqcup_{1 \le i \le m} \left(\bigsqcup_{H_{ij} \in \mathcal{I}_{H_N}(\hat{H}_i)} H_{ij} \right) \sqsupseteq H_N$$

Second, using Lemma 2, this implies:

$$H_N \models S : \bigsqcup_{1 \le i \le m} \left(\bigsqcup_{H_{ij} \in \mathcal{I}_{H_N}(\hat{H}_i)} H_{ij}' \right) \quad (*)$$

where $H_{ij} \vdash_a S : H_{ij}'$. From Definition 7, since H_{ij} and $\hat{H}_i$ are equivalent up to renaming, then H_{ij}' and $\hat{H}_i'$ are also equivalent up to this renaming, given by $\Pi_{H_{ij},\hat{H}_i}$. Together with (*), this implies $H_N \models S : \bigsqcup_{1 \le i \le m} \left(\bigsqcup_{H_{ij} \in \mathcal{I}_{H_N}(\hat{H}_i)} \Pi_{H_{ij},\hat{H}_i}(\hat{H}_i') \right)$. □

In other words, the heap defined as $\hat{H}_N'$ in Lemma 3 gives us a sound abstraction of the heap after analyzing program fragment S. Furthermore, $\hat{H}_N'$ is precise in the sense defined below:

LEMMA 4. *Let $\hat{H}_N'$ be the heap defined in Lemma 3, and let $H_1, \ldots, H_k$ be the canonical decomposition of the heap fragment reachable from N before analyzing S. If $H_j \vdash_a H_j'$, then:*

$$\hat{H}_N' = \bigsqcup_{1 \le j \le k} H_j'$$

PROOF 5. *This follows from Corollary 1 and Definition 6.* □

The following corollary states a stronger precision result:

COROLLARY 2. *Let H_N and $\hat{H}_N'$ be the heap abstractions from Lemma 3, and let $H_N \vdash_a S : H_N'$. If H_N does not contain summary locations with multiple points-to targets, then*

$$\hat{H}_N' \sqsubseteq H_N'$$

PROOF 6. *This follows from Lemma 4 and Corollary 1.*

2.4 From Decompositions to Modular Heap Analysis

We now show how the ideas described so far yield a basic modular heap analysis. In the remainder of this section, we assume there is a fixed bound on the total number of memory locations used by a program analysis. (In practice, this is achieved by, e.g., collapsing recursive fields of data structures to a single summary location.)

LEMMA 5. *Consider a function f, and let N denote the abstract memory locations associated with the formals and globals of f. Then, there is a finite set Q of skeletons such that the structural decomposition D w.r.t. N of the heap fragment reachable from N in any of f's call sites satisfies $D \subseteq Q$.*

PROOF 7. *Recall that in any canonical heap, every location has exactly one target. Second, observe that when there is bound b on the total number of locations in any heap, any canonical heap must have at most b locations. Thus, using a fixed set of nodes, we can only construct a finite set Q of structurally distinct graphs.* □

Since there are a bounded number of skeletons that can arise in any context, this suggests the following strategy for computing a complete summary of function f: Let N be the set of root locations (i.e., formals and globals) on entry to f, and let $\hat{H}_1, \ldots, \hat{H}_k$ be the set of all skeletons that can be constructed from root set N. We analyze f's body for each initial skeleton $\hat{H}_i$, obtaining a new heap $\hat{H}_i'$. Now, let C be a call site of f and let R be the subset of the skeletons $\hat{H}_1, \ldots, \hat{H}_k$ that occur in the structural decomposition of heap H in context C. Then, following Lemma 3, the heap fragment after the call to f can be obtained as:

$$\bigsqcup_{\hat{H}_i \in R} \left(\bigsqcup_{H_{ij} \in \mathcal{I}_H(\hat{H}_i))} \Pi_{H_{ij}, \hat{H}_i} (\hat{H}_i') \right)$$

This strategy yields a fully context-sensitive analysis because f's body is analyzed for any possible entry aliasing pattern $\hat{H}_i$, and at a given call site C, we only use the resulting heap $\hat{H}_i$ if $\hat{H}_i$ is part of the structural decomposition of the heap at C.

Furthermore, as indicated by Corollary 2, this strategy is as precise as analyzing the inlined body of the function if there are no summary locations with multiple points-to targets at this call site; otherwise, the precision guarantee is stated in Lemma 4.

2.5 Discussion

While the decompositions described here are useful for understanding the principle underlying modular heap analyses, the naive algorithm sketched in Section 2.4 is completely impractical for two reasons: First, since the number of skeletons may be exponential in the number of abstract locations reachable through arguments, such an algorithm requires analyzing a function body exponentially many times. Second, although two initial skeletons may be different, the resulting heaps after analyzing the function body may still be identical. In the rest of this paper, we describe a symbolic encoding of the basic algorithm that does not analyze a function more than once unless cycles are present in the callgraph (see Section 4). Then, in Section 5, we show how to identify only those initial skeletons that may affect the heap abstraction after the function call.

3. Language

To formalize our symbolic algorithm for modular heap analysis, we use the following typed, call-by-value imperative language:

$$\begin{array}{ll}
\textit{Program } P & := F^+ \\
\textit{Function } F & := \textit{define } f(a_1 : \tau_1, \ldots, a_n : \tau_n) = S; \\
\textit{Statement } S & := v_1 \leftarrow *v_2 \mid *\, v_1 \leftarrow v_2 \mid v \leftarrow \textit{alloc}^{\rho}(\tau) \\
 & \mid f^{\rho}(v_1, \ldots, v_k) \mid \textit{assert}(v_1 = v_2) \\
 & \mid \textit{let}^{\rho}\ v : \tau \textit{ in } S \textit{ end} \mid S_1; S_2 \mid \textit{choose}(S_1, S_2) \\
\textit{Type } \tau & := \textit{ int} \mid \textit{ptr}(\tau)
\end{array}$$

A program P consists of one or more (possibly recursive) functions F. Statements in this language are loads, stores, memory allocations, function calls, assertions, let bindings, sequencing, and the *choose* statement, which non-deterministically executes either S_1 or S_2 (i.e., a simplified conditional). Allocations, function calls, and let bindings are labeled with globally unique program points ρ.

Since this language is standard, we omit its operational semantics and highlight a few important assumptions: Execution starts at the first function defined, and an assertion failure aborts execution. Also, all bindings in the concrete store have initial value *nil*.

4. Modular & Symbolic Heap Analysis

In this section, we formally describe our symbolic algorithm for modular heap analysis. In Section 4.1, we first decribe the abstract domain used in our analysis. Section 4.2 formally defines *function summaries*, and Section 4.3 presents a full algorithm for summary-based heap analysis for the language defined in Section 3.

4.1 Abstract Domain

Abstract locations π represent a set of concrete locations:

$$\begin{array}{lcl}
\textit{Abstract Locations } \pi & := & \alpha \mid l \\
\textit{Location Variables } \alpha & := & \nu_i \mid *\, \alpha \\
\textit{Location Constants } l & := & \textit{loc}^{\vec{\rho}} \mid \textit{nil}
\end{array}$$

An abstract location π in function f is either a *location variable* α or a *location constant* l. Location constants represent stack or heap allocations in f and its transitive callees as well as *nil*. In contrast, location variables represent the unknown memory locations reachable from f's arguments at call sites, similar to access paths in [5]. Informally, location variables correspond to the node labels of a skeleton from Section 2.3. Recall from Section 2 that, in any canonical points-to graph, every abstract memory location points to at most one other abstract memory location; hence, location variable $*\nu_i$ describes the unknown, but unique, points-to target of f's i'th argument in some canonical heap at a call site of f.

Abstract environment $\mathbb{E}$ maps program variables v to abstract locations π, and abstract store $\mathbb{S}$ maps each abstract location π to an *abstract value set* θ of (abstract location, constraint) pairs:

$$\textit{Abstract value set } \theta := 2^{(\pi, \phi)}$$

The abstract store defines the edges of the points-to graph from Section 2. A mapping from abstract location π to abstract value set $\{(\pi_1, \phi_1), \ldots, (\pi_k, \phi_k)\}$ in $\mathbb{S}$ indicates that the heap abstraction contains a points-to edge from node labeled π to nodes labeled $\pi_1, \ldots, \pi_k$. Observe that, unlike the simple may points-to graph we considered in Section 2, points-to edges in the abstract store are qualified by constraints, which we utilize to symbolically encode all possible skeletons in one symbolic heap (see Section 4.3).

Constraints in our abstract domain are defined as follows:

$$\begin{array}{lcl}
\phi & := & \langle \varphi_{\textit{may}}, \varphi_{\textit{must}} \rangle \\
\varphi & := & T \mid F \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \neg\varphi \mid t_1 = t_2
\end{array}$$

Here, ϕ is a *bracketing constraint* $\langle \varphi_{\textit{may}}, \varphi_{\textit{must}} \rangle$ as in [4], representing the condition under which a property may and must hold. Recall from Section 1 that the simultaneous use of may and must information is necessary for applying strong updates whenever safe. In particular, updates to heap locations require negation (see Section 4.3). Since the negation of an overapproximation is an underapproximation, the use of bracketing constraints allows a sound negation operation, defined as $\neg\langle \varphi_{\textit{may}}, \varphi_{\textit{must}} \rangle = \langle \neg\varphi_{\textit{must}}, \neg\varphi_{\textit{may}} \rangle$. Conjunction and disjunction are defined on these constraints as expected:

$$\langle \varphi_{\textit{may}}, \varphi_{\textit{must}} \rangle \star \langle \varphi'_{\textit{may}}, \varphi'_{\textit{must}} \rangle = \langle \varphi_{\textit{may}} \star \varphi'_{\textit{may}}, \varphi_{\textit{must}} \star \varphi'_{\textit{must}} \rangle$$

where $\star \in \{\wedge, \vee\}$. In this paper, any constraint ϕ is a bracketing constraint unless stated otherwise. To make this clear, any time we do not use a bracketing constraint, we use the letter φ instead of ϕ. Furthermore, if the may and must conditions of a bracketing constraint are the same, we write a single constraint instead of a pair. Finally, for a bracketing constraint $\phi = \langle \varphi_{\textit{may}}, \varphi_{\textit{must}} \rangle$, we define $\lceil \phi \rceil = \varphi_{\textit{may}}$ and $\lfloor \phi \rfloor = \varphi_{\textit{must}}$.

In the definition of constraint φ, T and F represent the boolean constants true and false, and a term t is defined as:

$$\textit{Term } t := v \mid \textit{drf}(t) \mid \textit{alloc}(\vec{\rho}) \mid \textit{nil}$$

Here, v represents a variable, *drf* is an uninterpreted function, and *alloc* is an *invertible* uninterpreted function applied to a vector of constants $\vec{\rho}$. Thus, constraints φ belong to the theory of equality with uninterpreted functions. Our analysis requires converting between abstract locations and terms in the constraint language; we therefore define a *lift* operation, written $\overline{\pi}$, for this purpose:

$$\overline{\nu_i} = \nu_i \quad \overline{*\alpha} = \textit{drf}(\overline{\alpha}) \quad \overline{\textit{nil}} = \textit{nil} \quad \overline{\textit{loc}^{\vec{\rho}}} = \textit{alloc}(\vec{\rho})$$

Observe that a location constant $\textit{loc}^{\rho}$ is converted to a term $\textit{alloc}(\vec{\rho})$, which effectively behaves as a constant in the constraint language: Since *alloc* is an invertible function, $\textit{alloc}(\vec{\rho}) = \textit{alloc}(\vec{\rho'})$ exactly when $\rho = \rho'$. A location variable ν is converted to a constraint variable of the same name, and the location variable $*\nu$ is represented by the term $\textit{drf}(\nu)$ which represents the unknown points-to target

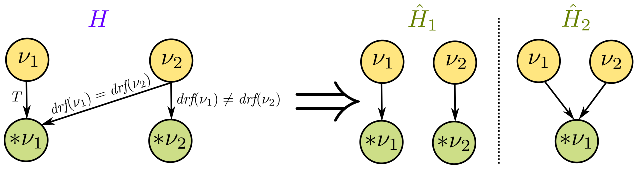


Figure 6. A symbolic heap representing two skeletons

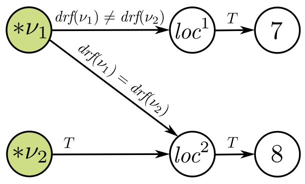


Figure 7. The abstract store in f's summary

of ν on function entry. We write $lift^{-1}(t)$ to denote the conversion of a term to an abstract location.

EXAMPLE 5. *In Figure 6, a symbolic heap H represents two skeletons $\hat{H}_1$ and $\hat{H}_2$. In H, the constraint* $\mathrm{drf}(\nu_1) = \mathrm{drf}(\nu_2)$ *describes contexts where the first and second arguments are aliases. Observe that, at call sites where the first and second arguments alias,* $\mathrm{drf}(\nu_1) = \mathrm{drf}(\nu_2)$ *instantiates to* true *and* $\mathrm{drf}(\nu_1) \neq \mathrm{drf}(\nu_2)$ *is* false*; thus at this call site, H instantiates exactly to $\hat{H}_2$. Similarly, if the first and second arguments do not alias, H instantiates to $\hat{H}_1$.*

4.2 Function Summaries

A *summary* Δ for a function f is a pair $\Delta = \langle \phi, \mathbb{S} \rangle$ where ϕ is a constraint describing the precondition for f to succeed (i.e., not abort), and $\mathbb{S}$ is a symbolic heap representing the heap abstraction after f returns. More specifically, let $\hat{H}_1, \ldots, \hat{H}_k$ be the set of all skeletons for any possible call site of f, and let $\hat{H}_i \vdash S : \hat{H}'_i$ where S is the body of f. Then, the abstract store $\mathbb{S}$ symbolically encodes that in contexts where the constraints in $\mathbb{S}$ are satisfied by initial heap $\hat{H}_i$, the resulting heap is $\hat{H}'_i$.

Observe that a summary can also be viewed as the Hoare triple $\{\phi\} f \{\mathbb{S}\}$. Thus, the computation of a summary for f is equivalent to the inference of sound pre- and post-conditions for f.

EXAMPLE 6. *Consider the function:*

```
define f(a1 : ptr(ptr(int)), a2 : ptr(ptr(int))) =
1 :  *a1 ← alloc^1(int);
2 :  *a2 ← alloc^2(int);
3 :  let t1 : ptr(int) in t1 ← *a1; *t1 ← 7 end;
4 :  let t2 : ptr(int) in t2 ← *a2; *t2 ← 8 end;
5 :  let t3 : ptr(int) in t3 ← *a1;
6 :      let t4 : int in t4 ← *t3; assert(t4 == 7) end;
7 :  end
```

The summary for f *is* $\langle\ \mathrm{drf}(\nu_1) \neq \mathrm{drf}(\nu_2),\ \mathbb{S}\ \rangle$ *where $\mathbb{S}$ is shown in Figure 7. The pre-condition* $\mathrm{drf}(\nu_1) \neq \mathrm{drf}(\nu_2)$ *indicates that the assertion fails in those contexts where arguments of f are aliases. Also, in symbolic heap $\mathbb{S}$, the abstract location reached by dereferencing a_1 (whose location is ν_1) is either* loc_1*, corresponding to the allocation at line 1, or* loc_2*, associated with the allocation at line 2, depending on whether a_1 and a_2 are aliases.*

A *global summary environment* $\mathbb{G}$ is a mapping from each function f in the program to a summary Δ_f.

$$\frac{\begin{array}{l}\mathbb{E} = [a_1 \leftarrow \nu_1, \ldots, a_k \leftarrow \nu_k] \\ \forall\ \alpha_i \in dom(\mathbb{A}). \\ \quad \mathbb{S}(\alpha_i) \leftarrow \cup_{k \leq i}(*\alpha_k, (\bigwedge_{j<k} \overline{*\alpha_i} \neq \overline{*\alpha_j}) \wedge \overline{*\alpha_i} = \overline{*\alpha_k})\end{array}}{\mathbb{A} \vdash \mathit{init\_heap}(a_1, \ldots, a_k) : \mathbb{E}, \mathbb{S}}$$

Figure 8. Local Heap Initialization

4.3 The Analysis

We now present the full algorithm for the language of Section 3. Section 4.3.1 describes the symbolic initialization of the local heap to account for all possible aliasing relations on function entry. Section 4.3.2 gives abstract transformers for all statements except function calls, which is described in Section 4.3.3. Finally, Section 4.3.4 describes the generation of function summaries.

4.3.1 Local Heap Initialization

To analyze a function f independent of its callers, we initialize f's abstract store to account for all possible relevant aliasing relationships at function entry. To perform this local heap initialization, we utilize an *alias partition environment* $\mathbb{A}$ with the signature $\alpha \rightarrow 2^{\alpha}$. This environment maps each location variable α to an *ordered* set of location variables, called α's *alias partition set*. If $\alpha' \in \mathbb{A}(\alpha)$, then f's summary may differ in contexts where α and α' are aliases. Since aliasing is a symmetric property, any alias partition environment $\mathbb{A}$ has the property $\alpha' \in \mathbb{A}(\alpha) \Leftrightarrow \alpha \in \mathbb{A}(\alpha')$. Any location aliases itself, and so $\mathbb{A}$ is also reflexive: $\alpha \in \mathbb{A}(\alpha)$. A correct alias partition environment $\mathbb{A}$ can be trivially computed by stipulating that $\alpha' \in \mathbb{A}(\alpha)$ if α and α' have the same type. We discuss how to compute a more precise alias partition environment $\mathbb{A}$ in Section 5.

A key component of the modular analysis is the *init_heap* rule in Figure 8. Given formal parameters $a_1, \ldots, a_k$ to function f, this rule initializes the abstract environment and store on entry to f. The environment $\mathbb{E}$ is initialized by binding a location variable ν_i to each argument a_i. The initialization of the abstract store $\mathbb{S}$, however, is more involved because we need to account for all possible entry aliasing relationships permitted by $\mathbb{A}$.

Intuitively, if $\mathbb{A}$ indicates that α_1 and α_2 may alias on function entry, we need to analyze f's body with two skeletal heaps, one where α_1 and α_2 point to the same location, and one where α_1 and α_2 point to different locations. To encode this symbolically, one obvious solution is to introduce three location variables, $*\alpha_1$, $*\alpha_2$, and $*\alpha_{12}$ such that α_1 (resp. α_2) points to $*\alpha_1$ (resp. $*\alpha_2$) if they do not alias (i.e., under constraint $drf(\alpha_1) \neq drf(\alpha_2)$) and point to a common location named $*\alpha_{12}$ if they alias (i.e., under $drf(\alpha_1) = drf(\alpha_2)$). While this encoding correctly describes both skeletal heaps, it unfortunately introduces an exponential number of locations, one for each subset of entry alias relations in $\mathbb{A}$.

To avoid this exponential blow-up, we impose a total order on abstract locations such that if α_i and α_j are aliases, they both point to a common location $*\alpha_k$ such that α_k is the least element in the alias partition class of α_i and α_j. Thus, in the *init_heap* rule of Figure 8, α_i points to $*\alpha_k$ where $k \leq i$ under constraint:

$$(\bigwedge_{j<k} \overline{*\alpha_i} \neq \overline{*\alpha_j}) \quad \wedge \quad \overline{*\alpha_i} = \overline{*\alpha_k}$$

This condition ensures that α_i points to a location named $*\alpha_k$ only if it does not alias any other location $\alpha_j \in \mathbb{A}(\alpha_i)$ with $j < k$.

EXAMPLE 7. *Consider the function defined in Example 6. Suppose the alias partition environment $\mathbb{A}$ contains the following mappings:*

$$\nu_1 \mapsto \{\nu_1, \nu_2\}, \nu_2 \mapsto \{\nu_1, \nu_2\}, *\nu_1 \mapsto \{*\nu_1\}, *\nu_2 \mapsto \{*\nu_2\}, \\ **\,\nu_1 \mapsto \{**\,\nu_1\}, **\,\nu_2 \mapsto \{**\,\nu_2\}$$

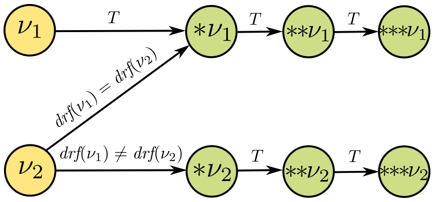


Figure 9. The initial heap abstraction for function from Example 6

Figure 9 shows the initial heap abstraction using $\mathbb{A}$ and the ordering $\nu_1 < \nu_2$. Since $\mathbb{A}$ includes ν_1 and ν_2 in the same alias partition set, ν_2 points to $*\nu_1$ under $\mathrm{drf}(\nu_1) = \mathrm{drf}(\nu_2)$ and to $*\nu_2$ under its negation. But ν_1 only points to $*\nu_1$ since $\nu_2 \not< \nu_1$.

The following lemma states that the initial heap abstraction correctly accounts for all entry aliasing relations permitted by $\mathbb{A}$:

LEMMA 6. *Let α_i and α_j be two abstract locations such that $\alpha_j \in \mathbb{A}(\alpha_i)$. The initial local heap abstraction $\mathbb{S}$ constructed in Figure 8 encodes that α_i and α_j point to distinct locations* exactly *in those contexts where they do not alias.*

PROOF 8. *Without loss of generality, assume $i < j$.*
$\Rightarrow$ Suppose α_i and α_j are not aliases in a context C, but $\mathbb{S}$ encodes they may point to the same location $*\alpha_k$ in context C. Let ϕ and ϕ' be the constraints under which α_i and α_j point to α_k respectively. By construction, $k \leq i$, and ϕ implies $\mathrm{drf}(\alpha_i) = \mathrm{drf}(\alpha_k)$ and ϕ' implies $\mathrm{drf}(\alpha_j) = \mathrm{drf}(\alpha_k)$. Thus, we have $\mathrm{drf}(\alpha_i) = \mathrm{drf}(\alpha_j)$, contradicting the fact that α_i and α_j do not alias in C.
$\Leftarrow$ Suppose α_i and α_j are aliases in context C, but $\mathbb{S}$ allows α_i and α_j to point to distinct locations $*\alpha_k$ and $*\alpha_m$. Let ϕ and ϕ' be the constraints under which α_i points to $*\alpha_k$ and α_j points to $*\alpha_m$ respectively. Case (i): Suppose $k < m$. Then, by construction, ϕ implies $\mathrm{drf}(\alpha_i) = \mathrm{drf}(\alpha_k)$, and ϕ' implies $\mathrm{drf}(\alpha_j) \neq \mathrm{drf}(\alpha_k)$. Hence, we have $\mathrm{drf}(\alpha_j) \neq \mathrm{drf}(\alpha_i)$, contradicting the assumption that α_i and α_j are aliases in C. Case (ii): $k > m$. Then, ϕ' implies $\mathrm{drf}(\alpha_j) = \mathrm{drf}(\alpha_m)$, and ϕ implies $\mathrm{drf}(\alpha_i) \neq \mathrm{drf}(\alpha_m)$, again contradicting the fact that α_i and α_j are aliases in C. □

LEMMA 7. *For each alias partition set of size n, the init_heap rule adds $n(n+1)/2$ points-to edges.*

As Lemma 7 states, this construction introduces a quadratic number of edges in the size of each alias partition set to represent all possible skeletal heaps. Furthermore, the number of abstract locations in the initial symbolic heap is no larger than the maximum number of abstract locations in any individual skeleton.

4.3.2 Abstract Transformers for Basic Statements

In this section, we describe the abstract transformers for all statements except function calls, which is the topic of Section 4.3.3. Statement transformers are given as inference rules of the form

$$\mathbb{E}, \mathbb{S}, \mathbb{G}, \phi \vdash S : \mathbb{S}', \phi'$$

which states that under abstract environment $\mathbb{E}$, store $\mathbb{S}$, summary environment $\mathbb{G}$, and precondition ϕ, statement S produces a new abstract store $\mathbb{S}'$ and a new precondition ϕ' of the current function. The operation $\mathbb{S}(\theta)$ looks up the value of each π_i in θ:

$$S(\{(\pi_1, \phi_1), \ldots, (\pi_k, \phi_k)\}) = \bigcup_{1 \leq i \leq k} \mathbb{S}(\pi_i) \wedge \phi_i$$

where $\mathbb{S}(\pi_i) \wedge \phi_i$ is a shorthand defined as follows:

$$\theta \wedge \phi = \{(\pi_j, \phi_j \wedge \phi) | (\pi_j, \phi_j) \in \theta\}$$

$$(1) \qquad \frac{\begin{array}{c}\mathbb{E}(v_1) = \pi_1 \;\; \mathbb{E}(v_2) = \pi_2 \\ \mathbb{S}(\pi_2) = \theta \;\; \mathbb{S}' = \mathbb{S}[\pi_1 \leftarrow \mathbb{S}(\theta)]\end{array}}{\mathbb{E}, \mathbb{S}, \mathbb{G}, \phi \vdash v_1 \leftarrow *v_2 : \mathbb{S}', \phi}$$

$$(2) \qquad \frac{\begin{array}{c}\mathbb{E}(v_1) = \pi_1 \;\; \mathbb{E}(v_2) = \pi_2 \\ \mathbb{S}(\pi_1) = \theta_1 \;\; \mathbb{S}(\pi_2) = \theta_2 \\ \mathbb{S}' = \mathbb{S}[\pi_i \leftarrow ((\theta_2 \wedge \phi_i) \cup (\mathbb{S}(\pi_i) \wedge \neg\phi_i)) \mid (\pi_i, \phi_i) \in \theta_1]\end{array}}{\mathbb{E}, \mathbb{S}, \mathbb{G}, \phi \vdash *v_1 \leftarrow v_2 : \mathbb{S}', \phi}$$

$$(3) \qquad \frac{\begin{array}{c}\mathbb{E}(v) = \pi \\ \mathbb{S}' = \mathbb{S}[\pi \leftarrow \{(\mathit{loc}^\rho, T)\},\ \mathit{loc}^\rho \leftarrow \{(\mathit{nil}, T)\}]\end{array}}{\mathbb{E}, \mathbb{S}, \mathbb{G}, \phi \vdash v \leftarrow \mathit{alloc}^\rho(\tau) : \mathbb{S}', \phi}$$

$$(4) \qquad \frac{\begin{array}{c}\mathbb{E}(v_1) = \pi \;\; \mathbb{E}(v_2) = \pi' \\ \mathbb{S}(\pi) = \{\ldots, (\pi_i, \phi_i), \ldots\} \\ \mathbb{S}(\pi') = \{\ldots, (\pi'_j, \phi'_j), \ldots\} \\ \phi' = \bigvee_{i,j}(\pi_i = \pi'_j \wedge \phi_i \wedge \phi'_j)\end{array}}{\mathbb{E}, \mathbb{S}, \mathbb{G}, \phi \vdash \mathit{assert}(v_1 = v_2) : \mathbb{S}, \phi \wedge \phi'}$$

$$(5) \qquad \frac{\begin{array}{c}\mathbb{E}' = \mathbb{E}[v \leftarrow \mathit{loc}^\rho] \\ \mathbb{S}' = \mathbb{S}[*\mathit{loc}^\rho \leftarrow \{(\mathit{nil}, T)\}] \\ \mathbb{E}', \mathbb{S}', \mathbb{G}, \phi \vdash S : \mathbb{S}'', \phi'\end{array}}{\mathbb{E}, \mathbb{S}, \mathbb{G}, \phi \vdash \mathit{let}^\rho\ v : \tau\ \mathit{in}\ S\ \mathit{end}\ : \mathbb{S}'' \backslash \mathit{loc}^\rho, \phi'}$$

$$(6) \qquad \frac{\begin{array}{c}\mathbb{E}, \mathbb{S}, \mathbb{G}, \phi \vdash S_1 : \mathbb{S}', \phi' \\ \mathbb{E}, \mathbb{S}', \mathbb{G}, \phi' \vdash S_2 : \mathbb{S}'', \phi''\end{array}}{\mathbb{E}, \mathbb{S}, \mathbb{G}, \phi \vdash S_1; S_2 : \mathbb{S}'', \phi''}$$

$$(7) \qquad \frac{\begin{array}{c}\mathbb{E}, \mathbb{S}, \mathbb{G}, \phi \vdash S_1 : \mathbb{S}_1, \phi_1 \\ \mathbb{E}, \mathbb{S}, \mathbb{G}, \phi \vdash S_2 : \mathbb{S}_2, \phi_2\end{array}}{\mathbb{E}, \mathbb{S}, \mathbb{G}, \phi \vdash\ \mathit{choose}\ (S_1, S_2) : \mathbb{S}_1 \sqcup \mathbb{S}_2, \phi_1 \wedge \phi_2}$$

Figure 10. Abstract Transformers for Basic Statements

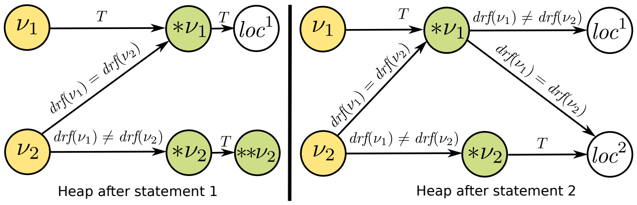

ν_1
T
$*\nu_1$
T
loc^1
$drf(\nu_1) = drf(\nu_2)$
ν_2
$drf(\nu_1) \neq drf(\nu_2)$
$*\nu_2$
T
$**\nu_2$
Heap after statement 1
ν_1
T
$*\nu_1$
$drf(\nu_1) \neq drf(\nu_2)$
loc^1
$drf(\nu_1) = drf(\nu_2)$
$drf(\nu_1) = drf(\nu_2)$
ν_2
$drf(\nu_1) \neq drf(\nu_2)$
$*\nu_2$
T
loc^2
Heap after statement 2

Figure 11. The symbolic heap before and after line 2 in Example 6

In Figure 10, rules (1) and (2) give the transformers for loads and stores respectively. The rule for loads is self-explanatory; thus, we focus on the store rule. In the third hypothesis of rule (2), each π_i represents a location that v_1 points to under constraint ϕ_i, and θ_2 is the value set for v_2. Since the write to π_i happens under constraint ϕ_i, the new value of π_i in $\mathbb{S}'$ is θ_2 under constraint ϕ_i and retains its old value set $\mathbb{S}(\pi_i)$ under $\neg\phi_i$. Observe that if ϕ_i is *true*, this rule performs a standard strong update to π_i. On the other hand, if v_1 points to π_i under some entry alias assumption, then there is a strong update to π_i exactly in those calling contexts where this alias assumption holds.

EXAMPLE 8. *Figure 11 shows the relevant portion of the heap abstraction before and after the store at line 2 in Example 6.*

Rule (3) processes allocations by introducing a new location loc^ρ and initializing its value in the store to *nil*. Rule (4) analyzes an assertion by computing the condition ϕ' for the assertion to hold such that if ϕ' can be proven valid in a calling context, then this assertion must hold at that call site. In rule (4), ϕ' is computed as

$$\frac{}{\mathbb{S},\mathbb{I} \vdash \mathit{map\_loc}(\nu : \mathit{int}) : \mathbb{I}} \qquad \frac{\mathbb{S},\mathbb{I} \vdash \mathit{map\_loc}(*\nu : \tau) : \mathbb{I}'}{\mathbb{S},\mathbb{I} \vdash \mathit{map\_loc}(\nu : \mathit{ptr}(\tau)) : \mathbb{I}'}$$

$$\frac{\mathbb{I}' = \mathbb{I}[*\alpha \leftarrow \mathbb{S}(\mathbb{I}(\alpha))]}{\mathbb{S},\mathbb{I} \vdash \mathit{map\_loc}(*\alpha : \mathit{int}) : \mathbb{I}'} \qquad \frac{\begin{array}{c}\mathbb{I}' = \mathbb{I}[*\alpha \leftarrow \mathbb{S}(\mathbb{I}(\alpha))] \\ \mathbb{I}' \vdash \mathit{map\_loc}(**\,\alpha : \tau) : \mathbb{I}''\end{array}}{\mathbb{S},\mathbb{I} \vdash \mathit{map\_loc}(*\alpha : \mathit{ptr}(\tau)) : \mathbb{I}''}$$

$$\frac{\begin{array}{c}\mathbb{S}, [\nu_1 \leftarrow \{(\mathbb{E}(v_1), T)\}] \vdash \mathit{map\_loc}(\nu_1) : \mathbb{I}_1 \\ \dots \\ \mathbb{S}, \mathbb{I}_{k-1}[\nu_k \leftarrow \{(\mathbb{E}(v_k), T)\}] \vdash \mathit{map\_loc}(\nu_k) : \mathbb{I}_k\end{array}}{\mathbb{E},\mathbb{S} \vdash \mathit{map\_args}(v_1 : \tau_1, \dots, v_k : \tau_k) : \mathbb{I}_k}$$

Figure 12. Rules for computing instantiation environment $\mathbb{I}$

$$\frac{}{\mathbb{I},\rho \vdash \mathit{inst\_loc}(\mathit{nil}) : \{(\mathit{nil}, T)\}} \qquad \frac{\theta = \{(\mathit{loc}^{\rho::\vec{\rho'}}, T)\}}{\mathbb{I},\rho \vdash \mathit{inst\_loc}(\mathit{loc}^{\vec{\rho'}}) : \theta}\ (\rho \notin \vec{\rho'})$$

$$\frac{}{\mathbb{I},\rho \vdash \mathit{inst\_loc}(\alpha) : \mathbb{I}(\alpha)} \qquad \frac{\theta = \{(\mathit{loc}^{\vec{\rho'}}, \langle T, F\rangle)\}}{\mathbb{I},\rho \vdash \mathit{inst\_loc}(\mathit{loc}^{\vec{\rho'}}) : \theta}\ (\rho \in \vec{\rho'})$$

Figure 13. Rules for instantiating locations

$$\frac{\begin{array}{l}\mathbb{I},\rho \vdash \mathit{inst\_loc}(\mathit{lift}^{-1}(t_1)) : \{(\pi_1,\phi_1),\dots,(\pi_k,\phi_k)\} \\ \mathbb{I},\rho \vdash \mathit{inst\_loc}(\mathit{lift}^{-1}(t_2)) : \{(\pi'_1,\phi'_1),\dots,(\pi'_m,\phi'_m)\} \\ \phi = \bigvee_i (k = \pi_i \wedge \phi_i) \quad \phi' = \bigvee_j (k' = \pi'_j \wedge \phi'_j)\ \ (k, k'\ \mathit{fresh})\end{array}}{\mathbb{I},\rho \vdash \mathit{inst}_{\varphi}(t_1 = t_2) : k = k', \phi \wedge \phi'}$$

$$\frac{b \in \{T, F\}}{\mathbb{I},\rho \vdash \mathit{inst}_{\varphi}(b) : b, T} \qquad \frac{\mathbb{I},\rho \vdash \mathit{inst}_{\varphi}(\varphi) : \varphi_1, \phi_2}{\mathbb{I},\rho \vdash \mathit{inst}_{\varphi}(\neg\varphi) : \neg\varphi_1, \phi_2}$$

$$\frac{\mathbb{I},\rho \vdash \mathit{inst}_{\varphi}(\varphi_1) : \varphi, \phi \quad \mathbb{I},\rho \vdash \mathit{inst}_{\varphi}(\varphi_2) : \varphi', \phi'}{\mathbb{I},\rho \vdash \mathit{inst}_{\varphi}(\varphi_1 \star \varphi_2) : \varphi \star \varphi', \phi \wedge \phi'}(\star \in \{\wedge, \vee\})$$

$$\frac{\begin{array}{l}\mathbb{I},\rho \vdash \mathit{inst}_{\varphi}(\varphi_{\mathit{may}}) : \varphi'_{\mathit{may}}, \phi'_{\mathit{may}} \quad \mathbb{I},\rho \vdash \mathit{inst}_{\varphi}(\varphi_{\mathit{must}}) : \varphi'_{\mathit{must}}, \phi'_{\mathit{must}} \\ \varphi''_{\mathit{may}} = \lceil \mathit{QE}(\exists\vec{k}.\ (\varphi'_{\mathit{may}} \wedge \phi'_{\mathit{may}}))\rceil \quad \varphi''_{\mathit{must}} = \lfloor \mathit{QE}(\exists\vec{k}.\ (\varphi'_{\mathit{must}} \wedge \phi'_{\mathit{must}}))\rfloor\end{array}}{\mathbb{I},\rho \vdash \mathit{inst}_{\phi}(\langle\varphi_{\mathit{may}}, \varphi_{\mathit{must}}\rangle) : \langle\phi''_{\mathit{may}}, \phi''_{\mathit{must}}\rangle}$$

Figure 14. Rules for instantiating constraints

the disjunction of all pairwise equalities of the elements in the two abstract value sets associated with v_1 and v_2, i.e., a case analysis of their possible values. Rule (5) describes the abstract semantics of let statements by binding variable v to a new location loc^{ρ} in $\mathbb{E}$. Rule (6) for sequencing is standard, and rule (7) gives the semantics of the *choose* construct, which computes the join of two abstract stores $\mathbb{S}_1$ and $\mathbb{S}_2$. To define a join operation on abstract stores, we first define *domain extension*:

DEFINITION 8. **(Domain Extension)** *Let π be any binding in abstract store $\mathbb{S}'$ and let (π_i, ϕ_i) be any element of $\mathbb{S}'(\pi)$. We say an abstract store $S'' = \mathbb{S}_{\mapsto\mathbb{S}'}$ is a domain extention of $\mathbb{S}$ with respect to $\mathbb{S}'$ if the following condition holds:*

1. *If $\pi \in \mathrm{dom}(\mathbb{S}) \wedge (\pi_i, \phi'_i) \in \mathbb{S}(\pi)$, then $(\pi_i, \phi'_i) \in \mathbb{S}_{\mapsto\mathbb{S}'}(\pi)$.*
2. *Otherwise, $(\pi_i, \text{false}) \in \mathbb{S}_{\mapsto\mathbb{S}'}(\pi)$*

DEFINITION 9. **(Join)** *Let $\mathbb{S}'_1 = \mathbb{S}_{1\mapsto\mathbb{S}_2}$ and let $\mathbb{S}'_2 = \mathbb{S}_{2\mapsto\mathbb{S}_1}$. If $(\pi', \langle\varphi^1_{\text{may}}, \varphi^1_{\text{must}}\rangle) \in \mathbb{S}'_1(\pi)$ and $(\pi', \langle\varphi^2_{\text{may}}, \varphi^2_{\text{must}}\rangle) \in \mathbb{S}'_2(\pi)$, then:*

$$(\pi', \langle\varphi^1_{\text{may}} \vee \varphi^2_{\text{may}}, \varphi^1_{\text{must}} \wedge \varphi^2_{\text{must}}\rangle) \in (\mathbb{S}_1 \sqcup \mathbb{S}_2)(\pi)$$

4.3.3 Instantiation of Summaries

The most involved statement transformer is the one for function calls, which we describe in this subsection. Figure 15 gives the complete transformer for function calls, making use of the helper rules defined in Figures 12- 14, which we discuss in order.

Given the actuals $v_1, \dots, v_k$ for a call to function f, Figure 12 computes the *instantiation environment* $\mathbb{I}$ with signature $\alpha \to \theta$ for this call site. This environment $\mathbb{I}$, which serves as the symbolic equivalent of the mapping Π from Section 2, maps location variables used in f to their corresponding locations in the current (calling) function. However, since $\mathbb{I}$ is symbolic, it produces an abstract value set $\{(\pi_1, \phi_1), \dots, (\pi_k, \phi_k)\}$ for each α such that α instantiates to π_i in some canonical heap under constraint ϕ_i.

Figure 13 describes the rules for instantiating any location π used in the summary. If π is a location variable, we use environment $\mathbb{I}$ to look up its instantiation. On the other hand, if π is a location constant allocated in callee f, we need to rename this constant to distinguish allocations made at different call sites for full context-sensitivity. In general, we rename the location constant $\mathit{loc}^{\vec{\rho'}}$ by prepending to $\vec{\rho'}$ the program point ρ associated with the call site. However, in the presence of recursion, we need to avoid creating an unbounded number of location constants; thus, in Figure 13, we check if this allocation is created on a cyclic path in the callgraph by testing whether the current program point ρ is already in $\vec{\rho'}$. In the latter case, we do not create a new location constant but weaken the bracketing constraint associated with $\mathit{loc}^{\vec{\rho'}}$ to $\langle T, F\rangle$, which has the effect of ensuring that stores into this location only apply weak updates [4], meaning that $\mathit{loc}^{\vec{\rho'}}$ behaves as a summary location.

In addition to instantiating locations, we must also instantiate the associated constraints, which is described in Figure 14. In the last rule of this figure, inst_{ϕ} instantiates a bracketing constraint, making use of inst_{φ} to map the constituent may and must conditions. The inst_{φ} rule derives judgments of the form $\mathbb{I}, \rho \vdash \mathit{inst}_{\varphi}(\varphi) : \varphi', \phi$, where φ' preserves the structure of φ by substituting each term t in φ with a temporary variable k and ϕ constrains the values of k.

The first rule in Figure 14 for instantiating a leaf $t_1 = t_2$ is the most interesting one: Here, we convert t_1 and t_2 to their corresponding memory locations using the lift^{-1} operation from Section 4.1 and instantiate the corresponding locations using $\mathit{inst\_loc}$ to obtain abstract value sets θ_1 and θ_2. We then introduce two temporary variables k and k' representing θ_1 and θ_2 respectively, and introduce constraints ϕ and ϕ', stipulating the equality between k and θ_1 and between k' and θ_2. Observe that in the last rule of Figure 14, these temporary variables k and k' are removed using a *QE* procedure to eliminate existentially quantified variables.

Figure 15 makes use of all the afore-mentioned rules to instantiate the summary of function f at a given call site ρ. In the last rule of this figure, we first look up f's summary $\langle\phi_f, \mathbb{S}_f\rangle$ in the global summary environment $\mathbb{G}$. The precondition ϕ_f is instantiated to ϕ'_f using inst_{ϕ}. Observe that if ϕ'_f is valid, then the potential assertion failure in f is discharged at this call site; otherwise, ϕ'_f is conjoined with the precondition ϕ of the current function.

Next, we compose the partial heap $\mathbb{S}_f$, representing the heap fragment reachable in f after the call, with the existing heap $\mathbb{S}$ before the function call. The *compose_partial_heap* rule used in *compose_heap* instantiates an entry $\pi \mapsto \theta$ in f's summary. Observe that if location π in f's summary instantiates to location π_i in the current function under ϕ_i, existing values of π_i are only preserved under $\neg\phi_i$. Hence, if ϕ_i is *true*, this rule applies a strong update to π_i. On the other hand, if π instantiates to π_i under some entry alias condition, then this rule represents a strong update to π_i only in those contexts where the entry aliasing condition holds.

EXAMPLE 9. *Consider a call to function f of Example 6:*

$$\mathtt{define\ g(a_1 : ptr(ptr(int))) = f^3(a_1, a_1)}$$

$$\frac{\mathbb{I}, \rho \vdash \textit{inst\_loc}(\pi_1) = \theta_1, \ldots \textit{inst\_loc}(\pi_k) = \theta_k}{\mathbb{I}, \rho \vdash \textit{inst\_theta}(\{(\pi_1, \phi_1), \ldots, (\pi_k, \phi_k)\}) : \bigcup_{1 \leq i \leq k} (\theta_i \wedge \phi_i)}$$

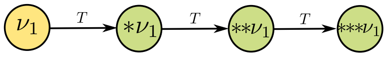
$$\frac{\begin{array}{c}\mathbb{I}, \rho \vdash \textit{inst\_loc}(\pi) = \theta_s \\ \mathbb{I}, \rho \vdash \textit{inst\_theta}(\theta) = \theta_t \\ \mathbb{S}' = \mathbb{S}[\pi_i \leftarrow (\theta_t \wedge \phi_i) \cup (\mathbb{S}(\pi_i) \wedge \neg\phi_i) \mid (\pi_i, \phi_i) \in \theta_s]\end{array}}{\mathbb{S}, \mathbb{I}, \rho \vdash \textit{compose\_partial\_heap}(\pi, \theta) : \mathbb{S}'}$$

$$\frac{\begin{array}{c}\mathbb{S}_f = [(\pi_1 \mapsto \theta_1), \ldots, (\pi_k \mapsto \theta_k)] \\ \mathbb{S}, \mathbb{I}, \rho \vdash \textit{compose\_partial\_heap}(\pi_1, \theta_1) : \mathbb{S}_1 \\ \ldots \\ \mathbb{S}_{k-1}, \mathbb{I}, \rho \vdash \textit{compose\_partial\_heap}(\pi_k, \theta_k) : \mathbb{S}_k\end{array}}{\mathbb{S}, \mathbb{I}, \rho \vdash \textit{compose\_heap}(\mathbb{S}_f) : \mathbb{S}_k}$$

$$\frac{\begin{array}{l}\mathbb{G}(f) = \langle \phi_f, \mathbb{S}_f \rangle \\ \mathbb{E}, \mathbb{S} \vdash \textit{map\_args}(v_1, \ldots, v_k) : \mathbb{I} \\ \mathbb{I}, \rho \vdash \textit{inst}_\phi(\phi_f) : \phi'_f \\ \mathbb{S}, \mathbb{I}, \rho \vdash \textit{compose\_heap}(\mathbb{S}_f) : \mathbb{S}'\end{array}}{\mathbb{E}, \mathbb{S}, \mathbb{G}, \phi \vdash f^\rho(v_1, \ldots, v_k) : \mathbb{S}', \phi \wedge \phi'_f}$$

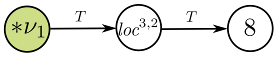
Figure 15. Summary Instantiation rules

Before the call to f, g's local heap is depicted as:



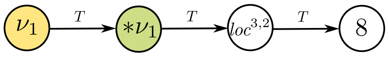
(*)

Recall from Example 6 that f's precondition is $\mathrm{drf}(\nu_1) \neq \mathrm{drf}(\nu_2)$, *which instantiates to* $\mathrm{drf}(\nu_1) \neq \mathrm{drf}(\nu_1) \Leftrightarrow$ false *at this call site, indicating that the assertion is guaranteed to fail. The store in f's summary from Figure 7 is instantiated at the call site to:*



(**)

Composing initial heap (*) with the instantiated heap (**), we obtain the final heap after the function call:



Observe that the resulting abstract heap is as precise as analyzing the inlined body of f.

4.3.4 Summary Generation and Fixed-point Computation

We now conclude this section by describing function summary generation, given in Figure 16. Before analyzing the body of f, the local abstract heap $\mathbb{S}$ is initialized as described in Section 4.3.1. Next, f's body is analyzed using the abstract transformers from Section 4.3.2 and 4.3.3, which yields a store $\mathbb{S}'$ and a precondition ϕ'. According to the last hypothesis in Figure 16, the summary $\langle \phi_f, \mathbb{S}_f \rangle$ is sound if $\mathbb{S}_f$ overapproximates $\mathbb{S} \backslash \{\nu_1, \ldots, \nu_k\}$ and ϕ_f implies ϕ'. Here, $\mathbb{S}_1 \sqsubseteq \mathbb{S}_2$ is defined as:

DEFINITION 10. *($\sqsubseteq$) Let* $\mathbb{S}'_1 = \mathbb{S}_{1 \mapsto \mathbb{S}_2}$ *and* $\mathbb{S}'_2 = \mathbb{S}_{2 \mapsto \mathbb{S}_1}$. *We say* $\mathbb{S}_1 \sqsubseteq \mathbb{S}_2$ *if for every* $\pi \in \mathrm{dom}(\mathbb{S}'_1)$ *and for every* π' *such that* $(\pi', \langle \varphi^1_{\mathrm{may}}, \varphi^1_{\mathrm{must}} \rangle) \in \mathbb{S}'_1(\pi)$, $(\pi', \langle \varphi^2_{\mathrm{may}}, \varphi^2_{\mathrm{must}} \rangle) \in \mathbb{S}'_2(\pi)$, *we have:*

$$\varphi^1_{\mathrm{may}} \Rightarrow \varphi^2_{\mathrm{may}} \wedge \varphi^2_{\mathrm{must}} \Rightarrow \varphi^1_{\mathrm{must}}$$

While the rule in Figure 16 verifies that $\langle \phi_f, \mathbb{S}_f \rangle$ is a sound summary, it does not give an algorithmic way of computing it. In the presence of recursion, we perform a least fixed-point computation where all entries in $\mathbb{G}$ are initially $\bot$ (i.e., any location points to any other location under *false*), and a new summary for f is obtained by computing the join of f's new and old summaries:

$$\langle \mathbb{S}_1, \phi_1 \rangle \sqcup \langle \mathbb{S}_2, \phi_2 \rangle = \langle \mathbb{S}_1 \sqcup \mathbb{S}_2, \phi_1 \wedge \phi_2 \rangle$$

$$\frac{\begin{array}{l}\mathbb{A} \vdash \textit{init\_heap}(a_1, \ldots, a_k) : \mathbb{E}, \mathbb{S} \\ \mathbb{E}, \mathbb{S}, \mathbb{G}, \textit{true} \vdash S : \phi', \mathbb{S}' \\ \mathbb{G} \vdash f : \langle \phi_f, \mathbb{S}_f \rangle \\ \phi_f \Rightarrow \phi' \quad \mathbb{S}_f \sqsupseteq (\mathbb{S}' \backslash \{\nu_1, \ldots, \nu_k\})\end{array}}{\mathbb{G}, \mathbb{A} \vdash \textit{define } f(a_1 : \tau_1, \ldots, a_k : \tau_k) = S : \langle \phi_f, \mathbb{S}_f \rangle}$$

Figure 16. Summary generation rule

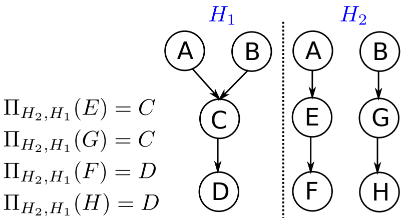


Figure 17. Heaps from Lemma 8

This strategy ensures that the analysis is monotonic by construction. Furthermore, since the analysis creates a finite number of abstract locations and the constraints are over a finite vocabulary of predicates, this fixed-point computation is guaranteed to converge. In fact, for an acyclic callgraph, each function is analyzed only once if a reverse topological order is used.

5. Computing Alias Partition Sets

In the previous section, we assumed the existence of an alias partition environment $\mathbb{A}$ that is used to query whether aliasing between locations α and α' may affect analysis results. One simple way to compute such an environment is to require that $\alpha' \in \mathbb{A}(\alpha)$ if α and α' have the same type (at least in a type-safe language). Fortunately, it is possible to compute a much more precise alias partition environment because many aliasing relations at a call site of f do not affect the state of the heap after a call to f. The following lemma elucidates when we can safely ignore potential aliasing between two locations in a code fragment S.

LEMMA 8. *Let H_1 and H_2 be the canonical heap fragments shown in Figure 17, and let S be a program fragment such that:*

- *There is either no store to A and no store to B, or*
- *There is a store to only A that is not followed by a load from B, or*
- *There are only stores to both A and B, but the store to A must happen after the store to B*

Let $H_1 \vdash S : H'_1$ and $H_2 \vdash S : H'_2$, and let O be a partial order such that $O(B) \prec O(A)$ if there must be a write to A after a write to B in S. Let H''_2 be the graph obtained by replacing G's targets with E's targets in H'_2 if $\Pi_{H_2,H_1}(A) = \Pi_{H_2,H_1}(B)$ and $O(B) \prec O(A)$. Then, $H'_1 = \Pi_{H_2,H_1}(H''_2)$.

PROOF 9. *(sketch) There are three cases: (i) If there is no store to A or B, then in H'_2, E and G still point to F and H, both of which are equivalent to D in H'_1. Thus, $H'_1 = \Pi_{H_2,H_1}(H'_2)$. (ii) There is only a store to A, not followed by a load from B: In H'_1, C will point to some set of new locations $T_1, \ldots, T_k$. In H'_2, E must also point to $T'_1, \ldots, T'_k$ such that $T_i = \Pi_{H_2,H_1}(T'_i)$ and G must point to H. First, the result of any load from B (i.e., H) can be correctly renamed to D, as the read happens before the store to A. Second,*

	LiteSQL	**OpenSSH**	**Inkscape widget lib.**	**Digikam**
Lines	16,030	22,615	37,211	128,318
Strong updates at instantiation				
Running time (1 CPU)	4.5 min	3.9 min	7.2 min	45.1 min
Running time (8 CPUs)	1.6 min	1.8 min	2.3 min	8.7 min
Memory use	430 MB	230 MB	195 MB	400 MB
Error reports	7	6	7	37
False positives	2	1	3	9
Weak updates at instantiation				
Running time (1 CPU)	7.1 min	4.8 min	8.1 min	60.0 min
Running time (8 CPUs)	4 min	3.6 min	2.5 min	10.1 min
Memory use	410 MB	250 MB	200 MB	355 MB
Error reports	312	209	730	1140
False positives	307	204	726	1112

Figure 18. Comparison of strong/weak updates at call sites

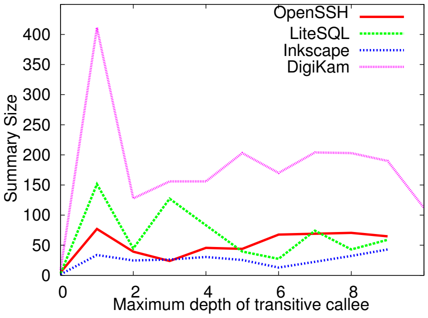


Figure 19. Callstack depth vs. summary size

	hostname	**chroot**	**rmdir**	**su**	**mv**
Lines	304	371	483	1047	1151
Modular analysis running time	0.53s	0.75s	1.54s	2.3s	2.55s
Whole program running time	3.1s	6.3s	21.6s	45.9s	30.7s

Figure 20. Comparison of modular and whole program analysis

H''_2 is obtained by removing the edge from G to H and adding edges from G to each T'_i. Thus, $\Pi_{H_2,H_1}(G) = \Pi_{H_2,H_1}(E) = C$ and G and E's targets are renamed to $T_1, \ldots, T_k$. (iii) Similar to (ii).

This lemma shows the principle that can be used to reduce the number of entries in $\mathbb{A}$: Assuming we can impose an order on the sequence of updates to memory locations and assuming we instantiate summary edges in this order, then the initial heap abstraction only needs to account for aliasing between α_1 and α_2 if there is a store to α_1 followed by a load from α_2, which is necessary because updates through α_1 to a location may now affect locations that are reachable through α_2. On the other hand, if there is no load after a store and the updates to memory locations can be ordered, it is possible to "fix up" the summary at the call site by respecting the order of updates during instantiation.

To allow such an optimization in the analysis described in Section 4, we impose a partial order $\prec$ on points-to relations such that $(\pi_1 \mapsto \theta_1) \prec (\pi_2 \mapsto \theta_2)$ indicates that π_1 must be assigned to θ_1 before π_2 is assigned to θ_2. Then, to respect the order of updates in the callee when instantiating the summary, we ensure that if $\pi_i \mapsto \theta_i \prec \pi_j \mapsto \theta_j$, the *compose_partial_heap* rule is invoked on $\pi_i \mapsto \theta_i$ before $\pi_j \mapsto \theta_j$ in the *compose_heap* rule of Figure 15.

Thus, assuming we modify the analysis from Section 4 as described above, we can compute a better alias partition environment $\mathbb{A}$ by performing a least fixed-point computation over the current function f. In particular, $\mathbb{A}(\alpha)$ is initialized to $\{\alpha\}$ for each location variable α. Then, if the analysis detects a store to α followed by a load from α' of the same type, then $\alpha' \in \mathbb{A}(\alpha)$ and $\alpha \in \mathbb{A}(\alpha')$. Similarly, if there is a store s_1 to α and a store s_2 to α' (of the same type) such that there is no happens-before relation between s_1 and s_2, then $\alpha' \in \mathbb{A}(\alpha)$ and $\alpha \in \mathbb{A}(\alpha')$.

6. Experiments

We have implemented the technique described in this paper in our COMPASS program verification framework for analyzing C and C++ applications. Our implementation extends the algorithm described in this paper in two ways: First, our analysis is fully (i.e., interprocedurally) path-sensitive and uses the algorithm of [8] for this purpose. Second, our implementation improves over the analysis presented here by employing the technique described in [4], which uses *indexed locations* to reason precisely about contents of arrays and containers. Hence, the algorithm we implemented is significantly more precise than a standard may points-to analysis.

Figure 18 summarizes the results of our first experiment, which involves verifying memory safety properties (buffer overruns, null dereferences, casting errors, and access to deleted memory) in four real C and C++ applications ranging from 16,030 to 128,318 lines. The first part of the table, labeled "Strong Updates at Instantiation", reports the results obtained by using the modular heap analysis described in this paper. Observe that the proposed technique is both scalable, memory-efficient, and precise. First, the running times on 8 CPU's range from 1.6 minutes to 8.7 minutes, and increase roughly linearly with the size of the application. Furthermore, observe that the modular analysis takes advantage of multiple CPUs to significantly reduce its wall-clock running time. Second, the maximum memory used by any process does not exceed 430 MB, and, most importantly, the memory usage is not correlated with the application size. Figure 19 sheds some light on the scalability of the analysis: This figure plots the maximum call stack depth against summary size, computed as the number of points-to edges weighted according to the size of the edge constraints plus the size of the precondition. In this figure, observe that summary size does not increase with depth of the callstack, confirming our hypothesis that summaries are useful for exploiting information locality and therefore enable analyses to scale.

Figure 18 also illustrates that performing strong updates at call sites is crucial for the precision required by verification tools. Observe that the analysis using strong updates at instantiation sites is very precise, reporting only a handful of false positives on all the applications. In contrast, if we use only weak updates when applying summaries, the number of false positives ranges from 200 to 1000, confirming that the application of strong updates interprocedurally is a key requirement for successful verification.

In a second set of experiments on smaller benchmarks, we compare the running times of our verification tool using the modular analysis described here with the running times of the same tool using a whole-program analysis. Figure 20 shows a comparison of the analysis running times of the modular and whole program analysis on five Unix Coreutils applications. As shown in this figure, the whole program analysis, which did not report any errors, takes ~50 seconds on a program with only 1000 lines, whereas the modular analysis, which also did not report any errors, analyzes the same program in 2.3 seconds. Furthermore, observe that the running time of the whole program analysis increases much more quickly in the size of the application than that of the modular analysis.

In a final set of experiments, we plot the size of the alias partition set vs. the frequency of this set size for the benchmarks from Figure 18. The solid (red) line shows the size of the alias partition sets obtained by assuming $\alpha' \in \mathbb{A}(\alpha)$ if α and α' have compati-

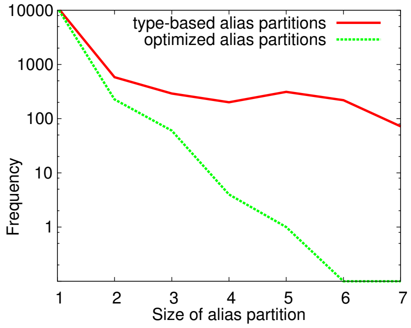


Figure 21. Size of alias partition set vs. Frequency

ble types. In contrast, the dashed (green) line shows the size of the alias partition sets obtained as described in Section 5. Observe that these optimizations significantly reduce the size of alias partition sets and substantially improve running time. In particular, without these optimizations, the benchmarks take an average of 2.7 times longer.

7. Related Work

Compositional Alias Analysis Modular alias analysis of a procedure performed by starting with unknown values to all parameters was also explored in [6] and then in Relevant Context Inference (RCI) [7]. The technique presented in [6] computes a new *partial transfer function* as new aliasing patterns are encountered at call sites and requires reanalysis of functions. In contrast, the technique in [7] is purely bottom-up, and uses equality and disequality queries to generate *summary transfer functions*. Our approach is similar to [7] in that we perform a strictly bottom-up analysis where the unknown points-to target of an argument is represented using one location variable and summary facts are predicated upon possible aliasing patterns at function entry. In contrast to our technique, RCI is only able to perform strong updates in very special cases intraprocedurally, and cannot perform strong updates at call sites. In fact, the summary computation described in [7] is only sound under the assumption that no points-to relations are killed by summary application. In contrast, summaries generated by our analysis are used to perform strong updates at call sites, and for the recursion-free fragment of the language from Section 3, applying a summary is as precise as analyzing the inlined body of the function.

The compositional pointer analysis algorithms given in [9, 10] assume there is no aliasing on function entry and analyze the function body under this assumption. However, since summaries computed in this way may be unsound, the summary is "corrected" using a fairly involved fixed-point computation at call sites. This approach is also much less precise than our technique because it only performs strong updates in a very limited number of situations.

Compositional Shape Analysis Recently, there has also been interest in compositional shape analysis using separation logic [11, 12]. Both of these works use *bi-abduction* to compute pre- and post-conditions on the shapes of recursive data structures. However, neither of these works guarantee precision. While this paper does not address computing summaries about shapes of recursive data structures, our technique can handle deep sharing and allows disjunctive facts.

General Modular Analysis Frameworks Theoretical foundations for modular program analysis are explored in [13], [14], and [15]. The work in [16] provides a framework for computing precise and concise summaries for IFDS [17] and IDE [18] dataflow problems. This framework is mainly specialized for typestate properties and relies on global points-to information. While it may be possible to apply this framework to obtain some form of modular heap analysis in principle, it is unclear how to do so, and the authors of [16] list this application as a future research direction.

References

[1] Aiken, A., Bugrara, S., Dillig, I., Dillig, T., Hackett, B., Hawkins, P.: An overview of the saturn project. In: PASTE, ACM (2007) 43–48

[2] Bush, W., Pincus, J., Sielaff, D.: A static analyzer for finding dynamic programming errors. Software: Practice and Experience **30**(7) (2000) 775–802

[3] Reps, T.W., Sagiv, S., Wilhelm, R.: Static program analysis via 3-valued logic. In: CAV. Volume 3114., Springer (2004) 15–30

[4] Dillig, I., Dillig, T., Aiken, A.: Fluid updates: Beyond strong vs. weak updates. In: ESOP. (2010)

[5] Landi, W., Ryder, B.G.: A safe approximate algorithm for interprocedural aliasing. SIGPLAN Not. **27**(7) (1992) 235–248

[6] Wilson, R.P., Lam, M.S.: Efficient context-sensitive pointer analysis for c programs. In: PLDI. (1995)

[7] Chatterjee, R., Ryder, B., Landi, W.: Relevant context inference. In: POPL, ACM (1999) 133–146

[8] Dillig, I., Dillig, T., Aiken, A.: Sound, complete and scalable path-sensitive analysis. In: PLDI, ACM (2008) 270–280

[9] Whaley, J., Rinard, M.: Compositional pointer and escape analysis for Java programs. In: OOPSLA, ACM (1999) 187–206

[10] Salcinau, A.: Pointer Analysis for Java Programs: Novel Techniques and Applications. PhD thesis, MIT (2006)

[11] Calcagno, C., Distefano, D., O'Hearn, P., Yang, H.: Compositional shape analysis by means of bi-abduction. POPL (2009) 289–300

[12] Gulavani, B., Chakraborty, S., Ramalingam, G., Nori, A.: Bottom-up shape analysis. SAS (2009) 188–204

[13] Cousot, P., Cousot, R.: Modular static program analysis. In: CC. (2002) 159–178

[14] Gulwani, S., Tiwari, A.: Computing procedure summaries for interprocedural analysis. ESOP (2007) 253–267

[15] Pnueli, M.: Two approaches to interprocedural data flow analysis. Program Flow Analysis: Theory and Applications (1981) 189–234

[16] Yorsh, G., Yahav, E., Chandra, S.: Generating precise and concise procedure summaries. POPL **43**(1) (2008) 221–234

[17] Reps, T.W., Horwitz, S., Sagiv, S.: Precise interprocedural dataflow analysis via graph reachability. In: POPL. (1995) 49–61

[18] Sagiv, S., Reps, T.W., Horwitz, S.: Precise interprocedural dataflow analysis with applications to constant propagation. Theor. Comput. Sci. **167**(1&2) (1996) 131–170