Symbolic Heap Abstraction with Demand-Driven Axiomatization of Memory Invariants *

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

Abstract

Many relational static analysis techniques for precise reasoning about heap contents perform an explicit case analysis of all possible heaps that can arise. We argue that such precise relational reasoning can be obtained in a more scalable and economical way by enforcing the memory invariant that every concrete memory location stores one unique value directly on the heap abstraction. Our technique combines the strengths of analyses for precise reasoning about heap contents with approaches that prioritize axiomatization of memory invariants, such as the theory of arrays. Furthermore, by avoiding an explicit case analysis, our technique is scalable and powerful enough to analyze real-world programs with intricate use of arrays and pointers; in particular, we verify the absence of buffer overruns, incorrect casts, and null pointer dereferences in OpenSSH (over 26,000 lines of code) after fixing 4 previously undiscovered bugs found by our system. Our experiments also show that the combination of reasoning about heap contents and enforcing existence and uniqueness invariants is crucial for this level of precision.

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

General Terms Languages, Verification, Experimentation

Keywords Heap Analysis, Relational Static Analysis, Array Analysis, Memory Invariants

* This work was supported by grants from NSF (CNS-050955, CCF-0430378) with additional support from DARPA.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

OOPSLA/SPLASH'10, October 17–21, 2010, Reno/Tahoe, Nevada, USA.
Copyright © 2010 ACM 978-1-4503-0203-6/10/10. . . $10.00

1. Introduction

In the past decade, there has been considerable progress in reasoning statically about the heap and contents of unbounded data structures. In particular, for reasoning about array contents, techniques such as [1–5], have focused on inferring and expressing interesting invariants that are shared between different elements of arrays. For example, consider the following code:

```
for(i = 0; i < n; i++) {
   if (*)
      a[i] = b[i]
   else
      a[i] = NULL;
}
```

Here we assume that the condition `(*)` is sufficiently complicated that whatever static analysis we are using cannot understand it. Even in the presence of such uncertainty, techniques for reasoning about the contents of arrays can still represent that for all `i` in the domain of arrays `a` and `b`, either `a[i]` is equal to `b[i]` or `a[i]` is `NULL`.

While having an accurate understanding of the contents of arrays is often necessary for proving non-trivial properties about real programs, this information alone is also often not sufficient. In the specific case of reasoning about arrays, one coarse but accurate intuition is that while these techniques are good at characterizing array writes, they can still lose information about array reads. Consider again the code fragment above. On exit from the loop, we know that `a[i]` is equal to either `NULL` or `b[i]`. While we do not which of the two values each `a[i]` holds, the information about the array contents is quite precise. In fact, it is the most precise information possible about what is written into array `a` given that we know nothing about the conditional's predicate. Now, consider the following code snippet, which immediately follows the loop above:

```
x = a[k];
y = a[k];
if(x != NULL)
    assert(y==b[k]);
```

What is needed to prove the assertion in this example? We need to know that (i) x is either NULL or b[k], (ii) y is also either NULL or b[k], (iii) the two successive reads from a[k] yield the same value regardless of a's contents. Precise heap analysis techniques such as [1, 3, 6–8] can naturally reason about (i) and (ii), but something more is needed to reason about (iii). The difficulty is the uncertainty involving the actual value of a[k]. If we proceed naively, the first read of a[k] can be NULL or b[k], and so x can be either value. Similarly, the second read of a[k] can be NULL or b[k], and so y can also be either value. Then, in reasoning about the assertion, it appears that x != NULL can hold (since one possibility is that x is b[k]) at the same time that y == NULL also holds (since one possibility is that y is NULL), and the assertion cannot be discharged. We have lost the relationship between x and y, namely that in all executions x == y.

Establishing property (iii) is very important because it allows *relational reasoning* in the presence of uncertainty by establishing correlations between values stored in different heap locations (e.g., the relationship between x and y above). A standard way to deal with this difficulty is to perform an explicit case split: Construct one heap abstraction H where a[k]'s value is NULL and another heap abstraction H' where a[k]'s value is b[k]. Since x and y *both* have the value NULL in heap H and *both* have the value b[k] in H', the equality of x and y can be established and the assertion is discharged [1, 6, 9]. Put another way, a case split on the possible values of a[k] allows us to know that both reads of a[k] in the example return the same value.

This paper is about avoiding case splits on the heap abstraction, which we consider problematic for both practical and philosophical reasons:

- Case splits on the heap are generally eager operations: as illustrated above, first the heap is split into the various possibilities and only then is the subsequent code analyzed. Thus, we pay the full price of the case analysis up front, without knowing whether the split is eventually needed to prove some property of interest.
- Case splits can (and do) result in an exponential blow-up: Every time an abstract location may point to n distinct memory locations, then n distinct copies of the heap must be created and separately analyzed, quickly resulting in an infeasible number of heap configurations. Even if the preceding point can be addressed and the case analysis is somehow performed lazily, the state space explosion problem from duplicating the abstract heaps still persists.
- The case splits are really just a form of disjunction (i.e., the disjunction of n possible worlds). Given that disjunction is already required to represent multiple possible contents of individual locations (e.g., a[k] may be either NULL or b[k]), it would be conceptually simpler and presumably easier to implement a system with only a single way of performing disjunctions.

In this paper, we address the problem of establishing relational reasoning without creating explicit copies of abstract heaps. We argue that the need for constructing duplicates of the heap arises from the lack of one very important and primitive invariant that real computer memories satisfy but that is not enforced directly by standard heap abstractions: first, every memory location has at least one value (*existence*) and second, every memory location has at most one value (*uniqueness*).

Consider the original heap abstraction described above, where a[k] may be either b[k] or NULL. As the informal reasoning we carried out suggests, this abstraction does not prevent a[k] from simultaneously being equal to both b[k] and NULL, and so even adjacent reads from a[k] cannot be proven to yield the same value. The case analysis, in essence, enforces the existence and uniqueness invariant by creating multiple disjoint heaps where the abstract memory location of interest has exactly one value.

The key insight underlying our technique is to create a single heap abstraction that enforces the existence and uniqueness invariants without requiring an explicit case analysis of heap values. To be concrete, consider a heap abstraction in which the possible points-to targets of a location a are x and y. Our technique qualifies points-to edges from a to x with a formula ϕ_x and the edge to y with a formula ϕ_y such that by construction, ϕ_x and ϕ_y are contradictory (guaranteeing that a cannot simultaneously point to both x and y, enforcing uniqueness) and their disjunction is valid (guaranteeing that a points to at least one of x or y in every possible world, enforcing existence). These formulas add no new mechanism, using the same language of formulas already needed just to describe the contents of the heap. The method is also inherently lazy; the formulas are small and all the work is deferred until constraint solving is performed. The main advantage of this symbolic approach is that, while a case analysis may eventually be needed in solving the constraints, constraint solvers often avoid the full case analysis because satisfiability or validity can often be easily established without examining the entire formula in detail, and furthermore several disjoint heaps do not need to be separately analyzed.

Enforcing existence and uniqueness of memory contents directly leads to precise relational reasoning. For instance, in the code example, suppose that the heap abstraction encodes a[k] is NULL under some constraint ϕ_1 and b[k] under some constraint ϕ_2 such that ϕ_1 and ϕ_2 are contradictory. Then, it is easy to see that x and y are equal to NULL under constraint ϕ_1 and equal to b[k] under constraint ϕ_2. Since ϕ_1 and ϕ_2 are contradictory, the heap abstraction directly encodes x

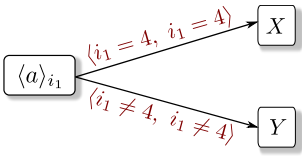

Figure 1. An exact symbolic heap

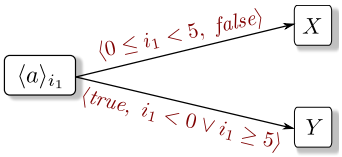

Figure 2. An inexact symbolic heap

and y must have the same value, allowing the assertion to be discharged.

To summarize, in this paper, we propose a technique that unifies reasoning about heap contents with enforcing the fundamental memory invariant that every concrete memory location has a unique value until its next write. Specifically, we extend the *symbolic heap abstraction* described in [4] for reasoning about heap contents to enforce existence and uniqueness of values stored in memory locations. Our approach combines the strengths of techniques for reasoning about heap contents, such as [1, 6, 9] with techniques that focus on the *axiomatization of memory invariants*, such as decision procedures like the *theory of arrays* [10].

1.1 A Quick Overview

In this subsection we give a high-level overview of the technical sections that follow. A *symbolic heap abstraction* [4] represents points-to relations in the heap as directed edges in a graph where nodes correspond to abstract memory locations. In general, abstract locations represent a non-empty set of concrete locations; for example, an array is represented by a single abstract location that represents all of the concrete elements of the array. Each points-to edge in the symbolic heap is labeled with a *bracketing constraint*, $\langle \phi_{may}, \phi_{must} \rangle$, identifying which concrete elements within a given abstract location *may* and *must* point to which concrete elements in the target location. Therefore, the symbolic heap abstraction simultaneously encodes both an over- and an underapproximation of the concrete heap. The simultaneous use of over- and underapproximations is useful in multiple ways, which are relevant to but not the topic of this paper. For example, bracketing constraints are needed in sound and precise path-sensitive analysis (and, in particular, in computing complements of path conditions) [11] and in defining a precise location update mechanism, called a *fluid update* [4]. The key soundness invariant of this symbolic heap abstraction is that the disjunction of all *may* conditions on edges outgoing from an abstract location A is valid, while the pairwise conjunction of any two *must* constraints on outgoing edges from A is unsatisfiable.

We say that a heap abstraction is *exact* if the over- and underapproximations are identical. An exact abstract heap describes precisely one concrete heap. Therefore, when the over- and underapproximations encoded by the symbolic heap are identical, the symbolic heap already encodes existence and uniqueness of values stored in memory locations. For example, the symbolic heap shown in Figure 1 is exact since the may and must conditions on points-to edges are identical. In particular, this abstract heap encodes a concrete heap where the fifth element of an array a points to X and all other elements point to Y. Observe that this symbolic heap encodes that no concrete element in array a can simultaneously point to both X and Y because the *may* conditions on the edges to X and Y are disjoint, thereby encoding uniqueness of the value stored in any concrete element in a. Similarly, this symbolic heap also encodes that every element in a has *some* value (i.e., existence) since the disjunction of the *must* conditions is *true*.

In practice, except for the simplest heaps, symbolic heaps are rarely exact. Consider the imprecise symbolic heap in Figure 2. This abstraction encodes that any element of array a in the range $[0, 4]$ *may* point to X, but no element *must* point to X. On the other hand, any element in the array *may* point to Y, but elements whose indices are not in the range $[0, 4]$ are guaranteed to point to Y. Such a symbolic heap no longer encodes existence and uniqueness of concrete elements; for example, elements in the range $[0, 4]$ may point to X or Y or neither. More technically, we can see that the conjunction of the may constraints is now satisfiable (allowing a memory location to point to two different places simultaneously), and the disjunction of the must constraints is not valid (allowing a memory location to possibly have no value at all).

Hence, as illustrated by these examples, while an exact symbolic heap, such as the one from Figure 1, encodes existence and uniqueness, the normal situation of an imprecise symbolic heap such as the one from Figure 2 does not. Observe that the use of bracketing constraints is not the source of this difficulty; any heap abstraction that encodes only an over- or an underapproximation is imprecise and will suffer from the same problem. In fact, bracketing constraints only improve the situation by making it explicit whether the abstraction enforces existence and uniqueness of memory contents.

To be able to reason about existence and uniqueness invariants in the presence of uncertainty without performing case splits, our approach augments the symbolic heap abstraction with a technique we call *demand-driven axiomatization* of memory invariants. Specifically, whenever a bracketing constraint on a points-to edge becomes imprecise (e.g., due to imprecise loop invariants or branches on values that are not statically known), our technique replaces this impre-

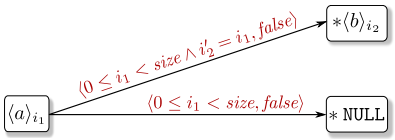

Figure 3. A symbolic heap abstraction

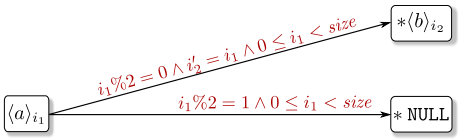

Figure 4. A precise heap abstraction

cise bracketing constraint with a special formula of the form

$$\Delta = \Delta_\delta \wedge \Delta_\tau$$

such that, by construction, the introduction of these Δ constraints enforces the existence and uniqueness of the value stored in each memory location. The demand-driven aspect of our method is again that we only introduce these extra constraints for edges in the points-to graph where the bracketing constraint is not exact.

Of course, we do not want to discard the information encoded by the original bracketing constraints because they might still provide useful information despite being imprecise. Hence, to combine reasoning about memory invariants and heap contents, our technique introduces a quantified axiom of the form

$$\forall i_1, \ldots i_m.\ \phi_{must} \Rightarrow \Delta_\delta \Rightarrow \phi_{may}$$

which preserves the partial information present in the imprecise heap. The introduction of these axioms enforces that any fact that can be proven under the original, but imprecise heap abstraction can still be proven to hold under the modified heap abstraction that enforces the existence and uniqueness of memory contents. Furthermore, this axiomatization strategy guarantees that the number of valid assertions that can be proven correct is monotonic with respect to the precision of the heap abstraction, a property that does not hold without enforcing existence and uniqueness of memory contents.

1.2 Organization and Contributions

The rest of this paper is organized as follows: Section 2 reviews the basic symbolic heap abstraction described in [4], and Section 3 describes how to evaluate assertions on this heap abstraction. Section 4 shows how to combine this heap abstraction with enforcing existence and uniqueness invariants. Section 5 describes our implementation; Section 6 presents experimental results. Finally, Section 7 surveys related work, and Section 8 concludes.

To summarize, this paper makes the following contributions:

- We argue that enforcing existence and uniqueness of memory contents allows for precise relational reasoning without performing an explicit case split on the possible concrete heaps that can arise.
- We propose *demand-driven axiomatization of memory invariants* as a way to combine the strength of symbolic heap abstraction with the ability to reason precisely in the presence of partial information.
- We define what it means for a heap abstraction to be more *precise* than another heap abstraction, and we show that the set of valid assertions that can be proven correct by our analysis is monotonic with the respect to the precision of the heap abstraction, a property that does not hold without enforcing existence and uniqueness of memory contents.
- We demonstrate that the combination of symbolic heap abstraction and demand-driven axiomatization is powerful and scalable enough to verify the absence of buffer overruns, incorrect casts, and null pointer dereferences in OpenSSH (over 26,000 lines) after fixing 4 unknown bugs discovered by our technique.
- We show experimentally that enforcing memory invariants is as important as reasoning about heap contents and that a substantial number of assertions requires combined reasoning about both heap contents and memory invariants.

2. Symbolic Heap Abstraction

In this section, we review the basic symbolic heap abstraction introduced in [4].

2.1 An Informal Overview

In a symbolic heap abstraction, each array is represented by a *single* abstract location qualified by an *index variable* ranging over the possible indices of the array. Invariants on array elements are expressed symbolically through constraints qualifying these index variables. A key feature of this technique is that it does not require constructing explicit partitions of the heap to perform strong updates and does not fix the "shape" of the invariants that are expressible by a given partitioning scheme a priori.

As an example, consider the symbolic heap abstraction in Figure 3, which represents an array `a` whose elements `a[i]` in the range $[0, size)$ may be equal to either `b[i]` or `NULL`. In this graph, nodes represent abstract memory locations, and directed edges represent points-to relations where the source of the edge points to the target of the edge. The abstract location $\langle a \rangle_{i_1}$ represents all elements of some array `a`, and its corresponding index variable i_1 ranges over the possible indices of this array. The abstract location $*\langle b \rangle_{i_2}$ represents the points-to targets of the elements of another array `b`, and $*$`NULL` represents the abstract location pointed to by `NULL`.

Constraint pairs $\langle \phi_{may}, \phi_{must} \rangle$ called *bracketing constraints* on the edges qualify the source and the target locations' index variables, selecting which concrete elements of the source *may* and *must* point to which concrete elements of the target. By convention, the target location's index variables are always primed in the constraints. If ϕ_{may} and ϕ_{must} are the same, we write a single constraint instead of a pair. The constraint

$$\langle 0 \leq i_1 < \mathit{size} \wedge i_2' = i_1, \mathit{false} \rangle$$

on the edge from $\langle a \rangle_{i_1}$ to $*\langle b \rangle_{i_2}$ indicates that all elements of array `a` in the range $[0, \mathit{size})$ may point to the same locations as the corresponding elements of array `b` (indicated by $i_2' = i_1$), but since ϕ_{must} is *false*, no element must point to the corresponding element in `b[i]`. Similarly, the constraint

$$\langle 0 \leq i_1 < \mathit{size}, \mathit{false} \rangle$$

on the edge between $\langle a \rangle_{i_1}$ and $*$`NULL` indicates that any element in the range $[0, \mathit{size})$ may be, but does not have to be, `NULL`.

A main motivation for using bracketing constraints in the points-to graph is to allow a more precise and general update mechanism, called *fluid update*, than techniques that identify all updates as either strong or weak. A strong update to an abstract location l removes all existing points-to edges from l, whereas a weak update preserves all existing edges. However, since an abstract location may correspond to many concrete locations, some points-to edges from l are removed by an update whereas others are not. To express the range of possibilities between these two extremes without case-splitting on an abstract location, a fluid update computes a constraint Ψ describing which elements of l are affected by the update, and adds a new edge under constraint Ψ while preserving existing outgoing edges of l under $\neg\Psi$. However, since it is, in general, impossible to obtain an exact description of the set of elements updated in loops, Ψ is often an overapproximation; thus, $\neg\Psi$ is an *underapproximation* of the elements *not* affected by the update. For this reason, fluid updates require that constraints used in the heap abstraction are bracketing constraints. Now, if $\langle \phi_{may}, \phi_{must} \rangle$ represents the concrete elements that *may* and *must* be updated,

$$\neg\langle \phi_{may}, \phi_{must} \rangle \;\Leftrightarrow\; \langle \neg\phi_{must}, \neg\phi_{may} \rangle$$

soundly identifies the elements that may and must be unchanged after the update.

In addition to allowing a precise and general update mechanism, bracketing constraints also make it explicit when the heap abstraction is exact. For example, in Figure 4, since ϕ_{may} and ϕ_{must} are the same (indicated by a single constraint instead of a pair), this heap encodes that every *even* element (indicated by $i_1 \% 2 = 0$) in the range $[0, \mathit{size})$ must be equal to `b[i]` whereas every odd element must be `NULL`. This heap might be a refinement of the symbolic heap from Figure 3, obtained, for example, using a more precise loop invariant.

2.2 Formal Definition of Symbolic Heaps

In a symbolic heap, access paths [12] name abstract locations and are defined by:

$$\textit{Access Path } \pi := l \mid \langle \pi \rangle_i \mid *\pi \mid s$$

Here, l names an abstract location corresponding to a variable or fresh allocation. Any array location is represented by an access path $\langle \pi \rangle_i$, where π represents the array and i is an *index variable* ranging over π's indices. The access path $*\pi$ represents the dereference of π. Finally, s denotes a scalar value, such as `NULL`. Access paths may contain multiple index variables, such as when modeling nested arrays. For example, if `x` is an array of pointer arrays, then $\langle *\langle x \rangle_{i_1} \rangle_{i_2}$ names the abstract location associated with each of the nested arrays.

Access paths may be converted to terms in the constraint language when they are used in constraints. Base locations l are represented by variables of the same name; the access path $\langle \pi \rangle_i$ is represented by the function term $\mathit{arr}(\pi, i)$ where *arr* is an uninterpreted function. The access path $*\pi$ is represented as the uninterpreted function term $\mathit{deref}(\pi)$, and finally s is represented by a constant of the same value. When access paths appear in constraints, we assume they are implicitly converted to terms.

DEFINITION 1. **(Bracketing Constraints)** A bracketing constraint $\langle \phi_{may}, \phi_{must} \rangle$ is a pair of constraints with the property $\phi_{must} \Rightarrow \phi_{may}$.

In general, ϕ_{may} and ϕ_{must} may be over any theory; in this paper, we only consider formulas in the combined theory of uninterpreted functions and linear integer arithmetic extended with divisibility (mod) predicates. In the remainder of this paper, any constraint is implicitly understood to be a bracketing constraint.

DEFINITION 2. **(Boolean operations)**

$$\begin{aligned}
\neg\langle \phi_{\text{may}}, \phi_{\text{must}} \rangle &= \langle \neg\phi_{\text{must}}, \neg\phi_{\text{may}} \rangle \\
\langle \phi_{\text{may1}}, \phi_{\text{must1}} \rangle \wedge \langle \phi_{\text{may2}}, \phi_{\text{must2}} \rangle &= \langle \phi_{\text{may1}} \wedge \phi_{\text{may2}}, \phi_{\text{must1}} \wedge \phi_{\text{must2}} \rangle \\
\langle \phi_{\text{may1}}, \phi_{\text{must1}} \rangle \vee \langle \phi_{\text{may2}}, \phi_{\text{must2}} \rangle &= \langle \phi_{\text{may1}} \vee \phi_{\text{may2}}, \phi_{\text{must1}} \vee \phi_{\text{must2}} \rangle
\end{aligned}$$

Satisfiability and validity of bracketing constraints is defined as follows:

DEFINITION 3. **(Satisfiability and Validity)**

$$\begin{aligned}
\text{SAT}(\langle \phi_{\text{may}}, \phi_{\text{must}} \rangle) &= \text{SAT}(\phi_{\text{may}}) \\
\text{VALID}(\langle \phi_{\text{may}}, \phi_{\text{must}} \rangle) &= \text{VALID}(\phi_{\text{must}})
\end{aligned}$$

DEFINITION 4. **(Symbolic Heap Abstraction)** A *symbolic heap abstraction* is a directed graph where nodes labeled with access paths denote abstract memory locations, and an edge from abstract location π to π' indicates that some concrete location in π may point to another concrete location in π'. Edges are labeled with bracketing constraints $\langle \phi_{may}, \phi_{must} \rangle$, where ϕ_{may} and ϕ_{must} respectively constrain which concrete elements of the abstract source location *may* and *must* point to which concrete elements of the target location.

3. Proving Assertions on the Symbolic Heap

In this section, we show how to prove assertions using the information encoded by the symbolic heap abstraction. To be able to prove assertions, we first need a way to retrieve the possible values stored in a location.

3.1 Determining Points-to Targets

First, as mentioned earlier and illustrated in the examples, the formulas on edges allow us to talk about properties of subsets of abstract memory locations. For example, in the formula in Figure 4, we can see that odd elements of the array point to NULL. Second, as is standard, we can use substitution to interpret which concrete source location may (or must) point to which concrete target. So, in the constraint on the edge from $\langle a \rangle_{i_1}$ to $*\langle b \rangle_{i_2}$, if we substitute 2 for index variable i_1, we obtain:

$$2\%2 = 0 \wedge i_2' = 2 \wedge 0 \leq 2 < \mathit{size}$$

Hence, assuming *size* is at least 3, this is equivalent to $i_2' = 2$, which tells us that a[2] is equal to b[2].

However, substitution is not powerful enough. In general, we may be reading from different indices of the array under different program conditions or we may be unsure about the value of the program variable that is used as an index into the array. For this reason, the index that is read from the array is described by an arbitrary constraint rather than a simple equality. For example, the constraint $(i = 2 \wedge \mathit{flag}) \vee (i = 5 \wedge \neg \mathit{flag})$ describes reading the second element of an array under program condition flag, but the fifth element under $\neg$flag. Similarly, for the array read a[v], we might know that v has some value less than 4, but we might not know its exact value; hence, the array indices that are read are described by the imprecise constraint $\langle i < 4, \mathit{false} \rangle$.

The right tool, then, for deducing possible values stored in a memory location is existential quantifier elimination, which generalizes substitution to arbitary constraints. For example, if we want to determine the result of reading an index whose exact value we do not know but that is definitely greater than 2, we can conjoin the constraint $\langle i_1 > 2, \mathit{false} \rangle$ with the appropriate edge constraint and then eliminate the existentially quantified variable i_1. For example, for the edge from $\langle a \rangle_{i_1}$ to $*\langle b \rangle_{i_2}$ in Figure 4, this would yield:

$$\exists i_1.(\langle i_1 > 2, \mathit{false} \rangle \wedge (i_1\%2 = 0 \wedge i_2' = i_1 \wedge 0 \leq i_1 < \mathit{size}))$$

After eliminating i_1, we obtain:

$$\langle i_2'\%2 = 0 \wedge 2 < i_2' < \mathit{size}, \mathit{false} \rangle$$

which tells us that the result of the read could be any even-indexed element of array b.

To be precise, we define a read operation on the heap abstraction, $\mathit{read}(\pi, \gamma)$, which given an abstract location π and a constraint γ on the index variables of π, yields a set of (access path, bracketing constraint) pairs representing the possible results of the read.

DEFINITION 5. ($\mathit{read}(\pi, \gamma)$) Let π be an abstract memory location, and let $\gamma = \langle \phi_{\mathit{may}}, \phi_{\mathit{must}} \rangle$ be a constraint such that ϕ_{may} selects at least one concrete element and ϕ_{must} selects at most one concrete element of π. Let e be an edge from π to π_i qualified by constraint ϕ_i in the symbolic heap, and let $\vec{I}$ be the vector of index variables in π. Then, let

$$\phi_i' = \mathit{Eliminate}(\exists \vec{I}.\gamma \wedge \phi_i)$$

where *Eliminate* performs existential quantifier elimination[1]. Finally, let ϕ_i'' be obtained by renaming primed (i.e., target's) index variables in ϕ_i' to their unprimed counterparts. Then:

$$(\pi_i, \phi_i'') \in \mathit{read}(\pi, \gamma)$$

EXAMPLE 1. *Consider the heap from Figure 3. Here, we have:*

$$\begin{aligned} \text{read}(\langle a \rangle_{i_1}, i_1 = 2) \quad = \quad & \{(*\langle b \rangle_{i_2}, \langle i_2 = 2 \wedge 2 < \text{size}, \text{false} \rangle), \\ & (*\texttt{NULL}, \langle 2 < \text{size}, \ \text{false} \rangle))\} \end{aligned}$$

3.2 Proving Assertions

Now, using this read operation, we describe how to evaluate simple assertions on a given symbolic heap configuration. We define an assertion primitive $\mathit{assert}(S = S')$ where $S = \mathit{read}(\pi, \gamma)$ and $S' = \mathit{read}(\pi', \gamma')$ for some arbitrary abstract locations π, π' and some index constraints γ, γ'. Intuitively, such an assertion is valid if the heap abstraction encodes that the values stored in the concrete locations identified by π, γ and π', γ' must be equal.

DEFINITION 6. **(Validity of Assertion)** Consider the assertion:

$$\mathit{assert}(\mathit{read}(\pi, \gamma) = \mathit{read}(\pi', \gamma'))$$

Let $(\pi_i, \phi_i) \in \mathit{read}(\pi, \gamma)$ and $(\pi_j', \phi_j') \in \mathit{read}(\pi', \gamma')$. Let $\vec{I_i}$ and $\vec{I_j'}$ be the index variables used in each π_i and π_j', let $\vec{F_i}, \vec{F_j'}$ denote fresh vectors of index variables, and let $\vec{F} = \bigcup_i \vec{F_i}$, $\vec{F'} = \bigcup_j \vec{F_j'}$. The assertion is *valid* if:

$$\mathit{VALID}\left(\exists \vec{F}, \vec{F'}. \quad \bigvee_{i,j} \left(\begin{array}{l} \pi_i[\vec{F_i}/\vec{I_i}] = \pi_j'[\vec{F_j'}/\vec{I_j'}] \\ \wedge\ \phi_i[\vec{F_i}/\vec{I_i}] \wedge \phi_j'[\vec{F_j'}/\vec{I_j'}] \end{array} \right) \right)$$

Intuitively, this definition first computes the constraint under which the two sets obtained from $\mathit{read}(\pi, \gamma)$ and $\mathit{read}(\pi', \gamma')$ are equal. As expected, this is a disjunction of all pairwise equalities of the elements in the two sets, i.e., a case analysis of their possible values. Now, for the assertion to be valid, this constraint must be valid. Observe that the constraints in this definition are all bracketing constraints, and the validity of bracketing constraints from Definition 2 uses the under-approximations $\phi_{i_{\mathit{must}}}, \phi'_{j_{\mathit{must}}}$ such that

$$\phi_{i_{\mathit{must}}} \Rightarrow (\pi = \pi_i) \textit{ and } \phi'_{j_{\mathit{must}}} \Rightarrow (\pi' = \pi'_j)$$

[1] Existential quantifier elimination in the combined theory of uninterpreted functions and linear integer arithmetic may not always be exact; however, since our technique uses bracketing constraints, we compute quantifier-free over- and underapproximations [13].

Hence, the validity of the above formula guarantees that the values of π and π' must be equal. Also, note that the renaming of index variables to fresh variables $\vec{F}, \vec{F'}$ is necessary to avoid naming collisions when π_i and π'_j share index variables. This can arise, for example, when π_i and π'_j refer to distinct concrete elements in the same abstract location.

We conclude this section with an example illustrating that the symbolic heap does not allow discharging a simple assertion because it does not enforce existence and uniqueness of memory contents in the presence of imprecision:

EXAMPLE 2. *Consider evaluating the following assertion on the heap from Figure 3:*

```
x=a[2];  y=a[2];  assert(x==y)
```

The possible values $V(x)$ and $V(y)$ of x *and* y *are obtained from $V(x) = V(y) = \text{read}(\langle a \rangle_{i_1}, i_1 = 2)$. Hence, assuming* size > 2*, we have:*

$$V(x) = V(y) = \left\{ \begin{array}{l} (*\langle b \rangle_{i_2}, \langle i_2 = 2, \text{false} \rangle), \\ (*\texttt{NULL}, \langle \text{true}, \text{false} \rangle)) \end{array} \right\}$$

Now, to evaluate the assertion, we query:

$$\text{VALID} \left(\exists f_1, f_2, f_3. \begin{array}{l} ((*\langle b \rangle_{f_1} = *\langle b \rangle_{f_2}) \wedge \langle f_1 = 2, \text{false} \rangle \wedge \\ \langle f_2 = 2, \text{false} \rangle) \vee \\ ((*\langle b \rangle_{f_3} = *\texttt{NULL}) \wedge \langle f_3 = 2, \text{false} \rangle \wedge \\ \langle \text{true}, \text{false} \rangle) \vee \\ ((*\texttt{NULL} = *\texttt{NULL}) \wedge \langle \text{true}, \text{false} \rangle \wedge \\ \langle \text{true}, \text{false} \rangle) \end{array} \right)$$

The result is false because the sufficient conditions (i.e., ϕ_{must}) of all the bracketing constraints are false. As this example illustrates, we cannot prove the validity of this simple assertion using the information encoded by the heap abstraction because the heap abstraction described so far does not enforce the memory invariant that every concrete location must have exactly one value.

4. Demand-Driven Axiomatization of Memory Invariants

The overapproximation encoded in the symbolic heap enforces that every abstract location must have at least one target for any possible index, while the underapproximation enforces that a specific concrete location cannot point to multiple concrete elements. Thus, if the heap abstraction is exact (i.e., the over- and underapproximations are the same, as in Figure 4), it follows immediately that the symbolic heap enforces the existence and uniqueness of memory contents. More formally, a key soundness requirement for the symbolic heap abstraction can be stated as follows:

DEFINITION 7. **(Soundness Requirement)** *Let π be a source location in the heap abstraction, and let*

$$\{\langle \phi_{\text{may}_1}, \phi_{\text{must}_1} \rangle, \ldots \langle \phi_{\text{may}_k}, \phi_{\text{must}_k} \rangle\}$$

be the constraints qualifying outgoing edges from π. Let I_i denote the primed index variables used in each constraint ϕ_i. Then,

$$\text{VALID}(\exists \vec{I}. \bigvee_i \phi_{\text{may}_i})$$

and

$$\text{UNSAT}(\exists \vec{I}. \phi_{\text{must}_i} \wedge \phi_{\text{must}_j}) \text{ for } i \neq j$$

However, observe that the soundness of the symbolic heap does not require the following invariants:

$$\mathit{UNSAT}(\exists \vec{I}. \phi_{\mathit{may}_i} \wedge \phi_{\mathit{may}_j})$$

$$\mathit{VALID}(\exists \vec{I}. \bigvee_i \phi_{\mathit{must}_i})$$

Thus, if the heap abstraction is not exact, as is often the case in any heap analysis, the overapproximation does not enforce that each concrete source has *at most* one concrete target, and the underapproximation does not enforce that each concrete source has *at least* one concrete target. Unfortunately, as we saw in Example 2, the lack of these invariants often prevents proving even simple assertions in the presence of imprecision.

In this section, we describe how to combine symbolic heap abstraction with enforcing existence and uniqueness of memory contents. The key idea underlying demand-driven axiomatization is to replace any imprecise bracketing constraint (i.e., $\phi_{\mathit{may}} \not\Leftrightarrow \phi_{\mathit{must}}$) with a constraint Δ serving two purposes: (i) it enforces that for each concrete source location, there is exactly one target location it can point to, and (ii) it allows us to retain all the information encoded in the original over- and underapproximations. We first develop (i), then (ii).

4.1 Enforcing Existence and Uniqueness

To enforce that concrete locations have exactly one target location (i.e., (i)), these Δ constraints must have the following properties:

1. They should enforce that the constraints on any pair of edges outgoing from the same abstract source are disjoint (required for uniqueness) and that there is at least one feasible abstract target location under any satisfiable index constraint (required for existence).
2. If there is an edge from π_s to π_t, the Δ's should enforce that any concrete element in π_s can point to *at most* one concrete target in π_t (also required for uniqueness).
3. The introduction of Δ's should not prevent different concrete elements in the same abstract location from pointing to the same target.

Of these, (1) and (2) are necessary to enforce the desired existence and uniqueness invariant, while (3) is necessary for soundness. By construction, these Δ's are of the form:

$$\Delta = \Delta_\delta \wedge \Delta_\tau$$

where Δ_δ enforces (1) and Δ_τ enforces (2), both while respecting (3). We first describe the construction of Δ_δ and then Δ_τ.

Given a source location π_s with index variables $\vec{I_s}$, let $\pi_{t_0}, \ldots, \pi_{t_k}$ be the set of targets of all outgoing edges from π_s. For the j'th edge from π_s to π_{t_j}, we construct Δ^j_δ as follows:

$$\Delta^j_\delta = \begin{cases} \delta_{\pi_s}(i_1, \ldots, i_m) \leq 0 & \text{if } j = 0 \\ \delta_{\pi_s}(i_1, \ldots, i_m) = j & \text{if } 0 < j < k \\ \delta_{\pi_s}(i_1, \ldots, i_m) \geq k & \text{if } j = k \end{cases} \quad (1)$$
$$\text{where } i_1, \ldots, i_m \in \vec{I_s}$$

By construction, each set of Δ_δ's for a location π_s enforces that the outgoing edge constraints are pairwise contradictory and their disjunction is valid. Here, δ_{π_s} is an uninterpreted function symbol unique to location π_s. For an abstract location containing m index variables, it is necessary to introduce an *m-ary* uninterpreted function symbol in order to enforce the soundness requirement (3). Observe that, for concrete assignments $\vec{v}, \vec{v'}$ to index variables $\vec{I_s}$ of π_s, $\delta_{\pi_s}(\vec{v})$ must be equal to $\delta_{\pi_s}(\vec{v'})$ only if $\vec{v} = \vec{v'}$. Hence, while the Δ_δ constraints prevent the same concrete source from having different targets, they do not force two distinct concrete locations in the same abstract source to have the same target.

We now consider how to construct Δ_τ. Recall that Δ_τ must enforce that a given concrete source location cannot have multiple concrete targets in the same abstract target location (i.e., (2)), a property that is not enforced by the Δ_δ constraints. Hence, to satisfy (2), we construct Δ_τ as follows. Let $i'_{j_1}, \ldots, i'_{j_n}$ be the index variables used in the j'th target π_{t_j}. Then,

$$\Delta^j_\tau = \bigwedge_{1 \leq k \leq n} i'_{j_k} = \tau_k(i_1, \ldots, i_m) \quad (2)$$

Here, τ_k is an uninterpreted function symbol unique to the k'th index variable of the target. Δ^j_τ stipulates that each index variable used in the target is a function of the source's index variables, thereby enforcing that each concrete source can have at most one concrete target in the same abstract target location. Finally, to enforce both requirements (1) and (2), we modify the constraint on the j'th outgoing edge from π_s to be:

$$\Delta^j = \Delta^j_\delta \wedge \Delta^j_\tau$$

LEMMA 1. *Let $e_1, \ldots e_k$ be the set of outgoing edges from an abstract location π_s. Let $\Delta^1, \ldots, \Delta^k$ be the new set of constraints constructed as above qualifying $e_1, \ldots, e_k$. Then, the symbolic heap abstraction enforces that each concrete source location must point to exactly one concrete target location, or alternatively, that each concrete location has exactly one value.*

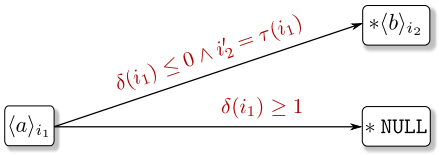

Figure 5. The modified version of the heap from Figure 3 enforcing existence and uniqueness invariants

First, we argue that the same concrete source cannot point to two different concrete targets. Let $\pi_s[\vec{s}/\vec{i}]$ denote a concrete source, obtained by a variable assignment $\vec{s}$ to the index variables $\vec{i}$ of π_s. Let $\pi_{t_1}[\vec{t_1}/\vec{i_{t_1}}]$ and $\pi_{t_2}[\vec{t_2}/\vec{i_{t_2}}]$ be two concrete targets, obtained by variable assignments $\vec{t_1}, \vec{t_2}$ to index variables $\vec{i_{t_1}}, \vec{i_{t_2}}$ of abstract locations π_{t_1} and π_{t_2}. If $\pi_{t_1}[\vec{t_1}/\vec{i_{t_1}}]$ and $\pi_{t_2}[\vec{t_2}/\vec{i_{t_2}}]$ are different, then either (i) π_{t_1} and π_{t_2} are different abstract locations, or (ii) $\vec{t_1} \neq \vec{t_2}$. For (i), observe that this is not possible since *UNSAT*$(\Delta^j_\delta[\vec{s}/\vec{i}] \wedge \Delta^k_\delta[\vec{s}/\vec{i}] \wedge j \neq k)$ for two edges e_j and e_k. For (ii), observe that:

$$\Delta^j_\tau = \left(\vec{t_1} = \begin{pmatrix} \tau_1(\vec{s}) \\ \ldots \\ \tau_n(\vec{s}) \end{pmatrix}\right) \text{ and } \Delta^j_\tau = \left(\vec{t_2} = \begin{pmatrix} \tau_1(\vec{s}) \\ \ldots \\ \tau_n(\vec{s}) \end{pmatrix}\right)$$

contradicting $\vec{t_1} \neq \vec{t_2}$. Now, we argue why each concrete location $\pi_s[\vec{s}/\vec{i}]$ must have at least one concrete target. Let $\vec{i'}$ denote the primed index variables used in the constraints on outgoing edges from π_s. Observe that the formula $\exists \vec{i'}. \bigvee_{0 \leq j \leq k} \Delta_j[\vec{s}/\vec{i}]$ is valid; thus each concrete source must have at least one concrete target. The following example shows that, using the modified symbolic heap, we can now prove assertions that could not be discharged using the basic symbolic heap.

EXAMPLE 3. *Consider the heap from Figure 3. For the edge from $\langle a \rangle_{i_1}$ to $*\langle b \rangle_{i_2}$, we construct*

$$\Delta_1 = (\delta(i_1) \leq 0 \wedge i'_2 = \tau(i_1))$$

and for the edge from $\langle a \rangle_{i_1}$ to `*NULL`*, we construct*

$$\Delta_2 = \delta(i_1) \geq 1$$

This modified heap is shown in Figure 5. Now, consider evaluating the assertion:

```
x = a[2];  y = a[2];  assert(x == y);
```

on this modified heap as described in Section 3. As before, the values $V(x)$ and $V(y)$ of `x` *and* `y` *are given by* read$(\langle a \rangle_{i_1}, i_1 = 2)$*:*

$$V(x) = V(y) = \begin{array}{l} \{(*\langle b \rangle_{i_2}, \delta(2) \leq 0 \wedge i_2 = \tau(2)), \\ (*\texttt{NULL},\ \delta(2) \geq 1))\} \end{array}$$

Now, to evaluate the assertion, we query:

$$\text{VALID}\left(\exists f_1, f_2, f_3.\ \begin{array}{l}(*\langle b\rangle_{f_1} = *\langle b\rangle_{f_2}) \wedge \delta(2) \leq 0 \\ \wedge f_1 = \tau(2) \wedge \delta(2) \leq 0 \wedge f_2 = \tau(2)) \vee \\ ((*\langle b\rangle_{f_3} = *\texttt{NULL}) \wedge \delta(2) \leq 0 \\ \wedge f_3 = \tau(2) \wedge \delta(2) \geq 1) \vee \\ ((*\texttt{NULL} = *\texttt{NULL}) \wedge \delta(2) \geq 1 \\ \wedge \delta(2) \geq 1)\end{array}\right)$$

In the first disjunct, $f_1 = f_2$, hence $(*\langle b\rangle_{f_1} = *\langle b\rangle_{f_2}) =$ true*. Simplifying this formula, we obtain:*

$$\exists f_1, f_2.\ (\ (\delta(2) \leq 0 \wedge f_1 = \tau(2) \wedge f_2 = \tau(2))\ \vee \ldots \vee\ \delta(2) \geq 1)$$

This formula is indeed valid, and we can now prove the assertion.

4.2 Preserving Existing Partial Information

We now consider the second part of demand-driven axiomatization: Recall that while replacing the imprecise edge constraints with the new Δ constraints ensures that every concrete source location points to exactly one concrete target, we would still like to retain the partial information present in the original, but imprecise heap abstraction. As an example, consider the following assertion:

```
if(a[2] != NULL) assert(a[2] == b[2]);
```

Clearly, the heap abstraction from Figure 3 encodes enough information to prove this assertion, however, the modified heap from Figure 5 no longer retains sufficient information to reason that a[2] must be either b[2] or NULL. In particular, the constraint on the edge to $*\langle b\rangle_{i_2}$ does not stipulate that $i'_2 = i_1$; hence, we do not know *which* element in $*\langle b\rangle_{i_2}$ a[2] points to; we only know that it points to *some* unique element if $\delta(2) \leq 0$ is satisfied.

Hence, to preserve the information encoded by the original imprecise bracketing constraints, we introduce axioms for each Δ^j_δ that encode the additional partial information present in the original symbolic heap. Let $\langle \phi^j_{may}, \phi^j_{must}\rangle$ be an imprecise bracketing constraint (i.e., $\phi^j_{may} \not\Leftrightarrow \phi^j_{must}$) on the j'th outgoing edge from source location π_s, and let Δ^j_δ be a constraint obtained as described above. As before, $\vec{I_s}$ denotes the index variables in π_s. Let σ_τ be a substitution replacing each target index variable with its corresponding $\tau_k(i_1, \ldots, i_m)$ from Equation 2. Then, to preserve the information present in the original heap abstraction, our technique introduces the axioms:

$$\forall \vec{I_s}.\ \sigma_\tau(\phi^j_{must}) \Rightarrow \Delta^j_\delta \quad \textit{and} \quad \forall \vec{I_s}.\ \Delta^j_\delta \Rightarrow \sigma_\tau(\phi^j_{may})$$

First, observe that Δ^j_δ, ϕ^j_{must}, and ϕ^j_{may} all qualify the source location's index variables. Since the heap abstraction states properties about any concrete location that satisfies the index constraint on edges, the source's index variables are all universally quantified in these axioms. Additionally, observe that ϕ^j_{may} and ϕ^j_{must} may also constrain the relationship between the source and the target's index variables, e.g., $i'_2 = i_1$. Since Δ^j_τ stipulates that each index variable used in the target is a function $\tau_k(i_1, \ldots, i_m)$ of the source's index variables, we apply the substitution σ_τ to both ϕ^j_{may} and ϕ^j_{must}. These axioms therefore restrict which set of concrete elements may and must be selected by each Δ^j_δ as stipulated by ϕ^j_{may} and ϕ^j_{must} as well as restricting the relationship between the source and the target's index variables.

As the following example shows, symbolic heap abstraction with demand-driven axiomatization allows combined reasoning about memory contents and invariants.

EXAMPLE 4. *Consider again the heap from Figure 3 and the modified heap from Figure 5. Our technique now introduces the following axioms:*

$$\begin{array}{l}\forall i_1.\ \delta(i_1) \leq 0\ \Rightarrow\ (0 \leq i_1 < \text{size} \wedge i_1 = \tau(i_1)) \\ \forall i_1.\ \text{false} \Rightarrow\ \delta(i_1) \leq 0\end{array}$$

$$\begin{array}{l}\forall i_1.\ \delta(i_1) \geq 1\ \Rightarrow\ 0 \leq i_1 < \text{size} \\ \forall i_1.\ \text{false} \Rightarrow\ \delta(i_1) \geq 1\end{array}$$

Now, consider the assertion:

$$\texttt{if}(\texttt{a}[2]! = \texttt{NULL})\ \texttt{assert}(\texttt{a}[2] == \texttt{b}[2])$$

As before:

$$\begin{array}{rl}\text{read}(\langle a\rangle_{i_1}, i_1 = 2) = & \{(*\langle b\rangle_{i_2}, \delta(2) \leq 0 \wedge i_2 = \tau(2)), \\ & (*\texttt{NULL}, \delta(2) \geq 1))\}\end{array}$$

and

$$\text{read}(\langle b\rangle_{i_2}, i_2 = 2) = \{(*\langle b\rangle_{i_2}, i_2 = 2)\}\}$$

Since the conditional requires that a[2] *is non-null, the assertion is guarded by the predicate:*

$$\neg(\delta(2) \geq 1)$$

Now, we need to show the validity of the formula

$$\exists f_1, f_2.\quad \begin{array}{l}*\langle b\rangle_{f_1} = *\langle b\rangle_{f_2} \wedge \delta(2) \leq 0 \wedge f_1 = \tau(2) \wedge f_2 = 2 \\ \vee\ (*\texttt{NULL} = *\langle b\rangle_{f_2} \wedge \delta(2) \geq 1 \wedge f_2 = 2)\end{array}$$

under the assumption $\neg(\delta(2) \geq 1)$. Simplifying the formula with respect to the assumption $\neg(\delta(2) \geq 1)$, we obtain:

$$\exists f_1, f_2.\quad *\langle b\rangle_{f_1} = *\langle b\rangle_{f_2} \wedge f_1 = \tau(2) \wedge f_2 = 2$$

Hence, it remains to show that under our axioms, $\tau(2)$ must be equal to 2*. Since one of the axioms is*

$$\forall i_1.\ \delta(i_1) \leq 0\ \Rightarrow\ (0 \leq i_1 < \text{size} \wedge i_1 = \tau(i_1))$$

it follows that:

$$\delta(2) \leq 0\ \Rightarrow\ (0 \leq 2 < \text{size} \wedge 2 = \tau(2))$$

Since $\delta(2) \leq 0$ is implied by the assertion guard, we have $\tau(2) = 2$; hence $f_1 = f_2$, establishing the validity of the asssertion.

While deciding quantified formulas in the combined theory of uninterpreted functions and linear integer arithmetic is, in general, undecidable, the axioms introduced by our technique belong to a decidable fragment, sometimes referred to as the *macro* fragment [14]. In particular, a syntactic instantiation of the axioms for each occurrence of the function term $\delta(\vec{t})$ is sufficient for completeness.

4.3 Monotonicity of Provable Assertions

If a heap abstraction does not enforce existence and uniqueness of memory contents, it turns out that it is possible to learn more about the contents of the heap while being able to prove strictly fewer assertions about the program! In other words, for such a heap abstraction, the number of provable assertions is not monotonic with respect to the precision of the heap abstraction. For instance, in Example 2, if we use a less precise heap abstraction that maps each element of `a` to an unknown location, we can prove the assertion `assert(x == y)`, which we cannot prove using the more precise heap from Figure 3.

We now describe what it means for a symbolic heap to be more precise than another heap abstraction of the same program, and we show that our technique never proves fewer assertions about the program using a more precise heap abstraction. For a heap H and a concrete location l, we use the notation $\alpha_H(l)$ to denote the abstract location that includes l in H. We write $\gamma(\pi)$ to denote the set of concrete locations that are represented by some abstract location π.

DEFINITION 8. *We say a symbolic heap H'* splits *an abstract location π in H into locations $\pi_1, \ldots, \pi_k$ (where $\gamma(\pi) = \gamma(\pi_1) \cup \ldots \cup \gamma(\pi_k)$) under constraints $\phi_1, \ldots, \phi_k$ if for every edge from π_s to π_t under constraint ϕ in H:*

1. *If $\pi_t = \pi$, then H' contains an edge from π_s to π_j under constraint $\phi \wedge \phi_j$.*
2. *If $\pi_s = \pi$, then H' contains an edge from π_j to π_t under constraint ϕ.*
3. *If $\pi_s \neq \pi \wedge \pi_t \neq \pi$, then H' also contains an edge from π_s to π_t under ϕ.*

Intuitively, if H' is obtained from H by splitting location π to more precise abstract locations $\pi_1, \ldots, \pi_k$ under constraints $\phi_1, \ldots, \phi_k$, then any edge to π in H is replaced by a set of edges to any abstract location π_j under its respective constraint ϕ_j.

DEFINITION 9. *We say a heap H is at least as precise as another heap H' if either of the following two conditions are satisfied:*

1. *For all concrete locations l_c that can arise during the execution of a program, $\alpha_H(l_c) = \alpha_{H'}(l_c)$, and for every edge from π_s to π_t qualified by constraint $\langle \phi_{\text{may}}, \phi_{\text{must}} \rangle$ in H, there is an edge in H' from π_s to π_t qualified by $\langle \phi'_{\text{may}}, \phi'_{\text{must}} \rangle$ such that:*

$$\phi_{\text{may}} \Rightarrow \phi'_{\text{may}} \ \wedge \ \phi'_{\text{must}} \Rightarrow \phi_{\text{must}}$$

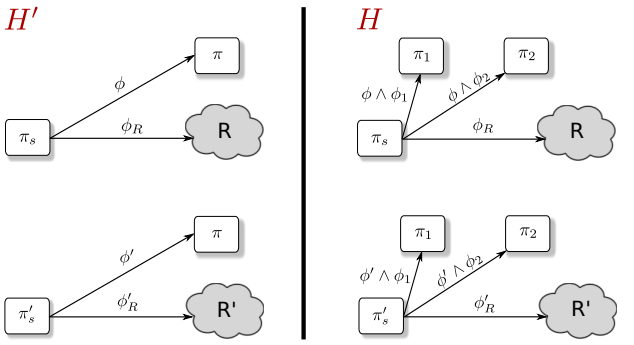

Figure 6. Heap H' and H from the proof.

2. *Otherwise, there must exist a concrete location l_c with $\alpha_{H'}(l_c) = \pi'$ and $\alpha_H(l_c) = \pi_0$ such that $\gamma(\pi_0) \subset \gamma(\pi')$, and there exists a set of abstract locations $\pi_1, \ldots \pi_k$ in H such that $\gamma(\pi') = \gamma(\pi_0) \cup \gamma(\pi_1) \ldots \cup \gamma(\pi_k)$. Furthermore, H must be at least as precise as H'_{split} where H'_{split} splits π' in H' into $\{\pi_0, \pi_1, \ldots \pi_k\}$ under constraints $\{\phi_1, \ldots, \phi_k\}$ such that for every edge e to π' under constraint ϕ' in H', there is an edge to π_j in H under constraint $\phi_j \wedge \phi'$.*

According to the first criterion in this definition, a heap H is at least as precise as H' if the abstract locations in the two heaps are the same and the over- and underapproximations encoded by the constraints in H are at least as "tight" as those in H'. The second condition in the definition states that if H and H' differ in at least one abstract location π, then H refines H' by replacing π with a set of abstract locations $\pi_1, \ldots, \pi_k$, each of which represent a portion of the concrete locations represented by π.

LEMMA 2. *Let H and H' be two sound symbolic heaps obtained from the same program such that H is at least as precise as H'. If H and H' enforce existence and uniqueness invariants, then any assertion provable under H' is also provable under H.*

(Sketch) If H is at least as precise as H', and for all l_c, $\alpha_H(l_c) = \alpha_{H'}(l_c)$, this lemma is easy to show. We consider the case where there exists some l_c such that $\gamma(\alpha_H(l_c)) \subset \gamma(\alpha_{H'}(l_c))$. For simplicity, we assume that there is exactly one abstract location π in H' that is now represented by two abstract locations π_1 and π_2 in H (if this is not the case, we can easily construct a sequence of more precise heaps from H' to H that have this property at each step). Consider an assertion of the form $\mathit{assert}(\mathit{read}(\pi_s, \gamma) = \mathit{read}(\pi'_s, \gamma'))$ that is provable in H'. Let $\mathit{read}(\pi_s, \phi_s) = \{\ldots, (\pi_i, \phi_i), \ldots\}$ and $\mathit{read}(\pi'_s, \phi'_s) = \{\ldots, (\pi'_i, \phi'_i), \ldots\}$ in heap H. Clearly, if there does not exist some π_i, π'_i such that $\pi = \pi_i$ or $\pi = \pi'_i$, then the assertion is also trivially provable in H. There are two cases to consider: (i) Only one of the read value sets contains π in H' or (ii) both of them contain π in H'. The first case is uninteresting since if π is in only one of the read

sets, π does not play a role in the validity of the assertion. Hence, we consider (ii).

In this case, heaps H and H' must differ in the way shown in Figure 6. In this figure, R and R' represent some set of abstract locations, and ϕ_R and ϕ'_R represent the disjunction of the constraints on the edges from π_s (resp. π'_s) to each location in R (resp. R'). (The constraints from π_s to R are the same in H and H' because all existing information is preserved, i.e., these constraints must be equivalent under the axioms from Section 4.2.)

To keep the proof understandable, we only consider the case where π does not contain index variables. Since both H and H' enforce existence and uniqueness of memory contents, we know:

$$\begin{array}{ll} \phi_R \wedge \phi = \mathit{false} & \phi'_R \wedge \phi' = \mathit{false} \\ \phi_R \vee \phi = \mathit{true} & \phi'_R \vee \phi' = \mathit{true} \\ \phi_R \wedge \phi \wedge \phi_1 = \mathit{false} & \phi'_R \wedge \phi' \wedge \phi_1 = \mathit{false} \\ \phi_R \wedge \phi \wedge \phi_2 = \mathit{false} & \phi'_R \wedge \phi' \wedge \phi_2 = \mathit{false} \end{array}$$
$$\phi \wedge \phi_1 \wedge \phi \wedge \phi_2 = \mathit{false}$$
$$\phi' \wedge \phi_1 \wedge \phi' \wedge \phi_2 = \mathit{false}$$
$$\phi_R \vee (\phi \wedge \phi_1) \vee (\phi \wedge \phi_2) = \mathit{true}$$
$$\phi'_R \vee (\phi' \wedge \phi_1) \vee (\phi' \wedge \phi_2) = \mathit{true}$$

These constraints imply $\phi \Leftrightarrow ((\phi \wedge \phi_1) \vee (\phi \wedge \phi_2))$ and $\phi' \Leftrightarrow ((\phi' \wedge \phi_1) \vee (\phi' \wedge \phi_2))$. Observe that this implies

$$\phi_1 \vee \phi_2 = \mathit{true} \quad (1)$$

Let I_s denote the index variables used in the source locations π_s and π'_s. For the assertion to be valid in H', we have:

$$\mathit{VALID}\left(\begin{array}{l} \exists I.\, \gamma \wedge \gamma' \wedge \\ (\pi = \pi \wedge \phi \wedge \phi') \vee \\ (\pi = R' \wedge \phi \wedge \phi'_R) \vee \\ (\pi = R \wedge \phi' \wedge \phi_R) \vee \\ (R = R' \wedge \phi_R \wedge \phi'_R) \end{array}\right)$$

Since we know that π is not in R or R', this formula is only valid if the following formula is also valid:

$$\mathit{VALID}\left(\begin{array}{l} \exists I.\, \gamma \wedge \gamma' \wedge \\ (\phi \wedge \phi') \vee (R = R' \wedge \phi_R \wedge \phi'_R \wedge \gamma \wedge \gamma') \end{array}\right) \quad (*)$$

Now, the validity of the assertion in H is checked using the formula:

$$\mathit{VALID}\left(\begin{array}{l} \exists I.\, \gamma \wedge \gamma' \wedge \\ (\pi_1 = \pi_1 \wedge \phi \wedge \phi_1 \wedge \phi' \wedge \phi_1) \vee \\ (\pi_2 = \pi_2 \wedge \phi \wedge \phi_2 \wedge \phi' \wedge \phi_2) \vee \\ (\pi_1 = \pi_2 \wedge \phi \wedge \phi_1 \wedge \phi' \wedge \phi_2) \vee \\ (\pi_2 = \pi_1 \wedge \phi \wedge \phi_2 \wedge \phi' \wedge \phi_1) \vee \\ (\pi_1 = R' \wedge \phi \wedge \phi_1 \wedge \phi'_R) \vee \\ (\pi_2 = R' \wedge \phi \wedge \phi_2 \wedge \phi'_R) \vee \\ (R = \pi_1 \wedge \phi_R \wedge \phi' \wedge \phi_1) \vee \\ (R = \pi_2 \wedge \phi_R \wedge \phi' \wedge \phi_2) \vee \\ (R = R' \wedge \phi_R \wedge \phi'_R) \end{array}\right)$$

Again, since π_1, π_2 are not in R or R' and π_1 and π_2 are distinct, this is equivalent to checking:

$$\mathit{VALID}\left(\begin{array}{l} \exists I.\, \gamma \wedge \gamma' \wedge \\ (\phi \wedge \phi_1 \wedge \phi') \vee \\ (\phi \wedge \phi_2 \wedge \phi') \vee \\ (R = R' \wedge \phi_R \wedge \phi'_R) \end{array}\right) \quad (**)$$

Now, observe that $(\phi \wedge \phi_1 \wedge \phi') \vee (\phi \wedge \phi_2 \wedge \phi') \Leftrightarrow \phi \wedge \phi' \wedge (\phi_1 \vee \phi_2) \Leftrightarrow \phi \wedge \phi'$, where the last equivalence follows from (1). Hence, the validity of $(*)$ implies the validity of $(**)$.

5. Implementation

We have implemented the ideas presented in this paper in our Compass verification framework for analyzing C programs. Compass supports most features of the C language, including structs, unions, multi-dimensional arrays, dynamic memory allocation, and pointer arithmetic. Compass does not assume type safety and handles casts soundly using a technique based on physical subtyping [15]. To check for buffer overruns, Compass tracks buffer and allocation sizes. Compass can be used for checking both user-provided assertions as well as many memory safety properties, such as null dereferences, buffer overruns and underruns, uninitialized reads, leaked stack allocations, and invalid casts. However, Compass currently does not check for integer overflows; hence, the safety of buffer accesses is predicated on the absence of integer overflows.

Compass performs flow-, path-, and context-sensitive program analysis. To achieve path-sensitivity, the constraints qualifying the edges in the symbolic heap abstraction not only qualify the source and the target's index variables, but can also mention constraints arising from path conditions. For interprocedural analysis, Compass performs a summary-based, context-sensitive analysis. For solving constraints, Compass utilizes a custom SMT solver called Mistral [16], which also provides support for on-line simplification of constraints [17].

6. Experimental Evaluation

To evaluate the precision and scalability of symbolic heap abstraction combined with axiomatization of memory invariants, we use Compass to check for memory safety properties (specifically, null dereferences, buffer overruns and underruns, and safety of casts) in OpenSSH 5.3p1 [18], totaling 26,615 lines of code. We believe OpenSSH to be a challenging and interesting target because it contains many complex array and pointer usage patterns, is heavily optimized for performance, is believed to be well-tested, and it is widely deployed.

The results of this experiment are presented in Figure 7. To quantify the relative importance of reasoning about heap contents and reasoning about memory invariants, we run our analysis in four different configurations: The first configura-

	Combined	Content Only	Mem-Inv Only	Smash
Time (s)	261	788	103	115
Max memory used (MB)	208	763	144	105
# reported buffer errors	2	77	117	371
# reported null errors	3	53	71	180
# reported cast errors	0	28	11	421
Total # of errors	5	158	199	972
Total # of false positives	1	154	195	968

Figure 7. Experimental results obtained on a single core of a 2.66 GHz Xeon CPU

	Lines	Combined				Content Only				Mem-Inv Only				Smash			
		Buffer	Null	Cast	Time	Buffer	Null	Cast	Time	Buffer	Null	Cast	Time	Buffer	Null	Cast	Time
hostname	304	0	0	0	0.14s	1	0	0	0.35s	3	1	0	0.35s	4	2	0	0.31s
chroot	371	0	0	0	0.15s	1	0	1	0.61s	2	1	0	0.60s	4	1	1	0.70s
rmdir	483	0	0	0	0.98s	2	0	0	1.39s	3	0	0	0.66s	3	1	0	0.51s
su	1047	0	0	0	1.63s	3	1	1	1.99s	2	1	1	1.62s	11	3	2	1.07s
mv	1151	0	0	0	0.79s	2	3	3	1.48s	1	1	2	1.01s	6	4	3	1.31s

Figure 8. False Positives by Category when selectively disabling memory invariants or reasoning about array contents, reported on five Unix Coreutils with running times. Experimental results obtained on a single core of a 2.66 GHz Xeon CPU

tion, called "Combined", employs the technique described in this paper, combining symbolic heap abstraction with demand-driven axiomatization of memory invariants. The second configuration, called "Content Only", tracks contents of memory locations, but it does not enforce existence and uniqueness of memory contents. The third configuration is "Mem-Inv Only", which enforces existence and uniqueness of concrete memory locations (i.e., introduces the Δ constraints from Section 4), but does not introduce the axioms described in Section 4. The fourth configuration is "Smash", which effectively smashes array contents by neither introducing memory invariants nor tracking the relationship between indices and contents. As described in Section 5, all configurations of the analysis are flow-, path- and context-sensitive.

As shown in the first column of Figure 7, using the technique proposed in this paper, Compass analyzes OpenSSH in ~ 4.4 minutes using no more than 208 MB of memory, finding one buffer overrun, one buffer underrun (unrelated to the first one), and three null dereference errors, one of which is a false positive. The only false positive reported by the analysis is due to an imprecise loop invariant, where the invariant generation aspect of our analysis cannot determine that an array element must be updated exactly once, rather than in multiple iterations, of a loop. In these experiments, we only annotated the relationship between `argv` and `argc` in `main` and provided suitable stubs for functions we did not analyze (e.g., system calls, OpenSSL functions called by OpenSSH). In addition, we had to annotate an invariant that relates two fields of a global data structure. We belive the statistics shown in the first (Combined) column of Figure 7 demonstrate that symbolic heap abstraction combined with demand-driven axiomatization is precise and scalable enough to verify memory safety properties in a real application with sufficiently useful precision.

In contrast, the analysis configuration (Content Only) that reasons about contents of arrays but that does not enforce memory invariants reports 154 false positives. It is interesting to observe that in addition to reporting significantly more false positives, the analysis also takes about three times as long as the first analysis configuration (Combined). This longer running time is explained by the fact that many constraints can be proven unsatisfiable by only taking memory invariants into account without needing extra information about the contents of memory locations. We believe the striking difference in precision between the first and second analysis configurations corroborates the hypothesis that reasoning about memory invariants is as important as reasoning about contents of memory locations.

We next consider the analysis configuration from Figure 7 that only enforces memory invariants but that does not track the relationship between indices and values. This configuration reports 195 false positives, confirming that precise reasoning about array contents is vital for successful verification of real-world applications. From the 154 and 195 false positives reported by the "Content Only" and "Mem-Inv Only" configurations, 56 error reports are shared. This observation indicates that at least 56 errors require combined reasoning about array contents as well as memory invariants and cannot be discharged by two separate analyses. The final configuration, which performs array smashing, reports 968 false positives, demonstrating that this level of precision is unlikely to be useful for verification of real-world applications.

We believe the reason that our analysis can scale to a program like OpenSSH with a few ten thousand lines of code while performing a very precise analysis of array and heap contents is that it avoids performing explicit case analyses in two important ways: First, by employing the axiomatization strategy described in this paper, our analysis can achieve

precise relational reasoning without explicitly considering different heap configurations. Second, by using the fluid update operation [4] for array updates, our technique avoids creating explicit partitions of arrays.

To demonstrate that other C programs also require reasoning about memory invariants in addition to heap contents, we also applied all four analysis configurations to five Unix Coreutils programs, ranging from 304 to 1151 lines of C code. While symbolic heap abstraction combined with axiomatization of memory invariants is powerful enough to prove the absence of buffer overruns, null dereferences, and casting errors with zero false positives in these programs, neither the "Content-Only" nor the "Mem-Inv Only" setting is able to prove all accesses are safe. As shown in Figure 8, the relative impact of reasoning about heap contents and memory invariants is roughly comparable, underscoring that reasoning about existence and uniqueness invariants is crucial for successful verification of real programs.

7. Related Work

There has been much interest in reasoning about the contents of arrays in the past decade; many of these techniques focus on generating invariants about array elements. Gopan et al. propose a 3-valued logic-based framework for reasoning about the contents of arrays [1]. In this work, array elements that share a common invariant are placed into a *partition*, and operations such as *focus* and *blur* are required to isolate and coalesce array elements. Jhala and McMillan adopt an approach similar to [1] using counterexample-guided abstraction refinement [2]. The approach presented in [5] also uses abstraction refinement for reasoning about array contents. Halbwachs and Peron propose an array content analysis based on abstract interpretation for a restricted class of so-called *simple programs* [3]. In this paper, in addition to precisely reasoning about array contents, we present a scalable technique that enforces existence and uniqueness invariants and achieves precise relational reasoning without performing explicit case splits.

Manevitch [19] proposes a heuristic to make TVLA-based analyses more scalable. To mitigate the state-space explosion that arises from analyzing the set of all possible heaps, he proposes *partial isomorphic heap abstraction*, which is a heuristic to merge two abstract heaps if they are *universe congruent*. While this technique considerably speeds up analysis on many benchmarks, it may lose information and is not as precise as analyzing all abstract heaps separately. Our technique reasons about only one abstract heap per program point, and achieves the same level of precision as creating multiple heaps by enforcing existence and uniqueness through constraints on points-to edges. This strategy effectively delays any disjunctive reasoning until constraint solving, and since a constraint solver typically does not need to analyze all cases to prove a constraint satisfiable or unsatisfiable, our approach appears to be more scalable without losing precision due to heuristic merging of abstract heaps.

An alternative to the graph-based heap representations considered in this paper is verification-condition generation based approaches for reasoning about heap contents (e.g., [20]). These approaches use combinations of various logics, such as the theory of arrays [10, 21–23] and pointer logic [24], to generate one large verification condition encoding all writes to and reads from the heap. Since these approaches encode the entire history of heap writes and reads in one formula (i.e., the verification condition), these techniques are able to establish relations and correlations between variables without requiring any extra machinery. In contrast, approaches based on per program-point heap representations such as [1, 4, 6], track the contents of the heap only at a given point in the program, and as a result, do not record a "history" of how this heap was established. For this reason, the latter approaches need extra tools to achieve precise relational reasoning but tend to be more scalable because they only encode the current state of the heap. The technique presented in this paper combines aspects of both approaches by allowing relational reasoning in a practical and scalable way and without requiring the history of updates to the heap. Effectively, our approach separates the task of reasoning about heap contents from answering queries about the heap, and we believe this separation is key to scaling our approach to a program as large as OpenSSH.

Work on array analysis from the parallel compiler work of the '80's and '90's also infers some aspects of the "memory invariant" in the form of may-dependences and dependence distances [25]. These techniques are targeted at a very different class of applications and are not as expressive as our approach.

We do not address the problem of reasoning about recursive pointer data structures. Techniques for reasoning about contents of recursive pointer data structures, such as lists and trees, include (but are not limited to) techniques based on *canonical abstraction* [6] and *separation logic* [8, 26]. We believe the techniques described in this paper can be extended to some recursive pointer data structures, such as lists; we leave this as future work.

8. Conclusion

We have presented a new and conceptually simple technique for enforcing correlations between abstract read operations on aggregate data structures: rather than splitting the abstract heap into multiple heaps so that the memory location of interest has a unique value in each individual heap, we enforce via constraints the existence and uniqueness of the value of every memory location. As a result, we are able to delay the cost of analyzing the possible values in the heap from the time when the heap representation corresponding to some program statement is first constructed to when we need to answer satisfiability and validity queries about a property of

the program. By delaying the cost we often avoid having to pay it at all, as in many cases queries can be answered by a solver without a full enumeration of all possibilities. We have also shown that this improved trade-off in theory actually pays off in practice: our implementation is able to analyze medium-sized program such as OpenSSH precisely enough to fully verify memory safety, even in the presence of intricate array and pointer operations.

9. Acknowledgments

We would like to thank Mooly Sagiv for several very useful discussions and comments on an earlier draft of this paper.

References

[1] Gopan, D., Reps, T., Sagiv, M.: A Framework for Numeric Analysis of Array Operations. In: POPL, NY, USA, ACM (2005) 338–350

[2] Jhala, R., Mcmillan, K.L.: Array Abstractions from Proofs. In: CAV. (2007)

[3] Halbwachs, N., Péron, M.: Discovering Properties about Arrays in Simple Programs. In: PLDI, NY, USA, ACM (2008) 339–348

[4] Dillig, I., Dillig, T., Aiken, A.: Fluid Updates: Beyond Strong vs. Weak Updates. In: ESOP. (2010) 246–266

[5] Seghir, M., Podelski, A., Wies, T.: Abstraction Refinement for Quantified Array Assertions. In: SAS, Springer-Verlag (2009)

[6] Reps, T.W., Sagiv, S., Wilhelm, R.: Static Program Analysis via 3-valued Logic. In: CAV. Volume 3114 of Lecture Notes in Comp. Sc., Springer (2004) 15–30

[7] Distefano, D., O Hearn, P., Yang, H.: A Local Shape Analysis Based on Separation Logic. Lecture Notes in Comp. Sc. **3920** (2006) 287

[8] Yang, H., Lee, O., Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.: Scalable Shape Analysis for Systems Code. CAV (2008) 385–398

[9] Bogudlov, I., Lev-Ami, T., Reps, T., Sagiv, M.: Revamping TVLA: Making Parametric Shape Analysis Competitive. Lecture Notes in Computer Science **4590** (2007) 221

[10] McCarthy, J.: Towards a Mathematical Science of Computation. In: IFIP. (1962)

[11] Dillig, I., Dillig, T., Aiken, A.: Sound, Complete and Scalable Path-sensitive Analysis. In: PLDI, ACM (2008) 270–280

[12] Landi, W., Ryder, B.G.: A Safe Approximate Algorithm for Interprocedural Aliasing. SIGPLAN Not. **27**(7) (1992) 235–248

[13] Gulwani, S., Musuvathi, M.: Cover Algorithms and Their Combination. In: ESOP. (2008) 193–207

[14] Ge, Y., de Moura, L.: Complete Instantiation for Quantified Formulas in Satisfiabiliby Modulo Theories. In: CAV, Springer (2009) 320

[15] Chandra, S., Reps, T.: Physical Type Checking for C. In: PASTE **24**(5) (1999) 66–75

[16] Dillig, I., Dillig, T., Aiken, A.: Cuts from proofs: A complete and practical technique for solving linear inequalities over integers. In: In CAV, Springer (2009)

[17] Dillig, I., Dillig, T., Aiken, A.: Small Formulas for Large Programs: On-line Constraint Simplification in Scalable Static Analysis. In: SAS. (2010)

[18] http://www.openssh.com/: Openssh 5.3p1

[19] Manevich, R.: Partially Disjunctive Shape Analysis. PhD thesis, Tel Aviv University (2009)

[20] Lahiri, S., Qadeer, S.: Verifying Properties of Well-founded Linked Lists. In: Proceedings of the Symposium on Principles of Programming Languages. (2006) 115–126

[21] Bradley, A., Manna, Z., Sipma, H.: What's Decidable About Arrays? Lecture notes in computer science **3855** (2006) 427

[22] Stump, A., Barrett, C., Dill, D., Levitt, J.: A Decision Procedure for an Extensional Theory of Arrays. In: IEEE Symposium on Logic in Computer Science. (2001) 29–37

[23] Habermehl, P., Iosif, R., Vojnar, T.: What Else is Decidable about Integer Arrays? Lecture Notes in Computer Science **4962** (2008) 474

[24] Kroening, D., Strichman, O.: Decision Procedures: An Algorithmic Point of View. Springer-Verlag New York Inc (2008)

[25] Pugh, W.: The Omega Test: A Fast and Practical Integer Programming Algorithm for Dependence Analysis. In: ACM Conference on Supercomputing. (1991) 4–13

[26] Reynolds, J.: Separation logic: A Logic for Shared Mutable Data Structures. In: 17th Annual IEEE Symposium on Logic in Computer Science. (2002) 55–74