PRECISE AND AUTOMATIC VERIFICATION OF CONTAINER-MANIPULATING PROGRAMS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Işıl Dillig
November 2011

Abstract

A key challenge for program verification systems is precise reasoning about the contents of unbounded data structures. A particularly important and widely-used family of data structures is *containers*, which support operations for inserting, retrieving, removing, and iterating over sets of elements. Typical examples of containers include arrays, lists, vectors, maps, sets, deques, queues, etc.

In this thesis, we propose a novel static analysis technique that allows precise reasoning for container-manipulating programs. We describe a symbolic heap abstraction which integrates reasoning about containers directly into a heap analysis, allowing the technique to precisely track heap objects as they flow in and out of containers. The proposed analysis is fully automatic and practical, scaling to real programs of up to 100,000 lines of code. Furthermore, the abstraction we propose can reason about key-value correlations and supports arbitrary nestings of container data structures.

More specifically, in this thesis, we first describe a static analysis for a language with only the most basic kind of container, namely arrays. We then refine this basic analysis to obtain a more precise heap abstraction, which is always guaranteed to be relational. We then generalize the framework used for reasoning about arrays to general-purpose containers, which may not expose a notion of position.

The symbolic heap abstraction we describe reduces a large part of the difficulty of reasoning about containers to linear inequalities over integers. Thus, the efficiency of our static analysis is contingent upon practical techniques for solving systems of linear integer inequalities. Another contribution of this thesis is a new and practical algorithm called *Cuts-from-Proofs* for solving linear inequalities over integers.

Acknowledgements

First of all, I would like to thank my advisor Alex Aiken for the incredible guidance, mentorship, and support he provided not only during my PhD career but also during my undergraduate years at Stanford. Alex introduced me to my current research area of program analysis and verification and taught me how to be a good researcher. He was always there for support whenever I needed some fresh insights about my research or ideas about how to improve a paper or a talk. He was also an indispensable source of encouragement during times when my papers kept getting rejected and my hard work did not seem to pay off. Perhaps most importantly, Alex has always been an exemplary mentor and a role model, not only teaching me how to do good research, but also showing by example how to be a supportive and caring advisor. I truly feel privileged and lucky to be his student, and I would be honored if I could do half as good a job as him in advising my own students in the future.

I am also extremely grateful to David Dill for his guidance as my undergraduate advisor and for his continued mentorship throughout my PhD career. My discussions with him about research have been extremely enlightening, and I am very grateful for his time and effort in patiently answering my various questions over the years and providing feedback on paper drafts and talks. I also owe him a special thanks for his support and useful advice in my recent job search process.

I would also like to thank Mooly Sagiv for his mentorship during the years he was on sabbatical at Stanford. It was a pleasure having him at Stanford for a year, and I really enjoyed coauthoring a paper with him and having many enlightening research-related discussions. I also thank Mooly for serving on my thesis reading committee and for his guidance in the job search process.

I am also very grateful to Henny Sipma for her continued support throughout the years. As an undergraduate, I learned so much from her program analysis class, and I really appreciate all her guidance and encouragement during the early years of my PhD. I also thank her for serving on my defense committee.

In addition, I would like to thank Martin Rinard, Tom Ball, Eran Yahav, Satish Chandra, Zohar Manna, and Dawson Engler. I have learned a lot from both Martin Rinard and Tom Ball over our discussions at conferences and other venues, and I am extremely grateful to them for their help in my job search process. It was a pleasure working with Eran Yahav and Satish Chandra during my internship at IBM, and I am grateful for their continued friendship and guidance. I also thank Zohar Manna for the wonderful classes he taught at Stanford, and I feel fortunate to have met him before his retirement. I also thank Dawson Engler for serving on my thesis defense committee.

I also thank my fellow PhD students at Stanford for their helpful comments and feedback on various paper drafts and talks over the years. In particular, I am very grateful to Philip Guo, Suhabe Bugrara, Peter Hawkins, Adam Oliner, Rahul Sharma, Brian Hackett, Eric Smith, Yichen Xie, Mayur Naik, Sorav Bansal, Mike Bauer, Eric Schkufza, Marc Schaub, and Robert Ikeda for their help on various occasions.

I thank all my friends and family for their unconditional support and encouragement throughout my PhD. My parents have *always* been there for me whenever I needed their help, and I am extremely grateful for having such wonderful and caring parents. My grandmother, who recently lost the difficult battle against cancer and whom I deeply miss, not only raised me as a child but also helped me become the person I am today. Among my friends, I would like to give special thanks to Ana Gardea, Aurelie Beaumel, and David Craig for giving friendship a whole new meaning.

Last but not least, I thank my husband, Tom, for his unconditional love and friendship. Without Tom, doing research would not be nearly as fun and rewarding as it has been, and I feel very fortunate to have a husband who is not only my best friend but also my favorite research collaborator.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

"If builders built buildings the way programmers write programs, then the first woodpecker that came along would destroy civilization." –Gerald M. Weinberg

1.1 Motivation

Today, as a civilization, we are becoming increasingly dependent on software: Every check we deposit, every business transaction we make, every trip on the subway, and even every phone call we make increasingly depends on the correct behavior of many layers of complex software. Furthermore, this trend is here to stay: In the near future, we will not be able to drive our cars, get medical treatment, or even heat our houses without being at the mercy of the alleged correctness of a web of computer software.

Unfortunately, despite this crucial reliance on software applications in every aspect of our daily lives, most software applications deployed today remain buggy, unreliable, and prone to crashes and security exploits. For example, just in the year 2010, various software errors were responsible for brake failures in Toyota Prius [12], for the removal of wrong organs from 25 donors in the UK [15], and for the double-charging of many customers on black Friday [13].

As these examples illustrate, software errors have very serious consequences in our daily lives, but standard techniques for enforcing software quality, such as testing,

do not guarantee the absence of catastrophic software errors. In contrast to more conventional testing and bug finding approaches, the goal of *software verification* is to discover and eradicate all potential errors of a given kind from software systems.

While proving the absence of certain kinds of software errors is a desirable goal, it is well known that deciding any non-trivial property of an arbitrary program written in a Turing-complete language is impossible. Thus, to guarantee the termination of the verification algorithm for all input programs, sound program analyses *over-approximate* the behavior of the input program rather than constructing an exact representation of the set of program states. Unfortunately, this overapproximation comes at a cost: Although the error states discovered by the analysis are always inside the overapproximation, they may be outside the actual feasible states of the program. In this case, the program analysis tool reports a so-called *false alarm* or *false positive*, which is a spurious report generated by the analysis that does not correspond to an actual error in the original program.

Thus, the key challenge in software verification is to construct *abstractions* (i.e., overapproximations) of the program that are sufficiently precise so as to minimize the number of false alarms generated by the tool. Furthermore, this abstraction must be practical enough to scale to programs of realistic size and should not consume too much of the programmer's time, for example by requiring many cumbersome annotations. Finally, the error reports generated by the tool should be accessible to programmers so that users of the analysis can easily understand and diagnose these error reports.

1.2 The Challenge

A particularly big challenge in program verification arises from the difficulty of statically reasoning about contents of heap data structures: Since the size of data structures such as dynamically allocated arrays, lists, maps etc. are unknown at compile time, program analysis techniques typically make gross over-approximations involving these unbounded data structures. For example, a common approach is to represent all elements in an array using one *abstract memory location* called a *summary node*

such that modifications to a particular array element contaminate the information available to the analysis about all the *other* array elements. The result of this coarse abstraction is a conservative, but very imprecise analysis that is unable to reason about individual elements of arrays or other unbounded data structures.

This thesis addresses the problem of precise and automatic static reasoning about an important class of commonly-used data structures known as *containers*, which support operations such as inserting, retrieving, removing, and iterating over a set of elements. Typical examples of containers include arrays, vectors, lists, maps, stacks, queues, sets, multimaps, and so on.

Precise reasoning about containers is, in practice, very important because these data structures are used extremely widely. For example, some containers such as arrays are often built-in language constructs, and other containers like maps and vectors are often provided as part of standard libraries, such as the Standard Template Library in C++ or the Collection libraries in Java.

Furthermore, the correctness argument of many program properties relies on a fairly detailed understanding of container elements. For instance, to prove that a value read from a map with a certain key k is valid (i.e., non-null), we must know that the key k was previously inserted into the map. Similarly, to prove the safety of a code fragment that iterates over a list and deletes all elements, we must know that no elements in this list alias each other. In some cases, even control flow in a program is determined by container elements: In C code, function pointers are frequently stored in arrays, making a detailed understanding of array contents a prerequisite for constructing a sufficiently precise call graph.

No previous fully-automatic and scalable technique for heap analysis is able to precisely reason about container elements. In particular, all practical techniques for this purpose treat container data structures as a bag of values and are unable to reason about key-value correlations. The consequence of this coarse abstraction is that modifying one element in the container pollutes the analysis information for all other elements. Thus, in realistic programs which make heavy use of containers, standard analysis techniques are woefully inadequate for verification purposes.

1.3 Contributions

In this thesis, we describe the first approach for precise, practical, and fully-automatic reasoning about container data structures. Our static analysis seamlessly integrates detailed, per-element reasoning about containers into a *heap analysis* for answering queries about contents of memory locations. Our technique is capable of tracking position-value and key-value correlations in data structures and allows precise reasoning about arbitrary nestings of containers. Furthermore, our technique is practical enough to be applied for the verification of memory safety properties in real, commonly used programs that manipulate heap objects through containers.

The main idea underlying our technique is to construct a *symbolic heap abstraction* that combines a graph-based representation of heap objects with logical formulas on points-to edges. These constraints qualifying points-to edges allow our technique to express correlations between keys and values and reduces much of the difficulty of data structure reasoning to a combination of standard logic operations and integer constraints.

More specifically, the rest of this thesis is organized as follows:

- Chapter 2 surveys the standard graph-based heap abstraction.
- Chapter 3 gives an overview of our symbolic heap representation.
- Chapter 4 presents a static analysis for a language with the most basic kind of container, namely arrays.
- Chapter 5 explains how the technique from Chapter 4 can be extended to enforce important memory invariants, which is necessary for a fully-relational abstraction.
- Chapter 6 generalizes the analysis described so far to general containers, such as maps, stacks, and sets.
- Chapter 7 surveys existing approaches for reasoning about arrays and heap data structures.

- Chapter 8 presents a new algorithm for solving linear inequalities over integers. Since our analysis partly reduces heap reasoning to constraints involving integer inequalities, a practical algorithm for solving these constraints greatly benefits the analysis.
- Finally, Chapter 9 concludes.

Chapter 2

Background

The most standard abstraction for reasoning about heap contents is a *may points-to graph*. A *may points-to graph* is a directed graph where nodes correspond to *abstract memory locations*, which represent one or more concrete (run-time) locations, and an edge from one node A to another node B in this graph indicates that any concrete location represented by A *may* point to any concrete location represented by node B.

To illustrate the standard heap representation, consider the following code snippet, written in C-like syntax:

```
int** foo(int** b, int size)
{
  int** a = malloc(sizeof(int*) * size);
  for(int i=0; i<size; i++)
  {
    a[i] = b[i];
  }
  return a;
}
```

In this code snippet, `b`, which is an input to the program, points to an array of integer arrays. The code snippet in `foo` dynamically allocates a new array pointed to by variable `a`, and after the loop, each `a[i]` points to the same elements pointed to by `b[i]`.

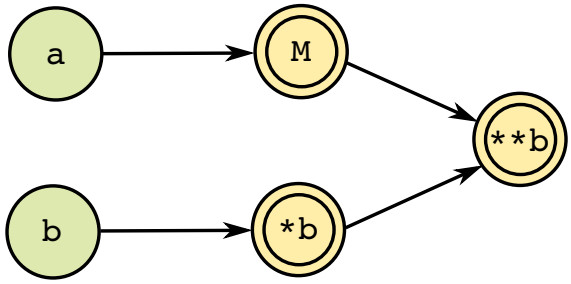


Figure 2.1: An example may points-to graph

Figure 2.1 shows an abstraction describing the state of the heap at the end of function `foo`. The abstract memory locations labeled `a` and `b` represent the locations of program variables `a` and `b`. The abstract location labeled `*b` represents the array of integer arrays pointed to by `b`, and `**b` abstracts *all* integer arrays pointed to by any `b[i]`. Finally, the node labeled `M` models all elements of the heap allocated array in function `foo`.

In this standard abstraction, observe that there is a distinction between two classes of abstract memory locations: Nodes indicated by double circles are *summary locations*, which represent multiple concrete locations, and nodes indicated by single circles are *non-summary* locations, representing *exactly one* concrete location. For instance, in Figure 2.1, the node labeled `M` is a summary node because it represents *all* concrete elements of the dynamically allocated array. On the other hand, the node labeled `a` is a non-summary location as it corresponds to one single concrete memory location.

Unfortunately, this standard abstraction has two important drawbacks for representing container data structures, such as arrays. First of all, since a points-to edge from some summary node A to another summary node B indicates that any concrete location in A may point to any concrete location in B, an edge in this representation corresponds to a *full cross-product* between the run-time locations modeled by the summary nodes. As a result, this abstraction cannot precisely encode correlations between the indices involved in the points-to relations. For instance, in our example, the standard abstraction cannot express that the i'th element in the heap allocated

array points to the same location as the i'th element in array `b`. Instead, the abstraction shown in Figure 2.1 encodes that the i'th element in array `a` may point to the same location as *any* element in array `b`.

The second drawback of the standard representation is that it requires making a distinction between two kinds of updates to abstract memory locations when performing a *flow-sensitive analysis*, i.e., an analysis that is sensitive to the order of statements in the program. Specifically, a *strong update* to an abstract memory location A removes all existing points-to edges outgoing from A before establishing a new points-to relation. On the other hand, a *weak update* preserves existing points-to edges from A but also adds new ones. Whenever safe, it is preferable to apply strong updates in order to increase the precision of the analysis.

Unfortunately, applying strong updates to an abstract memory location A requires that A be a non-summary location. If A is a summary location, we cannot apply a strong update to A, because modifying one concrete element represented by A does not affect the other concrete elements also abstracted by A. As a result, existing analyses deal with this difficulty in one of two ways: (i) They either allow only weak updates to summary locations, or (ii) more sophisticated techniques, such as 3-valued logic analysis [73], first isolate individual elements of an unbounded data structure via a *focus* operation to apply a strong update, and the isolated element is folded back into the summary location via a dual *blur* operation to avoid creating an unbounded number of abstract memory locations. While the latter approach allows precise reasoning about unbounded data structures, finding the right focus and blur strategies is challenging and hard to automate without the aid of user-provided instrumentation predicates [73].

Chapter 3

Overview of the Symbolic Heap

In this chapter, we give an overview of our heap representation, called the *symbolic heap abstraction*, that overcomes the main problems of the standard heap representation (described in Chapter 2) for reasoning about container data structures.

3.1 Key Ideas

Similar to the heap representation surveyed in Chapter 2, the symbolic heap abstraction is a directed graph where nodes correspond to abstract memory locations and a directed edge from one node A to another node B indicates that concrete elements modeled by A may point to concrete elements modeled by B. However, our symbolic heap abstraction differs from the standard heap representation in three important ways to facilitate reasoning about container data structures.

The first key idea that underlies our approach is that container data structures are modeled using *indexed locations* rather than as summary locations described in Chapter 2. More specifically, any abstract memory location associated with a container is labeled $\langle l \rangle_i$ where the subscript i is an *index variable* that ranges over the indices in the container. While an indexed location $\langle l \rangle_i$ abstracts *all* the elements in the container, the unique index variable i associated with this location allows us to select particular subsets of concrete elements by specifying constraints on i. For example, if an array `a` is modeled using an indexed location $\langle a \rangle_i$, then the constraint $i = 1$ selects

only the second element in the array. Similarly, the constraint $0 \leq i < \mathit{size}$ allows us to refer to all the elements of the array in the range $[0, \mathit{size})$.

The second important distinction between the symbolic heap and the standard heap abstraction is that points-to edges in this graph are qualified by constraints on the index variables associated with the source and target nodes. These constraints allow us to specify which elements in the source container point to which elements in the target container. For example, consider two arrays `a` and `b` modeled by the indexed locations $\langle a \rangle_{i_1}$ and $\langle b \rangle_{i_2}$. If an edge from $\langle a \rangle_{i_1}$ to $\langle b \rangle_{i_2}$ is qualified by the constraint $i_1 = i_2 \wedge 0 \leq i_1 < \mathit{size}$, then this symbolic heap fragment expresses that all elements of array `a` whose indices are in the range $[0, \mathit{size})$ point to an element with the same corresponding index in array `b`.

The third idea behind the symbolic heap analysis is that it overcomes the dichotomy between the traditional weak and strong updates to abstract memory locations described in Chapter 2. More specifically, points-to relations in the symbolic heap are modified by what we call a *fluid update* operation, which applies to any kind of abstract memory location, regardless of how many concrete locations this node represents. For example, consider a store operation `a[2] = c`, where array `a` is modeled by the node $\langle a \rangle_i$. To statically analyze a store operation to the container, we first compute a constraint ϕ_{index} on i that specifies which elements in the container are modified by the store. In our simple example, this index constraint is $i = 2$, expressing that only the third element in the array is modified. Once this constraint ϕ_{index} is computed, a fluid update operation then adds an edge from the node modeling the container to its new points-to target (in our example, a node modeling variable `c`) under this index constraint ϕ_{index}. Finally, to encode that those elements that were modified by the store operation no longer point to their old points-to targets, the fluid update operation conjoins all existing points-to edges outgoing from the container with $\neg\phi_{\mathit{index}}$. The symbolic heap after the fluid update now reflects that only those elements that satisfy ϕ_{index} are modified, whereas all other elements (i.e., those that satisfy $\neg\phi_{\mathit{index}}$) remain unchanged.

Unfortunately, in the general case, we may not know the exact subset of the concrete elements in the container that are modified by a store operation. For example,

consider the statement `a[complicated()%5] = c`, where `complicated` is a very complex function whose return value the static analysis cannot understand. In this case, we cannot write a constraint that *exactly* describes the only element in the array that is modified by the statement. Thus, in the general case, the constraint ϕ_{index} that we compute is not an exact description of the modified elements, but is instead an *overapproximation*. In our example, although we cannot give an exact description of the one element that is modified, we can nonetheless overapproximate those elements that *may* be modified using the constraint $0 \leq i \leq 4$, since any number modulo 5 must be in the range $[0, 4]$.

Now, recall that the fluid update operation sketched out above requires us to compute the negation of the constraint ϕ_{index} in order to describe those elements in the container that are *not* modified. However, if ϕ_{index} is an overapproximation of those container elements that are modified, then $\neg\phi_{index}$ is an *underapproximation* of the container elements that are *not* modified. This is problematic because, for our heap abstraction to be sound, we also need to be able to overapproximate container elements that are not modified by store operations.

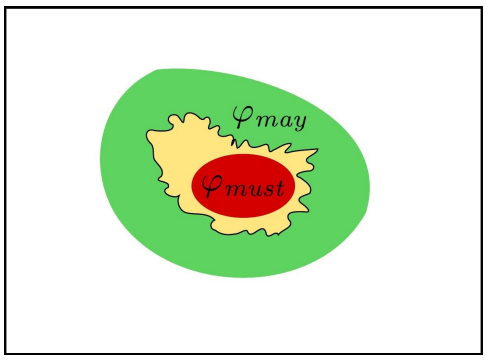
To deal with the difficulties that arise from combining the negation operation with approximations, the constraints we use in the symbolic heap are a special kind of constraints that we call *bracketing constraints*. More specifically, bracketing constraints are pairs of constraints of the form $\langle\phi_{may}, \phi_{must}\rangle$ where ϕ_{may} corresponds to an overapproximation and ϕ_{must} is an underapproximation.

The crucial property of bracketing constraints is that they preserve over- and underapproximations under the following negation operation:

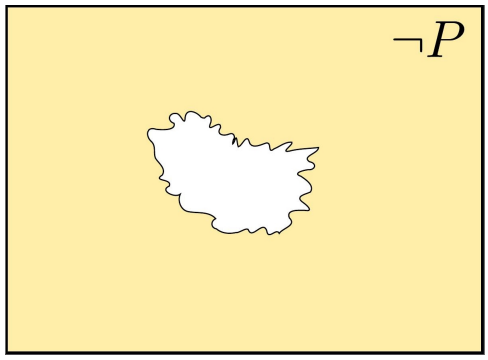
$$\neg\langle\phi_{may}, \phi_{must}\rangle \;=\; \langle\neg\phi_{must}, \neg\phi_{may}\rangle$$

In other words, if ϕ_{may} and ϕ_{must} are valid over- and underapproximations of some property P, then $\neg\phi_{must}$ and $\neg\phi_{may}$ are correct over- and underapproximations of the property $\neg P$. Figure 3.1 clarifies this discussion pictorially using Venn diagrams. Since bracketing constraints are closed under all the boolean connectives including negations, all the constraints that are used in the symbolic heap representation and

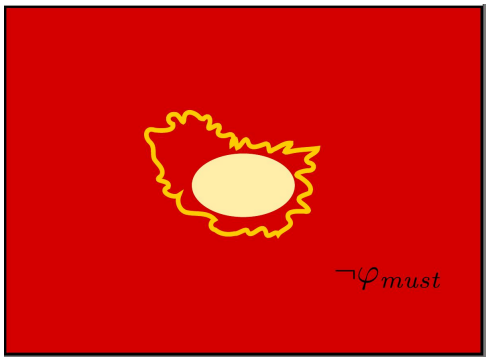
in the fluid update operation are bracketing constraints. Furthermore, since the fluid update operation relies on performing negation in a sound way, the use of bracketing constraints is crucial for the correctness of our approach.



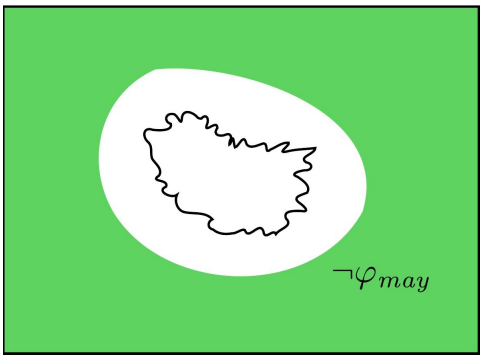
(a) Yellow region exact description of P, green region overapproximation, red region underapproximation



(b) Yellow region exact description of $\neg P$



(c) Taking the complement (negation) of underapproximation of P yields overapproximation of $\neg P$



(d) Taking the complement (negation) of overapproximation of P yields underapproximation of $\neg P$

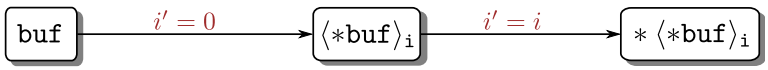
Figure 3.1: Complements of over- and underapproximations

3.2 Example

To illustrate the key ideas discussed so far, we now consider the following example program fragment written in C-like syntax:

```
void send_packets(struct packet** buf, int c, int size) {
  assert(2*c <= size);
  for(int j=0; j< 2*c; j+=2) {
    if(transmit_packet(buf[j]) == SUCCESS) {
      free(buf[j]); buf[j] = NULL;
    }
   }
}
```

The function `send_packets` takes an array `buf` of `packet`$*$'s, an integer `c` representing the number of high-priority packets to be sent, and an integer `size`, denoting the number of elements in `buf`. All even indices in `buf` correspond to high-priority packets whereas all odd indices are low-priority.[1] This function submits one high-priority packet at a time; if the transfer is successful (which may depend on network traffic), it sets the corresponding element in `buf` to `NULL` to indicate the packet has been processed.



The figure above shows the symbolic heap representation at the entry of `send_packets`. Here, the dereference of variable `buf` is an array, hence, it is qualified by an *index variable* i, and the location labeled $\langle *\texttt{buf} \rangle_{\texttt{i}}$ represents all elements of array $*\texttt{buf}$. We use the convention that primed index variables on an edge qualify the edge's target, and unprimed index variables qualify the source. If the over- and underapproximations on an edge are the same, we write a single constraint instead of a pair.

In the drawing, the edge from `buf` to $\langle *\texttt{buf} \rangle_{\texttt{i}}$ is qualified by $i' = 0$ because `buf` points to the first element of the array $\langle *\texttt{buf} \rangle_{\texttt{i}}$. The constraint $i = i'$ on the edge from $\langle *\texttt{buf} \rangle_{\texttt{i}}$ to $*\langle *\texttt{buf} \rangle_{\texttt{i}}$ indicates that the i'th element of array $*\texttt{buf}$ points to some corresponding target called $*\langle *\texttt{buf} \rangle_{\texttt{i}}$.

The concrete elements modified by the statement `buf[j] = NULL` cannot be specified exactly at analysis time since the success of `transmit_packet` depends on an

[1] The distinction between even and odd-numbered elements in a network buffer arises in many real network applications, for example in packet scheduling [65] and p2p video streaming [70] .

environment choice (i.e., network state). The loop may, but does not have to, set all even elements between 0 and $2c$ to `NULL`. Hence, the best over-approximation of the indices of $*$`buf` modified by this statement is $0 \leq i < 2c \ \wedge \ i \bmod 2 = 0$. On the other hand, the best underapproximation of the set of indices updated in the loop is the empty set (indicated by the constraint *false*) since no element is guaranteed to be updated by the statement `buf[j] = NULL`.

Figure 3.2 shows the symbolic heap representation at the end of `send_packets`. Since the set of concrete elements that may be updated by `buf[j] = NULL` is given by $\langle 0 \leq i < 2c \wedge i \bmod 2 = 0, \ \mathit{false} \rangle$, the fluid update adds an edge from $\langle *\texttt{buf} \rangle_{\texttt{i}}$ to $*\texttt{NULL}$ under this bracketing constraint. The existing edge from $\langle *\texttt{buf} \rangle_{\texttt{i}}$ to $*\langle *\texttt{buf} \rangle_{\texttt{i}}$ is preserved under $\neg\langle 0 \leq i < 2c \wedge i \bmod 2 = 0, \ \mathit{false} \rangle$. Thus, assuming $i \geq 0$, this is equivalent to $\langle \mathit{true}, \ i \geq 2c \vee i \bmod 2 \neq 0 \rangle$. Since the initial constraint on the edge stipulates $i = i'$, the edge constraint after the fluid update becomes $\langle i = i', \ (i \geq 2c \vee i \bmod 2 \neq 0) \wedge i = i' \rangle$. The new edge condition correctly and precisely states that any element of $*$`buf` may still point to its original target when the function exits, but only those elements whose index satisfies the constraint $i \geq 2c$ or $i \bmod 2 \neq 0$ *must* point to their original target.

As this example illustrates, our approach has the following salient characteristics:

- The combination of indexed location and constraints on points-to edges allows our abstraction to reason about position-value correlations in arrays and other containers.

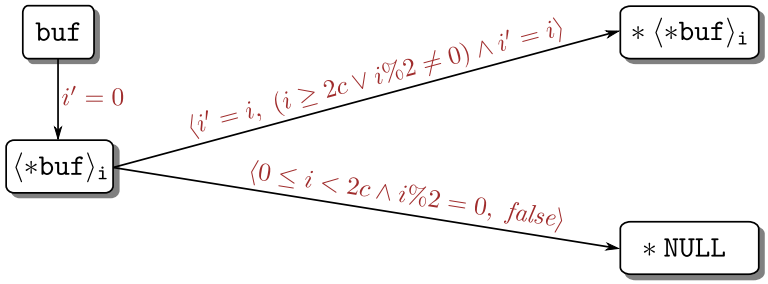


Figure 3.2: The points-to graph at the end of function `send_packets`

- Our technique does not require concretizing individual elements of containers to statically analyze updates in a precise way, making operations such as focus and blur [73] unnecessary.
- Our technique never constructs explicit partitions of containers, making this approach less vulnerable to the kind of state space explosion problem that other precise approaches, such as [73], are prone to.
- Our analysis preserves partial information despite imprecision and uncertainty. In the above example, although the result of `transmit_packet` is unknown, the analysis can still determine that no odd packet is set to `NULL`.

Chapter 4

Analysis of a Language with Arrays

In this chapter, we describe our full static analysis algorithm for a small imperative C-like language with arrays, pointers, and pointer arithmetic. This language is defined by the following grammar:

$$
\begin{array}{lcl}
\textit{Program } P & := & F^+ \\
\textit{Function } F & := & \textit{define } f(v_1, \ldots, v_n) = S \\
\textit{Statement } S & := & S_1; S_2 \mid v_1 = v_2 \mid v_1 = c \mid v_1 = \textit{alloc}(v_2) \mid v_1 = v_2[v_3] \mid v_2[v_3] = v_1 \\
 & & \mid v_1 = v_2 \oplus v_3 \mid v_1 = v_2 \text{ intop } v_3 \mid v_1 = v_2 \text{ predop } v_3 \mid \\
 & & \textit{if } v \neq 0 \textit{ then } S_1 \textit{ else } S_2 \mid \textit{while } v \neq 0 \textit{ do } S \textit{ end}
\end{array}
$$

In this grammar, v is a variable, and c is an integer constant. Types are defined by the grammar:

$$\tau := \text{int} \mid \text{pointer}(\text{array}(\tau))$$

Load ($v_1 = v_2[v_3]$) and store ($v_2[v_3] = v_1$) statements are defined on pointers v_2 and integers v_3, and we assume programs are well-typed. The expression $v[i]$ first dereferences v and then selects the i'th element of the array pointed to by v. Pointer arithmetic $v_1 = v_2 \oplus v_3$ makes v_1 point to offset v_3 in the array pointed to by v_2. Integer operations (intop) include $+, -$, and $\times$. Predicate operators (predop) are $=, \neq$ and $<$, and predicates evaluate to 0 (false) or 1 (true). The $\textit{alloc}(v_2)$ statement allocates an array with v_2 elements.

4.1 Operational Semantics

Figure 4.1 present an operational semantics of our language with arrays. In the operational semantics, a concrete memory location l_c is a pair (s, i) where s is a start address for a block of memory and i is an offset from s. For scalars, we use the notation $(v, \cdot)$ to indicate that the value stored in l_c is v and that the offset is not relevant. The environment E: $\texttt{Var} \rightarrow l_c$ maps variables to concrete locations, and the store S: $l_c \rightarrow l_c$ maps locations to other locations. The notation $S' = S[l \leftarrow \mathfrak{e}]$ denotes that store S' is identical to store S except that it maps location l to $\mathfrak{e}$. The function $\mathit{newloc}(S, c)$ returns the start address of freshly allocated memory containing c cells such that no cell overlaps existing memory cells.

4.1.1 Constraint Language

The constraints used in the analysis are defined by:

$$
\begin{array}{lcl}
\textit{Term } T & := & c \mid v \mid T_1 \text{ intop } T_2 \mid \mathit{select}(T_1, T_2) \mid \mathit{deref}(T) \\
\textit{Literal } L & := & \mathit{true} \mid \mathit{false} \mid T_1 \text{ predop } T_2 \mid T \bmod c = 0 \\
\textit{Atom } A & := & L \mid \neg A \mid A_1 \wedge A_2 \mid A_1 \vee A_2 \\
\textit{Constraint } C & := & \langle A_{NC}, A_{SC} \rangle
\end{array}
$$

Terms are constants, variables, arithmetic terms, and the uninterpreted function terms $\mathit{select}(T_1, T_2)$, which represents the result of selecting element at index T_2 of array T_1, and $\mathit{deref}(T)$, which represents the result of dereferencing T.

Literals are *true*, *false*, comparisons $(=, \neq, <)$ between two terms, and divisibility checks on terms. Atomic constraints A are arbitrary boolean combinations of literals. Satisfiability and validity of atomic constraints are decided over the combined theory of uninterpreted functions and linear integer arithmetic extended with divisibility predicates. Bracketing constraints C are pairs of atomic constraints of the form $\langle A_{NC}, A_{SC} \rangle$ representing necessary and sufficient conditions for some fact. A bracketing constraint is well-formed if and only if $A_{SC} \Rightarrow A_{NC}$. We write $\lceil \phi \rceil$ to denote the necessary condition of a bracketing constraint ϕ and $\lfloor \phi \rfloor$ to denote the sufficient condition of ϕ.

Sequence

$$\frac{E, S \vdash s_1 : S' \quad E, S' \vdash s_2 : S''}{E, S \vdash s_1; s_2 : S''}$$

Variable Assign

$$\frac{E \vdash v_1 : l_1, v_2 : l_2 \quad S \vdash l_2 : \mathfrak{e}}{E, S \vdash v_1 = v_2 : S[l_1 \leftarrow \mathfrak{e}]}$$

Constant Assign

$$\frac{E \vdash v : l \quad S' = S[l \leftarrow (c, \cdot)]}{E, S \vdash v = c : S'}$$

Alloc

$$\frac{\begin{array}{l} E \vdash v_1 : l_1, v_2 : l_2 \\ S \vdash l_2 : (c, 0) \\ s = newloc(S, c) \\ S' = S[(s, 0) \leftarrow 0, ..., (s, c-1) \leftarrow 0] \\ S'' = S'[l_1 \leftarrow (s, 0)] \end{array}}{E, S \vdash v_1 = alloc(v_2) : S''}$$

Array Load

$$\frac{\begin{array}{l} E \vdash v_1 : l_1, v_2 : l_2, v_3 : l_3 \\ S \vdash l_3 : (c, \cdot) \\ S \vdash l_2 : (s, i) \\ S \vdash (s, i + c) : \mathfrak{e} \\ S' = S[l_1 \leftarrow \mathfrak{e}] \end{array}}{E, S \vdash v_1 = v_2[v_3] : S'}$$

Array Store

$$\frac{\begin{array}{l} E \vdash v_1 : l_1, v_2 : l_2, v_3 : l_3 \\ S \vdash l_3 : (c, \cdot) \\ S \vdash l_2 : (s, i) \\ S \vdash l_1 : \mathfrak{e} \\ S' = S[(s, i + c) \leftarrow \mathfrak{e}] \end{array}}{E, S \vdash v_2[v_3] = v_1 : S'}$$

Pointer plus

$$\frac{\begin{array}{l} E \vdash v_1 : l_1, v_2 : l_2, v_3 : l_3 \\ S \vdash l_2 : (s, i) \\ S \vdash l_3 : (c, \cdot) \\ S' = S[l_1 \leftarrow (s, i + c)] \end{array}}{E, S \vdash v_1 = v_2 \oplus v_3 : S'}$$

Intop, Predop

$$\frac{\begin{array}{l} E \vdash v_1 : l_1, v_2 : l_2, v_3 : l_3 \\ S \vdash l_2 : (c, \cdot) \\ S \vdash l_3 : (c', \cdot) \\ S' = S[l_1 \leftarrow (c \text{ op } c', \cdot)] \end{array}}{E, S \vdash v_1 = v_2 \text{ op } v_3 : S'}$$

If-True

$$\frac{\begin{array}{c} E \vdash v : l \\ S \vdash l : (c, \cdot) \\ c \neq 0 : true \\ E, S \vdash s_1 : S' \end{array}}{E, S \vdash if v \neq 0 \ then \ s_1 else \ s_2 : S'}$$

If-False

$$\frac{\begin{array}{c} E \vdash v : l \\ S \vdash l : (c, \cdot) \\ c \neq 0 : false \\ E, S \vdash s_2 : S' \end{array}}{E, S \vdash if v \neq 0 \ then \ s_1 else \ s_2 : S'}$$

While-True

$$\frac{\begin{array}{l} E \vdash v : l \\ S \vdash l : (c, \cdot) \\ c \neq 0 : true \\ E, S \vdash s : S' \\ E, S' \vdash while \ v \neq 0 \ do \ s \ end : S'' \end{array}}{E, S \vdash while \ v \neq 0 \ do \ s \ end : S''}$$

While-False

$$\frac{\begin{array}{c} E \vdash v : l \\ S \vdash l : (c, \cdot) \\ c \neq 0 : false \end{array}}{E, S \vdash while \ v \neq 0 \ do \ s \ end : S}$$

Figure 4.1: Operational Semantics for the Language with Arrays

Example 1 Consider an edge from location $\langle *\mathtt{a}\rangle_{\mathtt{i}}$ to $*\mathtt{NULL}$ qualified by $\langle 0 \leq i < \mathit{size}, 0 \leq i < \mathit{size}\rangle$. This constraint expresses that *all* elements of the array with indices between 0 and *size* are `NULL`. Since it is sufficient that i is between 0 and *size* for $\langle *\mathtt{a}\rangle_{\mathtt{i}}$ to point to $*\mathtt{NULL}$, it follows that all elements in this range are `NULL`. On the other hand, if the constraint on the edge is $\langle 0 \leq i < \mathit{size},\ \mathit{false}\rangle$, any element in the array may be `NULL`, but no element must be `NULL`.

Boolean operators $\neg, \wedge$, and $\vee$ on bracketing constraints are defined as:

$$\begin{aligned} \neg\langle A_{NC}, A_{SC}\rangle &= \langle \neg A_{SC}, \neg A_{NC}\rangle \\ \langle A_{NC1}, A_{SC1}\rangle \star \langle A_{NC2}, A_{SC2}\rangle &= \langle A_{NC1} \star A_{NC2}, A_{SC1} \star A_{SC2}\rangle \quad (\star \in \{\wedge, \vee\}) \end{aligned}$$

Since the negation of the overapproximation for some set S is an underapproximation for the complement of S, necessary and sufficient conditions are swapped under negation. The following lemma is easy to show:

Lemma 1 *Bracketing constraints preserve the well-formedness property* $A_{\mathrm{SC}} \Rightarrow A_{\mathrm{NC}}$ *under boolean operations.*

Proof 1 *We only prove this for disjunction. We have* $A_{\mathrm{SC1}} \Rightarrow A_{\mathrm{NC1}}$ *and* $A_{\mathrm{SC2}} \Rightarrow A_{\mathrm{NC2}}$*. By weakening, this implies* $A_{\mathrm{SC1}} \Rightarrow A_{\mathrm{NC1}} \vee A_{\mathrm{NC2}}$ *and* $A_{\mathrm{SC2}} \Rightarrow A_{\mathrm{NC1}} \vee A_{\mathrm{NC2}}$*. From this, it follows immediately that* $A_{\mathrm{SC1}} \vee A_{\mathrm{SC2}} \Rightarrow A_{\mathrm{NC1}} \vee A_{\mathrm{NC2}}$*.*

Definition 1 (Satisfiability, Validity) *Satisfiability and validity of bracketing constraints are defined as follows:*

$$\mathrm{SAT}(\langle A_{\mathrm{NC}}, A_{\mathrm{SC}}\rangle) \equiv \mathrm{SAT}(A_{\mathrm{NC}}) \qquad \mathrm{VALID}(\langle A_{\mathrm{NC}}, A_{\mathrm{SC}}\rangle) \equiv \mathrm{VALID}(A_{\mathrm{SC}})$$

Lemma 2 *Bracketing constraints do not obey the law of the excluded middle and non-contradiction, but they satisfy the following weaker properties:*

$$\mathrm{VALID}(\lceil \langle A_{\mathrm{NC}}, A_{\mathrm{SC}}\rangle \vee \neg\langle A_{\mathrm{NC}}, A_{\mathrm{SC}}\rangle \rceil) \qquad \mathrm{UNSAT}(\lfloor \langle A_{\mathrm{NC}}, A_{\mathrm{SC}}\rangle \wedge \neg\langle A_{\mathrm{NC}}, A_{\mathrm{SC}}\rangle \rfloor)$$

Proof 2 $\lceil \langle A_{\mathrm{NC}}, A_{\mathrm{SC}}\rangle \vee \neg\langle A_{\mathrm{NC}}, A_{\mathrm{SC}}\rangle \rceil$ *is* $(A_{\mathrm{NC}} \vee \neg A_{\mathrm{SC}}) \Leftrightarrow (A_{\mathrm{SC}} \Rightarrow A_{\mathrm{NC}}) \Leftrightarrow \mathrm{true}$*, where the last equivalence follows from well-formedness. Similarly,* $\lfloor \langle A_{\mathrm{NC}}, A_{\mathrm{SC}}\rangle \wedge \neg\langle A_{\mathrm{NC}}, A_{\mathrm{SC}}\rangle \rfloor$ *is* $(A_{\mathrm{SC}} \wedge \neg A_{\mathrm{NC}}) \Leftrightarrow \mathrm{false}$*, where the last step follows from the well-formedness property.*

4.2 Abstract Locations and the Symbolic Heap

Abstract locations are named by *access paths* [64] and defined by the grammar:

$$\textit{Access Path } \pi \quad := \quad l_v \mid \textit{alloc}_{id} \mid \langle \pi \rangle_i \mid * \pi \mid c \mid \pi_1 \text{ intop } \pi_2 \mid \top$$

Here, l_v denotes the abstract location corresponding to variable v, and $\textit{alloc}_{id}$ denotes locations allocated at program point *id*. 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 the indices of π (similar to [35]). The location $*\pi$ represents the dereference of π. The access path c denotes constants, $\pi_1 \text{ intop } \pi_2$ represents the result of performing intop on π_1 and π_2, and $\top$ denotes any possible value.

A *memory access path*, denoted π_{mem}, is any access path that does not involve c, $\pi_1 \text{ intop } \pi_2$, and $\top$. We differentiate memory access paths because only locations that are identified by memory access paths may be written to; other kinds of access paths are only used for encoding values of scalars.

Given a concrete store S and an environment E mapping program variables to locations as defined in Section 4.1, a function γ maps abstract memory locations to a set of concrete locations $(s_1, i_1) \ldots (s_k, i_k)$:

$$\begin{array}{lcl}
\gamma(E, S, l_v) & = & \{E(v)\} \\
\gamma(E, S, \textit{alloc}_{id}) & = & \{(l, 0) \mid l \textit{ is the result of allocation at program point id }\} \\
\gamma(E, S, \langle \pi \rangle_i) & = & \{(l, \textit{index}_j) \mid (l, \textit{index}_j) \in S \wedge (l, 0) \in \gamma(E, S, \pi))\} \\
\gamma(E, S, *\pi) & = & \bigcup_{l_i \in \gamma(E,S,\pi)} S(l_i)
\end{array}$$

Since we will concretize abstract memory locations under a certain assumption about their index variables, we define another function γ_c, similar to γ but qualified by constraint ϕ. The only interesting modification is for $\langle \pi \rangle_i$:

$$\gamma_c(E, S, \langle \pi \rangle_i, \phi) = \{(l, \textit{index}_j) \mid (l, \textit{index}_j) \in S \ \wedge \ (l, 0) \in \gamma_c(E, S, \pi, \phi) \ \wedge \ \text{SAT}(\phi[\textit{index}_j / i])\}$$

As is standard in points-to graphs, we enforce that either

$$\pi_{mem} = \pi'_{mem} \quad or \quad \gamma(E, S, \pi_{mem}) \cap \gamma(E, S, \pi'_{mem}) = \emptyset$$

.

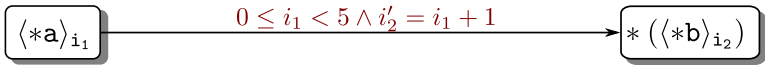
A *symbolic heap* is a directed graph where nodes denote abstract locations identified by access paths and edges qualified by bracketing constraints denote points-to relations. Since we want to uniformly encode points-to and value information, we extend the notion of points-to relations to scalars. For example, if an integer `a` has value 3, the symbolic heap representation contains a "points-to" edge from `a`'s location to some location named *3, thereby encoding that the value of `a` is 3. Hence, the memory graph also encodes the value of each scalar.

Formally, a symbolic heap is defined by

$$\Gamma : \pi_{mem} \rightarrow 2^{(\pi, \phi)}$$

mapping a source location to a set of (target location, constraint) pairs. The edge constraint ϕ may constrain program variables to encode the condition under which this points-to relation holds. More interestingly, ϕ may also qualify the source and the target location's index variables, thereby specifying which elements of the source may (and must) point to which elements of the target.

The combination of indexed locations and edge constraints qualifying these index variables makes the symbolic heap both expressive but also non-trivial to interpret. In particular, if the source location is an array, we might want to determine the points-to targets of a specific element (or some of the elements) in this array. However, the symbolic heap does not directly provide this information since edge constraints are parametric over the source and the target's index variables. Consider the following points-to relation:



Suppose we want to know which location(s) the fourth element of array $\langle *a \rangle_{i_1}$ points to. Intuitively, we can determine the target of the fourth element of $\langle *a \rangle_{i_1}$ by substituting the index variable i_1 by value 3 in the edge constraint $0 \leq i_1 < 5 \wedge i'_2 = i_1 + 1$.

This would yield $i'_2 = 4$, indicating that the fourth element of $\langle *a \rangle_i$ points to the target of the fifth element of $\langle *b \rangle_{i_2}$.

While a simple substitution allows us to determine the target of a specific array element as in the above example, in general, we need to determine the points-to targets of those array elements whose indices satisfy a certain constraint. Since this constraint may not limit the index variable to a single value, determining points-to targets of locations using the symbolic heap requires existential quantification in general. In the above example, we can determine the possible targets of elements of $\langle *a \rangle_{i_1}$ whose indices are in the range $[0, 3]$ (i.e., satisfy the constraint $0 \leq i_1 \leq 3$) by eliminating the existentially quantified variable i_1 from the following formula:

$$\exists i_1.(0 \leq i_1 \leq 3 \wedge (0 \leq i_1 < 5 \wedge i'_2 = i_1 + 1))$$

This yields $1 \leq i'_2 \leq 4$, indicating that the target's index must lie in the range $[1, 4]$.

To formalize this intuition, we define an operation $\phi_1 \downarrow_I \phi_2$, which yields the result of restricting constraint ϕ_1 to only those values of the index variables I that are consistent with ϕ_2.

Definition 2 **($\phi_1 \downarrow_I \phi_2$)** Let ϕ_1 be a constraint qualifying a points-to edge and let ϕ_2 be a constraint restricting the values of index variables I. Then,

$$\phi_1 \downarrow_I \phi_2 \equiv \mathit{Eliminate}(\exists I.\ \phi_1 \wedge \phi_2)$$

where the function *Eliminate* performs existential quantifier elimination.

The quantifier elimination performed in this definition is exact because index variables qualifying the source or the target never appear in uninterpreted functions in this context; thus the elimination can be performed using [29].

4.3 Analysis Using Fluid Updates

In this section, we give deductive rules describing the basic pointer and value analysis using fluid updates. An invariant mapping

$$\Sigma : Var \rightarrow \pi_{mem}$$

maps program variables to abstract locations, and the environment Γ defining the symbolic heap abstraction maps memory access paths to a set of (access path, constraint) pairs (recall Section 4.2). Judgments of the form

$$\Sigma \vdash \mathtt{a} : l_a$$

indicate that variable `a`'s location is represented using the abstract location l_a, and judgments of the form

$$\Gamma \vdash_j \pi_s : \langle \pi_{t_j, \phi_j} \rangle$$

state that $\langle \pi_{t_j}, \phi_j \rangle \in \Gamma(\pi_s)$.

We first explain some notation used in Figure 4.2. The function $U(\phi)$ replaces the primed index variables in constraint ϕ with their unprimed counterparts, e.g., $U(i'_1 = 2)$ is $(i_1 = 2)$. This operation is necessary when traversing edges of the symbolic heap because the target location of an incoming edge becomes the source of the outgoing edge from this location as we traverse points-to edges. We use the notation $\Gamma \wedge \phi$ as shorthand for:

$$\Gamma'(\pi) = \{\langle \pi_l, \phi_l \wedge \phi \rangle \mid \langle \pi_l, \phi_l \rangle \in \Gamma(\pi)\}$$

A union operation $\Gamma = \Gamma' \cup \Gamma''$ on symbolic heaps is defined as:

$$\langle \pi', \phi' \vee \phi'' \rangle \in \Gamma(\pi) \;\Leftrightarrow\; \langle \pi', \phi' \rangle \in \Gamma'(\pi) \;\wedge\; \langle \pi', \phi'' \rangle \in \Gamma''(\pi).$$

We write $\mathfrak{I}(\pi)$ to denote the set of all index variables used in π, and we say "i is index of π" if i is the outermost index variable in π.

The basic rules of the pointer and value analysis using fluid updates are presented in Figure 4.2. We start by explaining the Array Load rule. In this inference rule, each

Assign

$$\frac{\Sigma \vdash v_1 : l_{v_1},\ v_2 : l_{v_2} \quad \Gamma' = \Gamma[l_{v_1} \leftarrow \Gamma(l_{v_2})]}{\Sigma, \Gamma \vdash\ v_1 = v_2\ :\ \Gamma'}$$

Alloc

$$\frac{\Sigma \vdash v_1 : l_{v_1} \quad \Gamma' = \ \Gamma[l_{v_1} \leftarrow \langle alloc_{id} \rangle_i] \wedge i' = 0\ \ (i \text{ fresh})}{\Sigma, \Gamma \vdash\ v_1 = alloc(v_2)\ :\ \Gamma'}$$

Array Load

$$\frac{\begin{array}{l}\Sigma \vdash v_1 : l_{v_1},\ v_2 : l_{v_2}, v_3 : l_{v_3} \\ \Gamma \vdash_j l_{v_2} : \langle \pi_{2_j}, \phi_{2_j} \rangle \ \ (i \text{ index of } \pi_{2_j}) \\ \Gamma \vdash_k l_{v_3} : \langle *\pi_{3_k}, \phi_{3_k} \rangle \\ \Gamma \vdash_l \pi_{2_j} : \langle \pi_{t_{jl}}, \phi_{t_{jl}} \rangle \\ \phi'_{2_{jk}} = U(\phi_{2_j}[i' - \pi_{3_k}/i']) \\ \phi'_{t_{jkl}} = \phi_{t_{jl}} \downarrow_{\mathfrak{I}(\pi_{2_j})} \phi'_{2_{jk}} \\ \Gamma' = \Gamma[l_{v_1} \leftarrow (\bigcup_{jkl} \langle \pi_{t_{jl}}, \phi'_{t_{jkl}} \wedge \phi_{3_k} \rangle)]\end{array}}{\Sigma, \Gamma \vdash\ v_1 = v_2[v_3]\ :\ \Gamma'}$$

Array Store (Fluid Update)

$$\frac{\begin{array}{l}\Sigma \vdash v_1 : l_{v_1},\ v_2 : l_{v_2}, v_3 : l_{v_3} \\ \Gamma \vdash_j l_{v_1} : \langle \pi_{1_j}, \phi_{1_j} \rangle \\ \Gamma \vdash l_{v_2} : \{\langle \pi_{2_1}, \phi_{2_1} \rangle \ldots \langle \pi_{2_n}, \phi_{2_n} \rangle\}\ (i_k \text{ index of } \pi_{2_k}) \\ \Gamma \vdash_l l_{v_3} : \langle *\pi_{3_l}, \phi_{3_l} \rangle \\ \Gamma' = \begin{cases} \pi \leftarrow \Gamma(\pi)\ \textit{if}\ \pi \notin \{\pi_{2_1}, \ldots \pi_{2_n}\} \\ \pi \leftarrow \{\langle \pi'_k, \phi'_k \wedge \neg \bigvee_{kl} (U(\phi_{2_k}[i'_k - \pi_{3_l}/i'_k]) \wedge \phi_{3_l}) \rangle \\ \qquad \mid \langle \pi'_k, \phi'_k \rangle \in \Gamma(\pi_{2_k})\} \ \textit{if}\ \pi = \pi_{2_k} \in \{\pi_{2_1}, \ldots \pi_{2_n}\} \end{cases} \\ \Gamma'' = \begin{cases} \pi_{2_1} \leftarrow (\bigcup_{jl} \langle \pi_{1_j}, U(\phi_{2_1}[i'_1 - \pi_{3_l}/i'_1]) \wedge \phi_{3_l} \wedge \phi_{1_j} \rangle) \\ \ldots \\ \pi_{2_n} \leftarrow (\bigcup_{jl} \langle \pi_{1_j}, U(\phi_{2_n}[i'_n - \pi_{3_l}/i'_n]) \wedge \phi_{3_l} \wedge \phi_{1_j} \rangle) \end{cases}\end{array}}{\Sigma, \Gamma \vdash\ v_2[v_3] = v_1\ :\ \Gamma' \cup \Gamma''}$$

Pointer Arithmetic

$$\frac{\begin{array}{l}\Sigma \vdash v_1 : l_{v_1}, v_2 : l_{v_2}, v_3 : l_{v_3} \\ \Gamma \vdash_j l_{v_2} : \langle \pi_{2_j}, \phi_{2_j} \rangle \\ \Gamma \vdash_k l_{v_3} : \langle *\pi_{3_k}, \phi_{3_k} \rangle \\ \phi'_{2_{jk}} = \phi_{2_j}[(i' - \pi_{3_k})/i'] \ \ (i \text{ index of } \pi_{2_j}) \\ \Gamma' = \Gamma[l_{v_1} \leftarrow (\bigcup_{jk} \langle \pi_{2_j}, \phi'_{2_{jk}} \wedge \phi_{3_k} \rangle)]\end{array}}{\Sigma, \Gamma \vdash v_1 = v_2 \oplus v_3 : \Gamma'}$$

Predop

$$\frac{\begin{array}{l}\Sigma \vdash v_1 : l_{v_1}, v_2 : l_{v_2}, v_3 : l_{v_3} \\ \Gamma \vdash_j l_{v_2} : \langle *\pi_{2_j}, \phi_{2_k} \rangle \ \ \text{(rename all index variables to fresh } \vec{f_2}) \\ \Gamma \vdash_k l_{v_3} : \langle *\pi_{3_k}, \phi_{3_k} \rangle \ \ \text{(rename all index variables to fresh } \vec{f_3}) \\ \phi_{jk} = (\overline{\pi_{2_j}} \text{ predop } \overline{\pi_{3_k}}) \wedge \phi_{2_j} \wedge \phi_{3_k} \\ \phi_{jk}^{true} = \textit{Eliminate}(\exists \vec{f_2}, \vec{f_3}.\ \phi_{jk}) \\ \Gamma' = \Gamma[l_{v_1} \leftarrow (\bigcup_{jk} \langle *1, \phi_{jk}^{true} \rangle \cup \langle *0, \neg \phi_{jk}^{true} \rangle)]\end{array}}{\Sigma, \Gamma \vdash v_1 = v_2 \text{ predop } v_3 : \Gamma'}$$

If Statement

$$\frac{\begin{array}{l}\Sigma \vdash v : l_v \\ \Gamma \vdash l_v : \{\langle *1, \phi_{true} \rangle, \langle *0, \phi_{false} \rangle\} \\ \Sigma, \Gamma \vdash S_1 : \Gamma' \\ \Sigma, \Gamma \vdash S_2 : \Gamma'' \\ \Gamma_T = \Gamma' \wedge \phi_{true} \\ \Gamma_F = \Gamma'' \wedge \phi_{false}\end{array}}{\Sigma, \Gamma \vdash \textit{if}\ v \neq 0\ \textit{then}\ S_1\ \textit{else}\ S_2 : \Gamma_T \cup \Gamma_F}$$

While Loop

$$\frac{\begin{array}{l}\Gamma_P = \textit{Parametrize}(\Gamma) \\ \Sigma \vdash v : l_v \\ \Gamma_P \vdash l_v : \{\langle *1, \phi_{true} \rangle, \langle *0, \phi_{false} \rangle\} \\ \Sigma, \Gamma_P \vdash S : \Gamma'' \quad \Gamma''' = \Gamma'' \wedge \phi_{true} \\ \Delta = \Gamma''' - \Gamma_P \quad \Delta_n = \textit{fix}(\Delta) \\ \Delta_{gen} = \textit{Generalize}(\Delta_n) \\ \Gamma_{final} = \Gamma \circ \Delta_{gen} \text{ (Generalized Fluid Update)}\end{array}}{\Sigma, \Gamma \vdash \textit{while}\ v \neq 0\ \textit{do}\ S\ \textit{end}\ : \Gamma_{final}}$$

Figure 4.2: Rules describing the basic analysis

π_{2_j} represents one possible points-to target of v_2 under constraint ϕ_{2_j}. Because π_{2_j} is an array, the constraint ϕ_{2_j} qualifies π_{2_j}'s index variables. Now, each π_{3_k} represents one possible (scalar) value of v_3. Since we want to access the element at offset v_3 of v_2's target, we select the element at offset v_3 by substituting i' with $i' - \pi_{3_k}$ in the constraint ϕ_{2_j}, which effectively increments the value of i' by π_{3_k}. Now, we need to

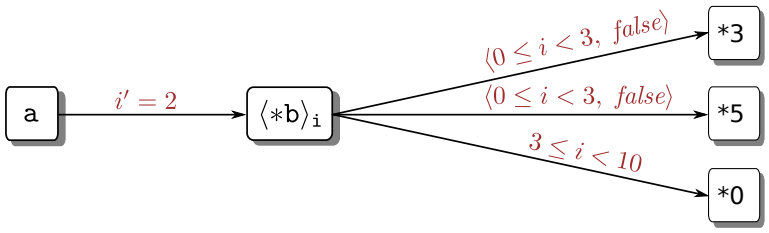


Figure 4.3: Here, a points to the third element of an array b of size 10. The first three elements of b may have the value 3 or 5, and the elements in the range $[3, 9]$ are guaranteed to have value 0.

determine the targets of those elements of π_{2_j} whose indices are consistent with $\phi'_{2_{jk}}$; hence, we compute $\phi_{t_{jl}} \downarrow_{\mathfrak{I}(\pi_{2_j})} \phi'_{2_{jk}}$ (recall Section 4.2) for each target $\pi_{t_{jl}}$ of π_{2_j}. The following example illustrates the analysis of loads from array elements.

Example 2 Consider performing the array load operation $\mathtt{t} = \mathtt{a}[1]$ on the symbolic heap shown in Figure 4.3. Here, l_{v_2} is the memory location labeled a, the only target π_{2_j} of l_{v_2} is $\langle *\mathtt{b} \rangle_{\mathtt{i}}$, and the only π_{3_k} is 1. The constraint $\phi'_{2_{jk}}$ is $U((i' = 2)[i'/i' - 1])$, which is $i = 3$. Thus, we need to determine the target(s) of the fourth element in array $\langle *\mathtt{b} \rangle_{\mathtt{i}}$. There are three targets $\pi_{t_{jl}}$ of $\langle *\mathtt{b} \rangle_{\mathtt{i}}$: $*3, *5, *0$; hence, we compute $\phi'_{t_{jkl}}$ once for each $\pi_{t_{jkl}}$. The only satisfiable edge under constraint $i = 3$ is the edge to *0 and we compute $\mathit{Eliminate}(\exists i.\ 3 \le i < 10 \wedge i = 3)$, which is *true*. Thus, the analysis determines that the value of t is 0 after this statement.

The Array Store rule performs a fluid update on an abstract memory location associated with an array. In this rule, each $\pi_{2_k} \in \{\pi_{2_1} \ldots \pi_{2_n}\}$ represents an array location, a subset of whose elements may be written to as a result of this store. Γ' represents the symbolic heap after removing the points-to edges from array elements that are written to by this store while preserving all other edges, and Γ'' represents all edges added by this store operation. Hence, Γ' and Γ'' are unioned to obtain the symbolic heap after the store. Note that Γ' preserves the existing targets of any access path $\pi \notin \{\pi_{2_1} \ldots \pi_{2_n}\}$. The points-to targets of those elements of $\pi_{2_1}, \ldots \pi_{2_n}$ that are not affected by this store are also preserved in Γ' while elements that are written to by the store are killed in Γ'. This is because elements that are updated by the store

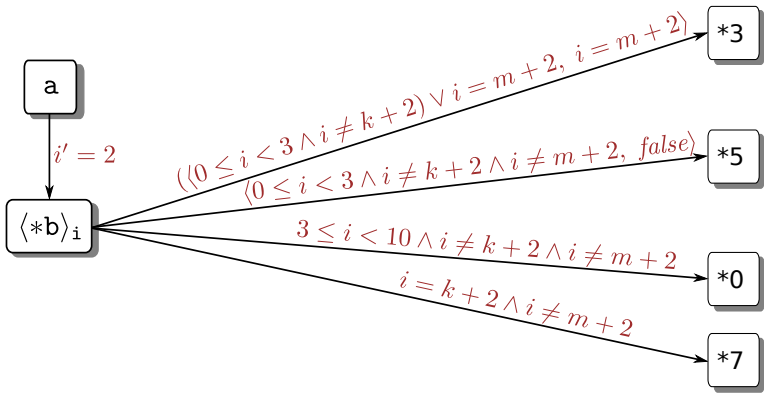


Figure 4.4: Graph after processing the statements in Example 3

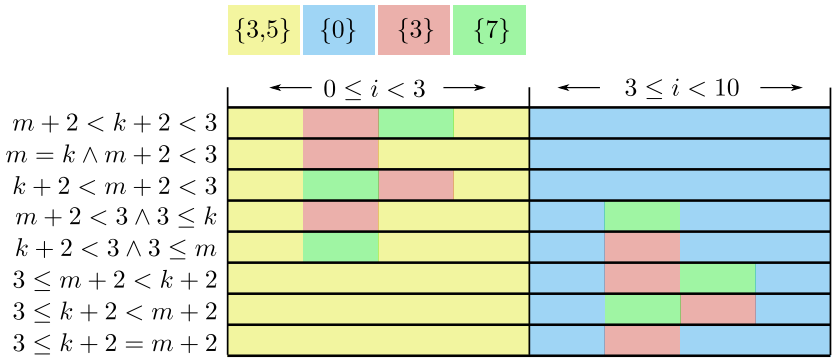


Figure 4.5: Colored rectangles illustrate the partitions in Example 3; equations on the left describe the ordering between variables.

must satisfy $U(\phi_{2_k}[i'_k - \pi_{3_l}/i'_k]) \wedge \phi_{3_l}$ for some k, l such that the edge to π'_k is effectively killed for those elements updated by the store. On the other hand, elements that are *not* affected by the store are guaranteed not to satisfy $U(\phi_{2_k}[i'_k - \pi_{3_l}/i'_k]) \wedge \phi_{3_l}$ for any k, l, i.e., $\neg \bigvee_{kl}(U(\phi_{2_k}[i'_k - \pi_{3_l}/i'_k]) \wedge \phi_{3_l}) = \mathit{false}$, and the existing edge to π'_k is therefore preserved. Note that negation is only used in the Fluid Update rule; the soundness of negation, and therefore the correctness of fluid updates, relies on using bracketing constraints.

Example 3 Consider the effect of the following store instructions

```
a[k] = 7;  a[m] = 3;
```

on the symbolic heap shown in Figure 4.3. Suppose `k` and `m` are symbolic, i.e., their values are unknown. When processing the statement `a[k] = 7`, the only location stored into, i.e., π_{2_k}, is $\langle *\texttt{b}\rangle_{\texttt{i}}$. The only π_{3_l} is `k` under *true*, and the only π_{1_j} is $*\texttt{7}$ under *true*. The elements of $\langle *\texttt{b}\rangle_{\texttt{i}}$ updated by the store are determined from $U((i' = 2)[i' - \texttt{k}/i']) = (i = \texttt{k} + 2)$. Thus, a new edge is added from $\langle *\texttt{b}\rangle_{\texttt{i}}$ to $*\texttt{7}$ under $i = \texttt{k} + 2$ but all outgoing edges from $\langle *\texttt{b}\rangle_{\texttt{i}}$ are preserved under the constraint $i \neq \texttt{k} + 2$. Thus, after this statement, the edge from $\langle *\texttt{b}\rangle_{\texttt{i}}$ to $*\texttt{3}$ and $*\texttt{5}$ are qualified by the constraint $\langle 0 \leq i < 3 \wedge i \neq k + 2,\ \mathit{false}\rangle$, and the edge to $*\texttt{0}$ is qualified by $3 \leq i < 10 \wedge i \neq k + 2$. The instruction `a[m] = 3` is processed similarly; Figure 4.4 shows the resulting heap abstraction after these store instructions. Note that if $k = m$, the graph correctly reflects `a[k]` must be 3. This is because if $k = m$, the constraint on the edge from $\langle *\texttt{b}\rangle_{\texttt{i}}$ to $*\texttt{7}$ is unsatisfiable. Since the only other feasible edge under the constraint $i = k + 2$ is the one to $*\texttt{3}$, $k = m$ implies `a[k]` must be 3.

As Example 3 illustrates, fluid updates do not construct explicit partitions of the heap when different symbolic values are used to store into an array. Instead, all "partitions" are implicitly encoded in the constraints, and while the constraint solver may eventually need to analyze all of the cases, it will often not need to do so because a query is more easily shown satisfiable or unsatisfiable for other reasons. As a comparison, in Example 3, approaches that eagerly construct explicit partitions, such as [48], may be forced to enumerate all partitions created due to stores using symbolic indices. Figure 4.3 shows that eight different heap configurations arise after performing the updates in Example 3. In fact, only one more store using a symbolic index could create over 50 different heap configurations.

In the Pointer Arithmetic rule, the index variable i' is replaced by $i' - \pi_{3_k}$ in the index constraint ϕ_{2_j}, effectively incrementing the value of i' by v_3. We also discuss the Predop rule, since some complications arise when array elements are used in predicates. In this rule, we make use of an operation $\overline{\pi}$ which converts an access path to a term in the constraint language:

$$\overline{\pi_R} = \pi_R \quad \text{if } \pi_R \in \{c, l_v, \mathit{alloc}_{id}\} \qquad \overline{*\pi} = \mathit{deref}(\overline{\pi})$$

$$\overline{\langle \pi \rangle_i} = \mathit{select}(\overline{\pi}, i) \qquad \overline{\pi_1 \text{ intop } \pi_2} = \overline{\pi_1} \text{ intop } \overline{\pi_2}$$

In this rule, notice that index variables used in the targets of l_{v_2} and l_{v_3} are first renamed to fresh variables $\vec{f_2}$ and $\vec{f_3}$ to avoid naming conflicts and are then existentially quantified and eliminated similar to computing $\phi_1 \downarrow_I \phi_2$. The renaming of index variables is necessary since naming conflicts arise when $\langle *\pi_{2_j}, \phi_{2_j} \rangle$ and $\langle *\pi_{3_k}, \phi_{3_k} \rangle$ refer to different elements of the same array. [1]

In the If Statement rule, observe that the constraint under which $v \neq 0$ evaluates to true (resp. false) is conjoined with all the edge constraints in Γ' (resp. Γ''); hence, the analysis is path-sensitive. We defer discussion of the While Loop rule until Section 4.4.

4.3.1 Soundness of the Memory Abstraction

We now state the soundness theorem for our symbolic heap abstraction. For convenience of presentation, we use the notation $S(l_s, l_t) = \mathit{true}$ if $S(l_s) = l_t$ and $S(l_s, l_t) = \mathit{false}$ otherwise. Similarly, we write $\Gamma(\pi_s, \pi_t) = \phi$ to denote that the bracketing constraint associated with the edge from π_s to π_t is ϕ, and ϕ is *false* if there is no edge between π_s and π_t. Recall that $\mathfrak{I}(\pi)$ denotes the set of index variables in π, and we write $\sigma_{\mathfrak{I}(\pi)}$ to denote some concrete assignment to the index variables in $\mathfrak{I}(\pi)$; $\sigma'_{\mathfrak{I}(\pi)}$ is an assignment to $\mathfrak{I}(\pi)$ with all index variables primed. The notation $\sigma(\phi)$ applies substitution σ to ϕ. Finally, we use a function $\mathit{eval}^{\star}(\phi, E, S)$ for $\star \in \{+, -\}$ which evaluates the truth value of constraint ϕ for some concrete E, S. To do this, we first define an eval_t function on terms in the constraint language as follows:

$$
\begin{array}{lcl}
\mathit{eval}_t(c, E, S) & = & \{c\} \\
\mathit{eval}_t(v, E, S) & = & \{S(l_i) |\ l_i \in \gamma(E, S, v)\} \\
\mathit{eval}_t(\mathit{select}(\mathit{deref}(t_1, t_2)), E, S) & = & \{S(s_1, i_1 + c) \mid (s_1, i_1) \in \mathit{eval}_t(t_1, E, S) \wedge (c, \cdot) \in \mathit{eval}_t(t_2, E, S)\} \\
\mathit{eval}_t(\mathit{deref}(t), E, S) & = & \{S(l_i) \mid l_i \in \mathit{eval}_t(t, E, S)\} \\
\mathit{eval}_t(t_1 \text{ intop } t_2, E, S) & = & \{v_{1i} + v_{2j} \mid (v_{1i}, \cdot) \in S(l_i) \wedge l_i \in \mathit{eval}_t(t_1, E, S) \\
 & & \wedge (v_{2j}, \cdot) \in S(l_j) \wedge l_j \in \mathit{eval}_t(t_2, E, S)\}
\end{array}
$$

[1] Quantifier elimination performed here may not be exact; but since we use bracketing constraints, we compute quantifier-free over- and underapproximations. For instance, [51] presents a technique for computing covers of existentially quantified formulas in combined theories involving uninterpreted functions. Another alternative is to allow quantification in our constraint language.

Since the language only allows pointers to arrays, terms involving *select* are guaranteed to be followed by a *deref*. Thus, we give a definition for $eval_t(select(deref(t_1, t_2)))$, but not for $eval_t(select(t_1, t_2))$. We define $val(t)$ for a term t as follows:

$$val(t,\ E,\ S) = \{s_i + off_i | (s_i, off_i) \in eval_t(t, E, S)\}$$

where $off_i = 0$ if $(s_i, \cdot) \in eval_t(t, E, S)$.

Finally, we define the $eval^\star$ ($\star \in \{+, -\}$) function for constraints in the following way:

$$\begin{array}{lcl}
eval^\star(b, E, S) & = & b, \quad b \in \{true, false\} \\
eval^\star(t_1 \text{ predop } t_2, E, S) & = & \begin{cases} \bigvee_{v_i \in val(t_1,E,S), v_j \in val(t_2,E,S)} (v_i \text{ predop } v_j) & if \star = + \\ \bigwedge_{v_i \in val(t_1,E,S), v_j \in val(t_2,E,S)} (v_i \text{ predop } v_j) & if \star = - \end{cases} \\
eval^\star(t \bmod c, E, S) & = & \begin{cases} \bigvee_{v_i \in val(t,E,S)} (v_i \bmod c) & if \star = + \\ \bigwedge_{v_i \in val(t,E,S)} (v_i \bmod c) & if \star = - \end{cases} \\
eval^\star(\neg\phi, E, S) & = & \begin{cases} \neg eval^-(\phi, E, S) & if \star = + \\ \neg eval^+(\phi, E, S) & if \star = - \end{cases} \\
eval^\star(\phi_1 \wedge \phi_2, S) & = & eval^\star(\phi_1, E, S) \wedge eval^\star(\phi_2, E, S) \\
eval^\star(\phi_1 \vee \phi_2, S) & = & eval^\star(\phi_1, E, S) \vee eval^\star(\phi_2, E, S)
\end{array}$$

Definition 3 (Agreement) We say a concrete environment and concrete store (E, S) *agree with* abstract environment and abstract store (Σ, Γ) (written $(E, S) \sim (\Sigma, \Gamma)$) if and only if the following conditions hold:

1. E and Σ have the same domain

2. If $S(l_s, l_t) = b$ and $\Gamma(\pi_s, \pi_t) = \langle \phi^+, \phi^- \rangle$, then for all substitutions $\sigma_{\mathfrak{I}(\pi_s)}, \sigma'_{\mathfrak{I}(\pi_t)}$ such that $l_s \in \gamma_c(E, S, \pi_s, \sigma_{\mathfrak{I}(\pi_s)})$ and $l_t \in \gamma_c(E, S, \pi_t, \sigma'_{\mathfrak{I}(\pi_t)})$, we have:

$$eval^-(\sigma'(\sigma(\phi^-)), E, S) \Rightarrow b \Rightarrow eval^+(\sigma'(\sigma(\phi^+)), E, S)$$

Theorem 1 (Soundness) Let P be any program. If $(E, S) \sim (\Sigma, \Gamma)$, then

$$E, S \vdash P : S' \Rightarrow (\Sigma, \Gamma \vdash P : \Gamma' \ \wedge \ (E, S') \sim (\Sigma, \Gamma'))$$

We sketch the proof of soundness of the fluid update operation in Section 4.8.

4.4 Fluid Updates in Loops

In loop-free code, a store modifies one array element, but stores inside a loop often update many elements. In this section, we describe a technique to over- and under-approximate the set of concrete elements updated in loops. While this step requires finding some invariants, the particular invariant generation technique is orthogonal to fluid updates; other invariant generation techniques than the one we propose can be used with the proposed analysis.

The main idea of our approach is to analyze the loop body and perform a fixed-point computation parametric over an *iteration counter*. Once a fixed-point is reached, we use quantifier elimination to infer elements that may and must be modified by the loop. [2]

4.4.1 Parametrizing the Abstraction

When analyzing loops, our analysis first identifies the set of scalars modified by the loop; we call such values *loop-dependent* scalars. We then infer equalities relating each loop-dependent scalar to the unique iteration counter k for that loop. The iteration counter k is assumed to be initialized to 0 at loop entry and is incremented by one along the back edge of the loop. We say that a loop-dependent value i is *linear* with respect to the loop if $i - i_0 = c * k$ for some constant $c \neq 0$. We compute a set of equalities relating loop-dependent scalars to the iteration counter using standard linear invariant generation techniques [58, 31]. At loop entry, we use these linear equalities to modify Γ as follows:

[2] In this section, we assume no pointer arithmetic occurs in loops; our implementation, however, does not make this restriction.

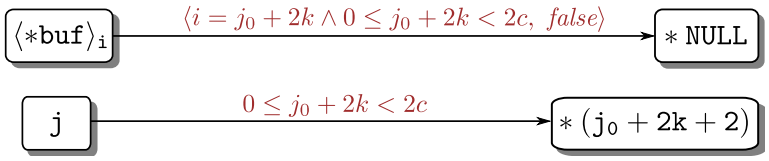


Figure 4.6: The effect set after analyzing the loop body once in function `send_packets`

- Let π be a linear loop-dependent scalar with the linear relation $\pi = \pi_0 + c * k$, and let $\langle *\pi_t, c_t \rangle \in \Gamma(\pi)$. Then, replace π_t by $\pi_t + c * k$.
- Let π be a loop-dependent value not linear in k. Then, $\Gamma(\pi) \leftarrow \{\langle \top, \mathit{true} \rangle\}$.

Thus, all loop-dependent scalars are expressed in terms of their value at iteration k or $\top$; analysis of the loop body proceeds as described in Section 4.3.

Example 4 Consider the `send_packets` function from Section 3.2. Here, we infer the equality $j = j_0 + 2k$, and Γ initially contains an edge from j to $*(j_0 + 2k)$.

4.4.2 Fixed-Point Computation

Next, we perform a fixed-point computation (parametric on k) over the loop's net effect on the symbolic heap abstraction. Performing a fixed-point computation is necessary because there may be loop carried dependencies through heap reads and writes. We define the net effect of the loop on the symbolic heap during some iteration k as the *effect set*:

Definition 4 (Effect Set Δ) Let Γ' be a symbolic heap obtained by performing fluid updates on Γ. Let $\Delta = \Gamma' - \Gamma$ be the set of edges such that if ϕ qualifies edge e in Γ and ϕ' qualifies e in Γ', then Δ includes e under constraint $\phi' \wedge \neg\phi$ (where $\phi = \mathit{false}$ if $e \notin \Gamma$). We call Δ the *effect set* of Γ' with respect to Γ.

Example 5 Figure 4.6 shows the effect set of the loop in `send_packets` after analyzing its body once. (Edges with *false* constraints are not shown.) Note that the constraints qualifying edges in this figure are parametric over k.

We define $\Gamma \circ \Delta$ as the generalized fluid update that applies Δ to Γ:

Definition 5 ($\mathbf{\Gamma \circ \Delta}$) Let π be a location in Γ and let S_π denote the edges in Δ whose source is π. Let $\delta(S_\pi)$ be the disjunction of constraints qualifying edges in S_π, and let I be the set of index variables used in the target locations in S_π but not the source. Let $\mathit{Update}(\pi) = \mathit{Eliminate}(\exists I.\delta(S_\pi))$. Then, for each $\pi \in \Gamma$:

$$(\Gamma \circ \Delta)[\pi] = (\Gamma(\pi) \wedge \neg \mathit{Update}(\pi)) \cup S_\pi$$

The above definition is a straightforward generalization of the fluid update operation given in the Store rule of Figure 4.2. Instead of processing a single store, it reflects the overall effect on Γ of a set of updates defined by Δ. The fixed-point computation is performed on Δ. We denote an edge from location π_s to π_t qualified by constraint ϕ as $\langle \pi_s, \pi_t \rangle \backslash \phi$. Since we compute a least fixed point, $\langle \pi_s, \pi_t \rangle \backslash \langle \mathit{false}, \mathit{true} \rangle \in \perp$ for all legal combinations (i.e., obeying type restrictions) of all $\langle \pi_s, \pi_t \rangle$ pairs. Note that the edge constraints in $\perp$ are the inconsistent bound $\langle \mathit{false}, \mathit{true} \rangle$ representing the strongest over- and underapproximations. We define a $\sqcup$ and $\sqsubseteq$ on effect sets as follows:

$$\begin{array}{c} \langle \pi_s, \pi_t \rangle \backslash \langle (\phi_{nc1} \vee \phi_{nc2}), (\phi_{sc1} \wedge \phi_{sc2}) \rangle \in \Delta_1 \sqcup \Delta_2 \\ \Longleftrightarrow \\ (\langle \pi_s, \pi_t \rangle \backslash \langle \phi_{nc1}, \phi_{sc1} \rangle \in \Delta_1 \wedge \\ \langle \pi_s, \pi_t \rangle \backslash \langle \phi_{nc2}, \phi_{sc2} \rangle \in \Delta_2) \end{array} \qquad \begin{array}{c} \Delta_1 \sqsubseteq \Delta_2 \\ \Longleftrightarrow \\ ((\phi_{nc1} \Rightarrow \phi_{nc2} \wedge \phi_{sc2} \Rightarrow \phi_{sc1}) \\ \forall \langle \pi_s, \pi_t \rangle \backslash \langle \phi_{nc1}, \phi_{sc1} \rangle \in \Delta_1 \wedge \\ \forall \langle \pi_s, \pi_t \rangle \backslash \langle \phi_{nc2}, \phi_{sc2} \rangle \in \Delta_2) \end{array}$$

Let Γ_0 be the initial symbolic heap abstraction before the loop. We compute $\Gamma^n_{\mathit{entry}}$ representing the symbolic heap at entry to the n'th iteration of the loop as:

$$\Gamma^n_{\mathit{entry}} = \begin{cases} \Gamma_0 & \mathit{if} \quad n = 1 \\ \Gamma_0 \circ (\Delta_{n-1}[k-1/k]) & \mathit{if} \quad n > 1 \end{cases}$$

Γ^n_{exit} is obtained by analyzing the body of the loop using $\Gamma^n_{\mathit{entry}}$ at the entry point of the loop. The substitution $[k-1/k]$ normalizes the effect set with respect to the iteration counter so that values of loop-dependent scalars always remain in terms of

their value at iteration k. We define Δ_n representing the total effect of the loop in n iterations as follows:

$$\Delta_n = \begin{cases} \bot & \text{if } n = 0 \\ (\Gamma^n_{exit} - \Gamma^n_{entry}) \sqcup \Delta_{n-1} & \text{if } n > 0 \end{cases}$$

First, observe that $\Delta_{n-1} \sqsubseteq \Delta_n$ by construction (monotonicity). Second, observe the analysis cannot create an infinite number of abstract locations because (i) arrays are represented as indexed locations, (ii) pointers can be dereferenced only as many times as their types permit, (iii) all allocations are named by their allocation site, (iv) scalars are represented in terms of their linear relation to k. However, our constraint domain does not have finite ascending chains, therefore we define a widening operator on bracketing constraints (but note widening was never required in our experiments). Let $\vec{\beta}$ denote the variables in the unshared literals between any constraint ϕ_1 and ϕ_2. Then, we widen bracketing constraints as follows:

$$\phi_1 \bigtriangledown \phi_2 = \ \langle \exists \vec{\beta}.\ (\lceil \phi_1 \rceil \vee \lceil \phi_2 \rceil),\ \forall \vec{\beta}.\ \ (\lfloor \phi_1 \rfloor \wedge \lfloor \phi_2 \rfloor) \rangle$$

Example 6 The effect set obtained in Example 5 does not change in the second iteration; therefore the fixed-point computation terminates after two iterations.

4.4.3 Generalization

In this section, we describe how to *generalize* the final effect set after a fixed-point is reached. This last step allows the analysis to extrapolate from the elements modified in the k'th iteration to the set of elements modified across all iterations and is based on existential quantifier elimination.

Definition 6 (Generalizable Location) We say a location identified by π is *generalizable* in loop l if (i) π is an array, (ii) if π_i is used as an index in a store to π, then π_i must be a linear function of the iteration counter, and (iii) if two distinct indices π_i and π_j may be used to store into π, then either only π_i, or only π_j (or neither) is used to index π across all iterations.

Intuitively, if a location π is generalizable in l, then all writes to π at different iterations of l must refer to distinct concrete elements. Clearly, if π is not an array, different iterations of the loop cannot refer to distinct concrete elements. If an index used to store into π is not a linear function of k, then the loop may update the same concrete element in different iterations. Furthermore, if two values that do not have the same relation with respect to k are used to store into π, then they may update the same element in different iterations.

In order to generalize the effect set, we make use of a variable N unique for each loop that represents the number of times the loop body executes. If the value of N can be determined precisely, we use this exact value instead of introducing N. For instance, if a loop increments i by 1 until $i \geq size$, then it is easy to determine that $N = size - i_0$, assuming the loop executes at least once. [3] Finally, we generalize the effect set as follows:

- If an edge qualified by ϕ has a generalizable source whose target does not mention k, the generalized constraint is $\phi' = \mathit{Eliminate}(\exists k.\ (\phi \wedge 0 \leq k < N))$.
- If an edge qualified by ϕ does not have a generalizable source, the generalized constraint is $\phi' = \mathit{Eliminate}\langle \exists k.\ \phi\ \wedge\ 0 \leq k < N,\ \ \forall k.\ 0 \leq k < N \Rightarrow \phi \rangle$ [4].
- If π is a loop-dependent scalar, then $\Delta[\pi] \leftarrow \Delta[\pi][N/k]$.

We now briefly explain these generalization rules. If the source of an edge is generalizable, for each iteration of the loop, there exists a corresponding concrete element of the array that is updated during this iteration; thus, k is existentially quantified in both the over- and underapproximation. The constraint after the existential quantifier elimination specifies the set of concrete elements updated by the loop. If the source is not generalizable, it is unsafe to existentially quantify k in the underapproximation since the same concrete element may be overwritten in future iterations. One way

[3] Even though it is often not possible to determine the exact value of N, our analysis utilizes the constraint $(\forall k.0 \leq k < N \Rightarrow \neg\phi_{term}(k))\ \wedge\ \phi_{term}(N)$ stating that the termination condition ϕ_{term} does not hold on iterations before N but holds at the N'th iteration. Our analysis takes this "background axiom" into account when determining satisfiability and validity.

[4] We can eliminate a universally quantified variable k from $\forall k.\phi$ by eliminating existentially quantified k in the formula $\neg\exists k.\neg\phi$.

to obtain an underapproximation is to universally quantify k because if the update happens in all iterations, then the update must happen after the loop terminates. According to the last rule, loop-dependent scalar values are assigned to their value on termination. Once the effect set is generalized, we apply it to Γ_0 to obtain the final memory graph after the loop.

Example 7 Consider the effect set given in Figure 4.6. In the `send_packets` function, $\langle *\texttt{buf}\rangle_{\texttt{i}}$ is generalizable since `j` is linear in k and no other value is used to index $\langle *\texttt{buf}\rangle_{\texttt{i}}$. Furthermore, if the loop executes, it executes exactly c times; thus $N = c$. To generalize the edge from $\langle *\texttt{buf}\rangle_{\texttt{i}}$ to $*\texttt{NULL}$, we perform quantifier elimination on $\langle \exists k.i = j_0 + 2k \wedge 0 \leq j_0 + 2k < 2c \wedge 0 \leq k < c,\ \mathit{false}\rangle$, which yields $\langle j_0 \leq i \wedge i < j_0 + 2c \wedge (i - j_0) \bmod 2 = 0,\ \mathit{false}\rangle$. Since j_0 is 0 at loop entry, after applying the generalized effect set to Γ_0 , we obtain the graph from Figure 3.2.

4.5 Implementation and Extensions

We have implemented the ideas presented in this chapter in the Compass program verification framework for analyzing C programs. For solving constraints, Compass utilizes a custom SMT solver called Mistral [36], which also provides support for simplifying constraints. Compass does not assume type safety and handles casts soundly using a technique based on physical subtyping [26]. Compass supports most features of the C language, including structs, unions, multi-dimensional arrays, dynamic memory allocation, and pointer arithmetic. To allow checking for buffer overruns, Compass also tracks buffer and allocation sizes. For interprocedural analysis, Compass performs a fully path- and context-sensitive summary-based bottom-up analysis. All loop bodies are analyzed in isolation before the function or loop in which they are defined; thus the techniques from Section 4.4 extend to nested loops.

While the language we consider in this chapter only allows loops with a single exit point, techniques described in this chapter can be extended to loops with multiple break points either by transforming them to loops with a single exit point or by introducing multiple iteration counters associated with each backedge, similar to

the technique used in [52] for complexity analysis. Our implementation uses the latter approach, since this approach is more precise for recovering linear invariants. While the technique for analyzing loops described in Section 4.4 are easily extended to tail-recursive functions, Compass performs a sound, but less precise, fixed-point computation in the case of general recursion.

Compass allows checking arbitrary assertions using a `static_assert`(...) primitive, which can be either manually or automatically inserted (e.g., for memory safety properties). The `static_assert` primitive also allows for checking quantified properties, such as "all elements of arrays `a` and `b` are equal" by writing:

```
static_assert(buffer_size(b) == buffer_size(a));
for(i=0; i<buffer_size(a); i++) {
  static_assert(a[i] == b[i]);
}
```

4.6 Experiments

4.6.1 Case Study on Example Benchmarks

To demonstrate the expressiveness of our technique, we evaluate it on 28 challenging array benchmarks available at `http://www.stanford.edu/~tdillig/array.tar.gz` and shown in Figure 4.7. The functions `init` and `init_noncost` initialize all elements of an array to a constant and an iteration-dependent value respectively. `init_partial` initializes part of the array, and `init_even` initializes even positions. `2D_array_init` initializes a 2-dimensional array using a nested loop. The programs labeled `_buggy` exhibit subtle bugs, such as off-by-one errors. Various versions of `copy` copy all, some, or odd elements of an array to another array. `reverse` reverses elements, while `swap` (shown in Figure 4.8) swaps the contents of two arrays. `double_swap` invokes `swap` twice and checks that both arrays are back in their initial state. `strcpy`, `strlen`, and `memcpy` implement the functionality of the standard C library functions and assert their correctness. `find` (resp. `find_first_nonnull`) looks for a specified (resp. non-null) element and returns its index (or -1 if element is not found). `append` appends

the contents of one array to another, and `merge_interleave` interleaves odd and even-numbered elements of two arrays into a result array. The function `alloc_fixed_size` initializes all elements of a double array to a freshly allocated array of fixed size, and then checks that buffer accesses to the element arrays are safe. The function `alloc_nonfixed_size` initializes elements of the double array `a` to freshly allocated arrays of different size, encoded by the elements of another array `b` and checks that accessing indices $[0, \texttt{b}[\texttt{i} - 1]]$ of array $\texttt{a}[\texttt{i}]$ is safe. Compass can automatically verify the full functional correctness of all of the correct programs without any annotations and reports all errors present in buggy programs. To check functional correctness, we add static assertions as described in Section 4.5 and as shown in Figure 4.8.

Figure 4.7 reports for each program the total running time, memory usage (including the constraint solver), number of queries to the SMT solver, and constraint solving time. All experiments were performed on a 2.66 GHz Xeon workstation. We believe these experiments demonstrate that Compass reasons precisely and efficiently about array contents despite being fully automatic. As a comparison, while Compass takes 0.01 seconds to verify the full correctness of `copy`, the approach described in [48] reports a running time of 338.1 seconds, and the counterexample-guided abstraction refinement based approach described in [56] takes 3.65 seconds. Furthermore, our technique is naturally able to verify the correctness of programs that manipulate non-contiguous array elements (e.g., `copy_odd`), as well as programs that require reasoning about arrays inside other arrays (e.g., `alloc_nonfixed_size`). Figure 4.7 also shows that the analysis is memory efficient since none of the programs require more than 2 MB. We believe this to be the case because fluid updates do not create explicit partitions.

4.6.2 Checking Memory Safety on Unix Coreutils Applications

To evaluate the usefulness of our technique, we also check for memory safety errors on five Unix Coreutils applications [14] that manipulate arrays and pointers in complex ways. In particular, we verify the safety of buffer accesses and dereferences with no annotations or false positives, except for two required annotations describing inputs

Program	**Time**	**Memory**	**#Sat queries**	**Solve time**
init	0.01s	< 1 MB	172	0s
init_nonconst	0.02s	< 1 MB	184	0.01s
init_partial	0.01s	< 1MB	166	0.01s
init_partial_buggy	0.02s	< 1 MB	168	0s
init_even	0.04s	< 1 MB	146	0.04s
init_even_buggy	0.04s	< 1 MB	166	0.03s
2D_array_init	0.04s	< 1 MB	311	0.04s
copy	0.01s	< 1 MB	209	0.01s
copy_partial	0.01s	< 1 MB	220	0.01s
copy_odd	0.04s	< 1 MB	243	0.02s
copy_odd_buggy	0.05s	< 1 MB	246	0.05s
reverse	0.03s	< 1 MB	273	0.01s
reverse_buggy	0.04s	< 1 MB	281	0.02s
swap	0.12s	2 MB	590	0.11s
swap_buggy	0.11s	2 MB	557	0.06s
double_swap	0.16s	2 MB	601	0.1s
strcpy	0.07s	< 1 MB	355	0.04s
strlen	0.02s	< 1 MB	165	0.01s
strlen_buggy	0.01s	< 1 MB	89	0.01s
memcpy	0.04s	< 1 MB	225	0.04s
find	0.02s	< 1 MB	119	0.02s
find_first_nonnull	0.02s	< 1 MB	183	0.02s
append	0.02s	< 1 MB	183	0.01s
merge_interleave	0.09s	< 1 MB	296	0.07s
merge_interleave _buggy	0.11s	< 1 MB	305	0.09s
alloc_fixed_size	0.02s	< 1 MB	176	0.02s
alloc_fixed_size_buggy	0.02s	< 1 MB	172	0.02s
alloc_nonfixed_size	0.03s	< 1 MB	214	0.02

Figure 4.7: Case Study

to `main`: `assume(buffer_size(argv) == argc)` and `assume(argv! = NULL)`). Compass is even able to discharge some arbitrary assertions inserted by the original programmers. Some of the buffer accesses that Compass can discharge rely on complex dependencies that are difficult even for experienced programmers to track; see Section 4.7 for an interesting example.

```
void swap(int* a, int* b, int size) {
  for(int i=0; i<size; i++) {
    int t = a[i]; a[i] = b[i]; b[i] = t; }
}
void check_swap(int size, int* a, int* b) {
  int* a_copy = malloc(sizeof(int)*size);
  int* b_copy = malloc(sizeof(int)*size);
  for(int i=0; i<size; i++)  a_copy[i] = a[i];
  for(int i=0; i<size; i++)  b_copy[i] = b[i];
  swap(a, b, size);
  for(i=0; i<size; i++) {
    static_assert(a[i] == b_copy[i]);
    static_assert(b[i] == a_copy[i]);
  }
  free(a_copy); free(b_copy);
}
```

Figure 4.8: Swap Function from Figure 4.7. The static assertions check that all elements of `a` and `b` are indeed swapped after the call to the `swap` function. Compass verifies these assertions automatically in 0.12 seconds.

The chosen benchmarks are challenging for static analysis tools for multiple reasons: First, these applications heavily use arrays and string buffers, making them difficult for techniques that do not track array contents. Second, they heavily rely on path conditions and correlations between scalars used to index buffers. Finally, the behavior of these applications depends on environment choice, such as user input. Our technique is powerful enough to deal with these challenges because it is capable of reasoning about array elements, is path-sensitive, and uses bracketing constraints to capture uncertainty. To give the reader some idea about the importance of these components, 85.4% of the assertions fail if array contents are smashed and 98.2% fail if path-sensitivity is disabled.

As Figure 4.9 illustrates, Compass is able to analyze all applications in under 2 seconds, and the maximum memory used both for the program verification and constraint solving combined is less than 35 MB. We believe these running times and memory requirements demonstrate that the current state of Compass is useful and practical for verifying memory safety in real modest-sized C applications manipulating

Program	Lines	Total Time	Memory	#Sat queries	Solve Time
hostname	304	0.13s	5 MB	1533	0.12s
chroot	371	0.13s	3 MB	1821	0.10s
rmdir	483	1.05s	12 MB	3461	1.02s
su	1047	1.86s	32 MB	6088	1.69s
mv	1151	0.70s	21 MB	7427	0.68s
Total	3356	3.87s	73 MB	20330	3.61

Figure 4.9: Experimental results on Unix Coreutils applications

arrays, pointers, and scalars in complex ways.

4.7 Example from a Real Application

We now discuss an interesting buffer access from the Coreutils `chroot` application that is quite challenging for a human to prove safe. Figure 4.10 presents a simplified slice of the relevant segment of the `chroot` program. Here, our goal is to prove the safety of two buffer accesses marked `Buffer check 1` and `Buffer check 2` in comments. First, observe that there is no buffer access in `main` if `getopt_long` does not return -1 because `usage()` is an exit function, i.e., a call to `usage` terminates the program. Therefore, only the return points 1, 2, and 4 in `getopt_long` could have been taken at points where a buffer is accessed. Second, observe that the if statement marked `Cond 1` exits if `argc <= optind`. Therefore, if program point `(***)` is reached, only the exit point `Return 4` could have been taken. This is because `Return 1` is taken if `argc < 1`; since `optind` is initialized to 0, `Cond 1` would hold and `(***)` could not be reached. Similarly, return point 2 could also not have been taken if `(***)` is reached because return point 2 implies `argc <= optind.` Furthermore, observe that if `getopt_long` returns at return point 4, `optind` is at least 2. Thus, `Buffer check 1` is safe because `argc` is at least 3 inside the if statement marked `Cond 2`. It is now easy to see why `Buffer check 2` is also safe if `Cond 2` holds. If `Cond 2` does not hold, we still need to prove that `argv` has at least one remaining element since `argv` is incremented by `optind + 1` in the else branch. If the else branch is taken, from `Cond 2`, we

have `argc != optind+1`, and from `Cond 1`, we know `argc > optind.` Therefore, `argc` is strictly greater than `optind+1`, and `Buffer check 2` is safe even if the else branch is taken. Compass is able to prove these buffer accesses and many other challenging ones safe fully automatically without any difficulty. As this example demonstrates, the techniques presented in this chapter are not just limited to tracking contents of arrays; they are equally powerful at reasoning about scalar and pointer values.

4.8 Soundness of the Fluid Update Operation

In this section, we present a proof by contradiction of the the fluid update rule from Figure 4.2.

By assumption, $(E, S) \sim (\Sigma, \Gamma)$ before the fluid update, but suppose $(E, S') \not\sim (\Sigma, \Gamma')$ after the fluid update, i.e., there exist two concrete locations $l_s = (s, o)$ and l_t in Σ' and two abstract locations π_s and π_t in Γ' such that $l_s \in \gamma_c(E, S', \pi_s, \sigma_{\mathfrak{I}(\pi_s)})$ and $l_t \in \gamma_c(E, S', \pi_t, \sigma'_{\mathfrak{I}(\pi_t)})$ for some substitutions σ, σ' and one of the following two conditions holds:

1. $S'(l_s, l_t) = \mathit{true}$, but $\mathit{eval}^{+}(\sigma'(\sigma(\Gamma'(\pi_s, \pi_t))), E, S') = \mathit{false}$, or
2. $S'(l_s, l_t) = \mathit{false}$, but $\mathit{eval}^{-}(\sigma'(\sigma(\Gamma'(\pi_s, \pi_t))), E, S') = \mathit{true}$.

Since the arguments for conditions (1) and (2) are symmetric, we focus on disproving (1). To disprove (1), we consider two cases:

1. Either the points-to edge from l_s to l_t was added due to this store instruction, or
2. There was an edge from l_s to l_t before this instruction that was not killed by the store.

We first consider (1). By assumption, $(E, S) \sim (\Sigma, \Gamma)$; hence, π_s must correspond to some π_{2_k} in the array store rule. Furthermore, σ must assign i_k to o, otherwise $l_s \notin \gamma_c(E, S', \pi_{2_k}, \sigma_{\mathfrak{I}(\pi_{2_k})})$. Also, π_t must correspond to some π_{1_j} such that $l_t \in$

$\gamma_c(E, S, \pi_{1_j}, \sigma'_{\mathfrak{I}(\pi_{1_j})})$ and $eval^+(\sigma'(\phi_{1_j}), E, S)$ is *true*. Consider any π_{3_l} that represents the value of $v_3 = o'$ in this execution; for such a π_{3_l}, $eval^+(\phi_{3_l}, E, S)$ must be *true*.

Thus, if an edge from l_s to l_t was added by the current store instruction but $eval^+(\sigma'(\sigma(\Gamma'(\pi_s, \pi_t))), E, S')$ is *false*, then by the argument above and the fluid update rule, it must be the case that:

$$eval^+(\sigma'(\sigma((U(\phi_{2_k}[i'_k - \pi_{3_l}/i'_k]) \wedge \phi_{3_l} \wedge \phi_{1_j}))), E, S) = false$$

From above, we already have $eval^+(\sigma'(\phi_{1_j}), E, S) = true$, and $eval^+(\phi_{3_l}, E, S) = true$. Thus,

$$eval^+(\sigma(U(\phi_{2_k}[i'_k - \pi_{3_l}/i'_k])), E, S) = false \quad (*)$$

must hold for some σ such that $i_k : o$. Consider substitution σ'' that is the same as σ, but all index variables are replaced by their primed counterparts. Clearly, (*) implies:

$$eval^+(\sigma''(\phi_{2_k}[i'_k - \pi_{3_l}/i'_k]), E, S) = false$$

Now consider an assignment that is identical to σ'' but it assigns $o - o'$ to i_k instead of o. Since $S \sim \Gamma$:

$$eval^+(\sigma''[i'_k \leftarrow (o - o')](\phi_{2_k}), E, S) = true$$

because ϕ_{2_k} is the constraint under which the dereference of v_2 is π_{2_k} and $v_3 = o'$. However, this contradicts $eval^+(\sigma''(\phi_{2_k}[i'_k - \pi_{3_l}/i'_k]), E, S) = false$ since $\pi_{3_l} = o'$.

We now consider (2), i.e., suppose there is an existing edge between l_s and l_t that was not killed by this store, but $\sigma'(\sigma(eval^+(\Gamma(\pi_s, \pi_t))), E, S') = false$. As before, π_s corresponds to some π_{2_k} of the store rule, otherwise, none of the constraints on the outgoing edges of π_s could have been weakened in the abstraction. The fluid update weakens constraints by conjoining $\neg(\bigvee_{l,k}(U(\phi_{2_k}[i'_k - \pi_{3_l}/i'_k]) \wedge \phi_{3_l}))$ with existing edge constraints. For the edge between π_s and π_t to be killed, we must have

$$eval^-(\sigma(U(\phi_{2_k}[i'_k - \pi_{3_l}/i'_k]) \wedge \phi_{3_l}), E, S) = true$$

for some l, k. If we construct σ'' from σ as in case (1), clearly:

$$eval^{\perp}(\sigma''(\phi_{2_k}[i'_k - \pi_{3_l}/i'_k] \wedge \phi_{3_l}), E, S) = true$$

As in the previous case, σ must assign i_k to o; otherwise $l_s \notin \gamma_c(E, S', \pi_{2_k}, \sigma_{\mathfrak{I}(\pi_{2_k})})$; hence σ'' assigns i'_k to o. Assume the concrete value of v_3 is o'. If this store did not update l_2 and since $S \sim \Gamma$ before the store, $eval^{\perp}(\sigma''[i'_k \leftarrow (o - o')](\phi_{2_k}), E, S) = false$. Again, this contradicts

$$eval^{\perp}(\sigma''(\phi_{2_k}[i'_k - \pi_{3_l}/i'_k] \wedge \phi_{3_l}), E, S) = true$$

since π_{3_l} must represent the value of o'.

```
int optind = 0;
int getopt_long (int argc, char **argv,...)
{
  if(argc < 1)
     return -1;  /* Return 1 */
  if(optind == 0)
     optind = 1;
  while( skip(argv[optind])
     && optind<argc) optind++;
  if(optind>=argc)
     return -1;  /* Return 2 */
  optind++;
  if(str_prefix(options,
         argv[optind-1])) {
    optarg = argv[optind-1];
    return 0;  /* Return 3 */
  }
  return -1;      /* Return 4 */
}
```

```
int main (int argc, char **argv) {
  if (getopt_long (argc, argv, "+",
      NULL, NULL) != -1) usage (EXIT_FAILURE);

  if (argc <= optind) {  /* Cond 1 */
    error (0, 0, "missing operand");
    usage (EXIT_FAILURE);
  }
  (***)
  if (argc == optind + 1) { /* Cond 2 */
    /* Buffer check 1 */
    static_assert(buffer_size(argv) > 2);
    argv[0] = shell; argv[1] = bad_cast ("-i");
    argv[2] = NULL;
  }
  else argv += optind + 1;
  /* Buffer check 2 */
  static_assert(buffer_size(argv) > 0);
  execvp (argv[0], argv);
}
```

Figure 4.10: A challenging buffer access from `chroot`

Chapter 5

Relational Symbolic Heap

In the previous chapter, we considered an analysis that allows precise reasoning about programs that manipulate heap objects stored inside arrays. Unfortunately, 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 to successfully verify properties of array-manipulating programs. One coarse but accurate intuition is that while the technique proposed so far is good at characterizing array writes, it can still lose information about array reads. To illustrate this point, consider the following code example:

```
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, the technique described in Chapter 4 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` on exit from the loop. 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. The technique described in Chapter 4 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 [73, 22, 48]. 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.

In this chapter, we describe a relational version of the basic technique described in Chapter 4 which avoids 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.

This chapter describes a fully-relational heap analysis that does not construct explicit case splits. More specifically, we develop a relational version of the basic symbolic heap described in Chapter 4 that always enforces 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`. Thus, 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.

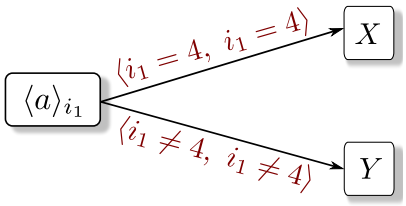


Figure 5.1: An exact symbolic heap

The key insight underlying this chapter is that we can 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` and `y` must have the same value, allowing the assertion to be discharged.

5.1 A Quick Overview

In this section, we give a high-level overview of our approach to relational heap reasoning that directly builds on top of the symbolic heap abstraction described in Chapter 4.

We say that a symbolic heap abstraction of Chapter 4 is *exact* if the over- and underapproximations encoded in the bracketing constraints are identical. Observe that if a symbolic heap is exact, it describes precisely one concrete heap such that the abstraction already encodes the existence and uniqueness of values stored in memory locations. This is the case because a key soundness invariant of the 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.

As an example, consider the symbolic heap shown in Figure 5.1. This symbolic heap 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 5.2. 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

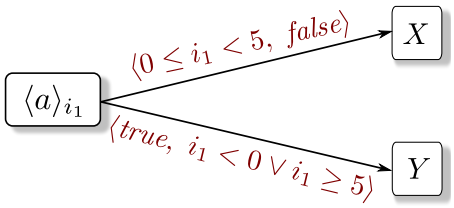


Figure 5.2: An inexact symbolic heap

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 5.1, encodes existence and uniqueness, the normal situation of an imprecise symbolic heap such as the one from Figure 5.2 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 imprecise 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.

5.2 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 review how to retrieve the possible values stored in a location. 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 7 ($\mathbf{\mathit{read}}(\pi, \gamma)$) Let π be an abstract memory location, and let $\gamma = \langle \phi_{may}, \phi_{must} \rangle$ 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. Finally, let ϕ_i'' be obtained by renaming primed (i.e., target's) index variables in ϕ_i' to their unprimed

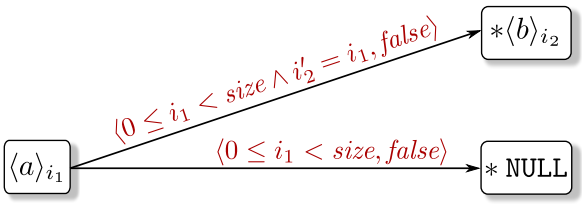


Figure 5.3: A symbolic heap abstraction

counterparts. Then:

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

Example 8 *Consider the heap from Figure 5.3. Here, we have:*

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

5.2.1 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 8 (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:

$$\mathit{VALID}\left(\exists \vec{F}, \vec{F}'. \ \bigvee_{i,j}\left(\begin{array}{l} \pi_i[\vec{F}_i/\vec{I}_i] = \pi'_j[\vec{F}'_j/\vec{I}'_j] \\ \wedge\ \phi_i[\vec{F}_i/\vec{I}_i] \wedge \phi'_j[\vec{F}'_j/\vec{I}'_j] \end{array}\right)\right)$$

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 1 uses the underapproximations $\phi_{i_{must}}, \phi'_{j_{must}}$ such that

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

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 basic symbolic heap described in Chapter 4 does not allow discharging a simple assertion because it does not enforce existence and uniqueness of memory contents in the presence of imprecision:

Example 9 *Consider evaluating the following assertion on the heap from Figure 5.3:*

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

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

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

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.

5.3 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 5.1), 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 9 (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:

$$\mathit{UNSAT}(\exists \vec{I}.\ \phi_{may_i} \wedge \phi_{may_j})$$

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

Thus, if the heap abstraction is not exact, as is often the case, 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 9, 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., $\phi_{may} \not\Leftrightarrow \phi_{must}$) 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).

5.3.1 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_δ^j as follows:

$$\Delta_\delta^j = \begin{cases} \delta_{\pi_s}(i_1, \ldots, i_m) \leq 0 & \text{if } j = 0 \\ \delta_{\pi_s}(i_1, \ldots, i_m) = j & \text{if } 0 < j < k \\ \delta_{\pi_s}(i_1, \ldots, i_m) \geq k & \text{if } j = k \end{cases} \quad \text{where } i_1, \ldots, i_m \in \vec{I_s} \tag{5.1}$$

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,

$$\Delta_\tau^j = \bigwedge_{1 \leq k \leq n} i'_{j_k} = \tau_k(i_1, \ldots, i_m) \tag{5.2}$$

Here, τ_k is an uninterpreted function symbol unique to the k'th index variable of the target. Δ_τ^j 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:

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

Lemma 3 *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.*

Proof 3 *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* $\text{UNSAT}(\Delta_\delta^j[\vec{s}/\vec{i}] \wedge \Delta_\delta^k[\vec{s}/\vec{i}] \wedge j \neq k)$ *for two edges e_j and e_k. For (ii), observe that:*

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

contradicting $\vec{t_1} \neq \vec{t_2}$. Now, we argue why each concrete location $\pi_s[\vec{s}/\vec{i}]$ must have

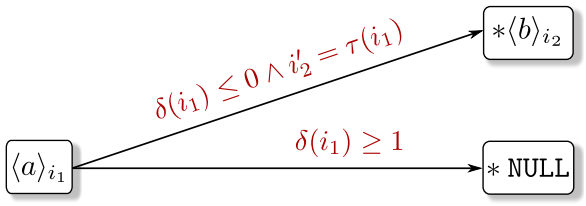


Figure 5.4: The modified version of the heap from Figure 5.3 enforcing existence and uniqueness invariants

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 10 *Consider the heap from Figure 5.3. 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 5.4. Now, consider evaluating the assertion:

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

on this modified heap as described in Section 5.2. As before, the values $V(x)$ *and* $V(y)$ *of* `x` *and* `y` *are given by* $\text{read}(\langle a\rangle_{i_1}, i_1 = 2)$*:*

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

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* $(*\langle b\rangle_{f_1} = *\langle b\rangle_{f_2}) = \text{true}$*. 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.

5.3.2 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 5.3 encodes enough information to prove this assertion, however, the modified heap from Figure 5.4 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 5.2. Then, to preserve the information present in the original heap abstraction, our technique introduces the axioms:

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

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 11 *Consider again the heap from Figure 5.3 and the modified heap from Figure 5.4. Our technique now introduces the following axioms:*

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

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

Now, consider the assertion:

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

As before:

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

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 [47]. In particular, a syntactic instantiation of the axioms for each occurrence of the function term $\delta(\vec{t})$ is sufficient for completeness.

5.3.3 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 9, 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 5.3.

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 10 *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 11 *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:*

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

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 4 *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.*

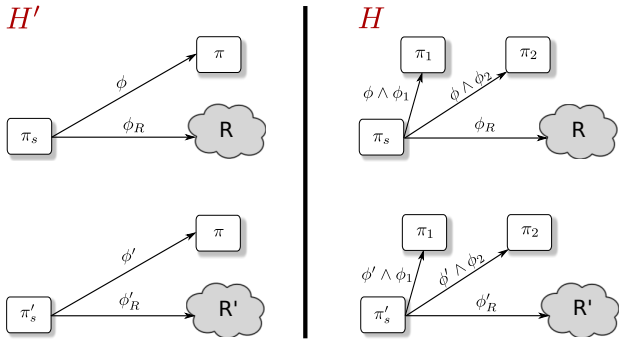


Figure 5.5: Heap H' and H from the proof.

Proof 4 *(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* $\text{assert}(\text{read}(\pi_s, \gamma) = \text{read}(\pi'_s, \gamma'))$ *that is provable in H'. Let* $\text{read}(\pi_s, \phi_s) = \{\ldots, (\pi_i, \phi_i), \ldots\}$ *and* $\text{read}(\pi'_s, \phi'_s) = \{\ldots, (\pi'_i, \phi'_i), \ldots\}$ *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 5.5. 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 5.3.2.)

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:

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

$$\phi \wedge \phi_1 \wedge \phi \wedge \phi_2 = \text{false}$$

$$\phi' \wedge \phi_1 \wedge \phi' \wedge \phi_2 = \text{false}$$

$$\phi_R \vee (\phi \wedge \phi_1) \vee (\phi \wedge \phi_2) = \text{true}$$

$$\phi'_R \vee (\phi' \wedge \phi_1) \vee (\phi' \wedge \phi_2) = \text{true}$$

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*

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

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:*

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

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

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

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

	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 5.6: Experimental results obtained on a single core of a 2.66 GHz Xeon CPU

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

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:

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

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 $(**)$.

	Lines	Combined		Content Only		Mem-Inv Only		Smash	
		FP	Time	FP	Time	FP	Time	FP	Time
hostname	304	0	0.14s	1	0.35s	4	0.35s	6	0.31s
chroot	371	0	0.15s	2	0.61s	3	0.60s	6	0.70s
rmdir	483	0	0.98s	2	1.39s	3	0.66s	4	0.51s
su	1047	0	1.63s	5	1.99s	4	1.62s	16	1.07s
mv	1151	0	0.79s	8	1.48s	4	1.01s	13	1.31s

Figure 5.7: False Positives (abbreviated FP) 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

5.4 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 [9], 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 5.6. 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 configuration, called "Combined", employs the technique described in this chapter, 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 5.3), but does not introduce the axioms described in Section 5.3. The fourth configuration is "Smash", which effectively smashes array contents by neither introducing memory invariants nor tracking the relationship between indices and contents.

As shown in the first column of Figure 5.6, using the technique proposed in this chapter, Compass analyzes OpenSSH in $\sim$ 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 5.6 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 5.6 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 chapter, our analysis can achieve precise relational reasoning without explicitly considering different heap configurations. Second, by using the fluid update operation [38] 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 5.7, 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.

Chapter 6

Analysis of a Language with Containers

So far in this thesis, our discussion has focused only on the precise reasoning of array contents. In this chapter, we extend the techniques described so far to a more general family of data structures, known as *containers*, which provide functionality for inserting, retrieving, removing, and iterating over elements. Examples of containers include maps, lists, vectors, sets, multimaps, deques, as well as their combinations.

We classify containers as either *position-dependent* or *value-dependent*: In position-dependent containers, each element e has a *position* that is used for inserting e into or reading e from the container. Position-dependent containers include vectors and lists, which support inserting and reading elements at a specified position, as well as queues and stacks, which allow inserting and reading elements at the first or last position. In contrast, value-dependent containers expose no notion of position, and each element is added and retrieved using its value. Instances of value-dependent containers include various kinds of maps, sets, bags, and multimaps. For instance, in a map, elements are inserted and looked up using a key; similarly, in a set, elements are inserted and found by the value of their elements rather than a position in the container.

Both kinds of containers are ubiquitous in modern programming, and many languages, such as C++, Java, and C#, provide a standard set of containers that programmers use as basic building blocks for the implementation of other more complex data structures and software. For this reason, successful verification of programs written in higher-level programming paradigms requires a fairly sophisticated understanding of how individual elements are modified as they flow in and out of containers. In fact, even basic safety properties often require reasoning about individual elements stored inside containers:

- To prove that the result of looking up a key k from a map m is non-null, we need to know that an element with key k is present in m and that the value associated with k is non-null.
- In languages with explicit memory management (such as C++), the safety of sequentially deallocating elements in a list or vector depends on the absence of aliasing pointers in the container.

As these examples illustrate, proving even simple properties may require a richer abstraction than treating container contents as sets. In the first example, we need to know not only which values are present in the map, but also which keys are associated with which values. Similarly, the second example requires proving the uniqueness of elements stored at different positions of the container. Hence, successful verification of these properties requires a detailed, per-element understanding of container contents.

We are interested in verifying properties of container-using programs, such as the examples above. We focus on verification of the client program, divorcing checking of the client from the separate problem of verifying the container implementation itself. We believe this separation is advantageous for several reasons:

1. *Understanding the contents of a container does not require understanding the container's implementation.*

   For example, while a map may be implemented as a hash table or a red-black tree, they both export the functionality of associating a key with a value. From the client's perspective, the difference between a hash map and a red-black tree lies primarily in the performance trade-off between various operations.

2. *Verifying container implementations requires different techniques and degrees of automation then verifying their clients.*
   Hence, separating these two tasks allows us to choose the verification techniques best-suited for each purpose. While we might need heavy-weight, semi-automatic approaches for verifying container implementations, we can still develop fully automatic and more scalable techniques for verifying their clients.

3. *There are orders of magnitude more clients of a container than there are container implementations.*
   This fact makes it possible to annotate a handful of library interfaces in order to analyze many programs using these containers.

In this chapter, we describe a precise and fully automatic technique for static reasoning about container contents. By separating the internal implementation of containers from their client-side use, our technique provides a uniform representation and analysis methodology for any position or value-dependent container. Rather than modeling containers as sets of values, our technique provides a per-element understanding of containers, enabling the abstraction to distinguish properties that hold for different elements. Our abstraction naturally models arbitrary nestings of containers, commonly used in real programs. For example, our technique can reason precisely about a map of lists, expressing which lists are associated with which keys, which nested lists are shared or distinct, while also tracking the contents of the nested lists.

6.1 An Informal Overview

To develop a unified representation for containers, we model any container as a function that converts a *key* to an abstract *index* (an integer), which is then mapped to a value at that index. In this abstraction, a key corresponds to any term that is used for inserting an element into or reading an element from the container. For example, in a vector, keys are integers identifying a position in the vector; in a set, keys are the elements that are inserted into the set. For any container, keys are converted

```
1: vector< map<string, int>* > exam_scores;
2:
3: for(int j=0; j<NUM_EXAMS; j++)
4: {
5:  map<string, int>* m = new map<string, int>();
6:  exam_scores.push_back(m);
7: }
8:
9: map<string, vector<int>*>::iterator it =
10:  student_scores.begin();
11: for(; it != student_scores.end(); it++)
12: {
13:  string student = it->first;
14:  vector<int>* scores = it->second;
15:  for(int k=0; k <NUM_EXAMS; k++)
16:  {
17:    (*exam_scores[k])[student] = (*scores)[k];
18:  }
19:}
```

Figure 6.1: Example illustrating key features of the technique

to abstract indices using a *key-to-index mapping*, but this mapping differs between position- and value-dependent containers: For position-dependent containers (such as a vector), the key-to-index mapping is the identity, as the key is the position in the data structure. For value-dependent containers, we leave the function converting keys to indices uninterpreted; clients of value-dependent containers cannot rely on elements being stored in any particular place, just that they are stored somewhere in the container.

A key advantage of introducing an extra level of indirection from keys to indices is that this strategy allows us to treat position- and value-dependent containers uniformly, while providing the ability to differentiate between distinct elements by using integer constraints on the indices. Specifically, we model containers using *indexed locations* of the form $\langle\alpha\rangle_i$ where the index variable i ranges over possible abstract indices of the container. *All* elements in the container are represented by a single abstract location $\langle\alpha\rangle_i$, but constraints on the index variable i allow distinctions to

be made among the different elements of the container. This model generalizes the symbolic heap abstraction that we have developed for reasoning about array contents in Chapter 4 to both position- and value-dependent containers.

This combination of indexed locations and constraints on index variables allows for a much more detailed understanding of containers than representing their contents as a set. For example, if a container c's contents are modeled by the set of values $\{13, 5, 8\}$, this abstraction encodes that any element in c may have *any* of the values 13, 5, and 8, effectively mixing values associated with different elements. On the other hand, by representing c using an indexed abstract location $\langle\alpha\rangle_i$, we can qualify each of the values 13, 5, and 8 by constraints ϕ_1, ϕ_2, and ϕ_3, restricting which indices in $\langle\alpha\rangle_i$ may have which value. The latter abstraction encodes that only those values whose keys are consistent with the index constraint ϕ_i may have value v_i, and thereby retains the correlations between positions and values for position-dependent containers and key-value correlations for value-dependent containers.

To illustrate important features of our technique, consider the C++ code snippet in Figure 6.1. Here, the container `student_scores` maps each student to a vector of integers, indicating the score received by each student on every exam. To keep the example simple, suppose that there are only two students, Tom and Isil, and Tom received scores 76 and 65, and Isil received scores 87 and 72 on two exams. The code in Figure 6.1 builds a reverse mapping `exam_scores` where the i'th element in `exam_scores` is a map from each student to this student's score on the i'th exam.

Figure 6.2 shows a graphical representation of the facts established about the contents of `exam_scores` after analyzing the code from Figure 6.1. In this figure, nodes in the graph represent abstract locations, a directed edge from node A to B qualified by constraint ϕ indicates that B is one of the values stored in A and ϕ constrains at which index of A the value B may be stored. We highlight important features of the abstraction based on Figure 6.2:

1. **Abstract containers:** Observe that the vector `exam_scores` is qualified by an index variable i_1 and the maps nested inside `exam_scores` are also qualified by an index variable i_3. Both of these index variables allow us to select different elements in the container by constraining the values of i_1 and i_3.

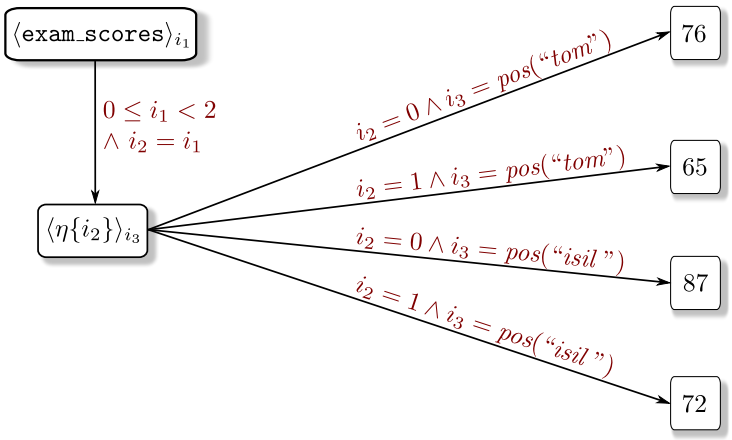


Figure 6.2: The representation of container `exam_scores` after the analysis of code from Figure 6.1

2. **Memory allocations:** A key prerequisite for precise reasoning about nested containers is differentiating different allocations. In the figure, memory locations arising from the allocation at line 5 are described by $\eta\{i_2\}$, where i_2 is also an index variable. Hence, just as we use index variables to differentiate between elements in a container, we also use them for distinguishing different memory allocations arising from the same expression.

3. **Key-to-index mapping:** On the edge from $\langle\eta\{i_2\}\rangle_{i_3}$ to 76, index variable i_3 is equal to *pos(“tom”)*, where *pos* is an invertible, uninterpreted function representing the mapping from key *“tom”* to a unique, but unspecified index. On the other hand, since `exam_scores` is a position-dependent container, the key-to-index mapping is the identity function; hence, the outgoing edge from $\langle$`exam_scores`$\rangle_{i_1}$ is qualified by $0 \le i_1 < 2$.

4. **Nesting of data structures:** On the edge from $\langle$`exam_scores`$\rangle_{i_1}$ to the nested maps modeled by $\langle\eta\{i_2\}\rangle_{i_3}$, i_2 is equal to i_1. This constraint indicates that there is a *unique* allocation for every index of the container `exam_scores` because there is exactly one i_2 for each i_1. Furthermore, together with the constraints on edges outgoing from $\langle\eta\{i_2\}\rangle_{i_3}$, the abstraction encodes that the map stored at position 0 of the vector `exam_scores` associates key *tom* with

value 76, but the map stored at position 1 of the vector associates *tom* with value 65.

5. **Iterators:** The indirection from keys to indices provides a natural way to model iterators by accessing every element in increasing order of their abstract indices. Since the key-to-index mapping is always an invertible function, the abstraction encodes that every element is visited exactly once. This abstraction is also consistent with the expected semantics that iteration order over value-dependent containers is, in general, unspecified (since *pos* is uninterpreted) while elements of position-dependent containers are visited according to their position.

6.2 Language and Concrete Semantics

To formally describe our analysis for container-mainpulating programs, we first introduce a simple statically-typed language:

$$
\begin{array}{lcl}
\textit{Program } P & := & e^{+} \\
\textit{Expression } e & := & v \mid c \mid \texttt{nil} \mid \texttt{new}^{\rho}\ \tau \mid e_1; e_2 \\
& & \mid \texttt{let}^{\rho}\ v : \tau = e\ \texttt{in}\ e' \\
& & \mid v_1.\texttt{read}(v_2) \mid v_1.\texttt{write}(v_2,\ v_3) \\
& & \mid \texttt{foreach}^{\rho_0}\ (v_1^{\rho_1}, v_2^{\rho_2})\ \texttt{in}\ v\ \texttt{do}\ e\ \texttt{od} \\
& & \mid \texttt{if}\ v\ \neq \texttt{nil}\ \texttt{then}\ e_1\ \texttt{else}\ e_2\ \texttt{fi}
\end{array}
$$

A program consists of one or more expressions. Expressions include variables v, non-negative integer constants c, the special constant `nil`, container allocations (`new` τ), sequencing ($e_1; e_2$), and let expressions. A read operation v_1.`read`(v_2) reads the value of element with key v_2 from container v_1, and v_1.`write`($v_2,\ v_3$) writes value v_3 with key v_2 to container v_1. A `foreach` construct iterates over container v, binding the current key to v_1 and the value to v_2. Finally, an `if` expression evaluates expression e_1 or e_2, depending on whether variable v is `nil` or not. The `let`, `new` and `foreach` expressions are labeled with superscripts ρ which are globally unique

expression identifiers. When irrelevant, we omit ρ.

Types in this language are defined by the grammar:

$$\textit{Type}\ \tau := \texttt{Int} \mid \texttt{Nil} \mid \texttt{pos\_adt}(\tau) \mid \texttt{val\_adt}(\tau) \mid \texttt{maybe}(\tau)$$

Base types in this language are `Int` and `Nil`. Position-dependent containers with elements of type τ have type $\texttt{pos\_adt}(\tau)$, and value-dependent containers with value type τ have type $\texttt{val\_adt}(\tau)$. To simplify the technical presentation, we require keys of value-dependent containers to be integers; Section 6.4 discusses how to extend our technique to keys with arbitrary types and custom equality operators. We also introduce a type $\texttt{maybe}(\tau)$ for elements whose type can be either `Nil` or τ. A subtyping relation is defined as:

$$\tau <: \tau \quad \texttt{Nil} <: \texttt{maybe}(\tau) \quad \tau <: \texttt{maybe}(\tau)$$

We write $\texttt{adt}(\tau)$ as shorthand for $\texttt{pos\_adt}(\tau) \vee \texttt{val\_adt}(\tau)$. Type checking rules for this language are given in Figure 6.3.

Observe that this language allows arbitrary nestings of containers because the element type of a container can be another container. Also, while this language does not have explicit `contains` and `remove` operations that are commonly defined on containers, elements can be removed by writing `nil` and the presence of key `k` can be checked by testing whether the result of reading `k` is `nil`.

6.2.1 Operational Semantics

In the operational semantics of our language, we view memory as a two-dimensional array where each row stores a container, and each column identifies the index of a specific element in the container. We model scalar values (integers) as rows where only the 0'th column is used. A concrete memory location is a pair (l, i), where l is the *base location* (i.e., the row) and i is an offset (i.e, the column).

Figure 6.4 gives the operational semantics. The general structure of the rules are of the form $E, S, C \vdash e : l', S'$. Here, environment E maps program variables to base

$$\frac{}{\Gamma \vdash c : \texttt{Int}} \qquad \frac{}{\Gamma \vdash \texttt{nil} : \texttt{Nil}} \qquad \frac{\Gamma(v) = \tau}{\Gamma \vdash v : \tau} \qquad \frac{\tau = \texttt{adt}(\tau') \quad \tau' \neq \texttt{Nil}}{\Gamma \vdash \texttt{new}\ \tau : \tau}$$

$$\frac{\Gamma \vdash v_1 : \texttt{adt}(\tau) \quad \Gamma \vdash v_2 : \texttt{Int}}{\Gamma \vdash v_1.\texttt{read}(v_2) : \texttt{maybe}(\tau)} \qquad \frac{\Gamma \vdash v_1 : \texttt{adt}(\tau) \quad \Gamma \vdash v_2 : \texttt{Int} \quad \Gamma \vdash v_3 : \tau_3,\ \tau_3 <: \texttt{maybe}(\tau)}{\Gamma \vdash v_1.\texttt{write}(v_2, v_3) : \texttt{Nil}}$$

$$\frac{\Gamma \vdash v : \texttt{adt}(\tau) \quad \Gamma[\texttt{Int}/v_1, \tau/v_2] \vdash e : \tau_e}{\Gamma \vdash \texttt{foreach}\ (v_1, v_2)\ \texttt{in}\ v\ \texttt{do}\ e\ \texttt{od} : \texttt{Nil}} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1; e_2 : \tau_2}$$

$$\frac{\Gamma \vdash v : \tau \quad \tau <: \texttt{maybe}(\tau') \quad \Gamma[\tau'/v] \vdash e_1 : \tau'' \quad \Gamma[\texttt{Nil}/v] \vdash e_2 : \tau''}{\Gamma \vdash \texttt{if}\ v\ \neq \texttt{nil}\ \texttt{then}\ e_1\ \texttt{else}\ e_2\ \texttt{fi} : \tau''} \qquad \frac{\Gamma \vdash e : \tau', \quad \tau' <: \tau \quad \Gamma[\tau/v] \vdash e' : \tau''}{\Gamma \vdash \texttt{let}\ v : \tau = e\ \texttt{in}\ e' : \tau''}$$

Figure 6.3: Type checking rules

locations l, store S maps concrete memory locations (l, i) to an integer, identifying another base location or a constant, and C is a vector of integers denoting the current iteration number of each loop in scope. The judgment $E, S, C \vdash e : l', S'$ states that under environment E, store S, and counter vector C, expression e evaluates to value l', producing a new store S'. In Figure 6.4, we use the notation $S \backslash l$ to denote store S with binding l removed. In the rules, we also assume that type environment Γ is available to differentiate between position- and value-dependent containers.

Most of the rules in Figure 6.4 are straightforward; we only highlight important features of the language semantics. There are two key differences between position- and value-dependent containers that our language semantics tries to capture: First, position-dependent containers require filled positions of the container to be contiguous whereas value-dependent containers do not. Second, iteration over position-dependent containers visits elements in increasing order of their position, but iteration over value-dependent containers visits elements in arbitrary order in general.

To capture the first difference, observe that the language semantics requires position-dependent containers to use a contiguous region of memory whereas value-dependent

$$\frac{E(v) = l \quad S(l,0) = l'}{E,S,C \vdash v : l', S} \qquad \frac{}{E,S,C \vdash c : c, S} \qquad \frac{}{E,S,C \vdash \texttt{nil} : \texttt{NIL}, S} \qquad \frac{l_n \notin dom(S) \quad S' = S[\forall i.(l_n, i) \leftarrow \texttt{NIL}]}{E,S,C \vdash \texttt{new}\ \tau : l_n, S'}$$

$$\frac{\begin{array}{l} E,S \vdash e : l, S' \\ E' = E[v \leftarrow l_n] \quad (l_n \notin dom(S')) \\ S'' = S'[(l_n,0) \leftarrow l] \\ E', S'', C \vdash e' : l', S''' \end{array}}{E,S,C \vdash \texttt{let}\ v : \tau = e\ \texttt{in}\ e' : l',\ S'''\backslash l_n} \qquad \frac{\begin{array}{l} E(v_2) = l_2 \quad S(l_2,0) = key \\ E(v_1) = l_1 \quad S(l_1,0) = l_1' \\ S(l_1', key) = l_{res} \end{array}}{E,S,C \vdash v_1.\texttt{read}(v_2) : l_{res}, S}$$

$$\frac{\begin{array}{l} E(v_2) = l_2 \quad S(l_2,0) = key \\ E(v_3) = l_3 \quad S(l_3,0) = val \\ E(v_1) = l_1 \quad S(l_1,0) = l_1' \\ S' = S[(l_1', key) \leftarrow val] \end{array}}{E,S,C \vdash v_1.\texttt{write}(v_2, v_3) : \texttt{NIL}, S'}\ (v_1\texttt{val\_adt}) \qquad \frac{\begin{array}{l} E(v_2) = l_2 \quad S(l_2,0) = pos \\ E(v_3) = l_3 \quad S(l_3,0) = elem \\ E(v_1) = l_1 \quad S(l_1,0) = l_1' \\ S(l_1', pos-1) \neq \texttt{NIL} \quad if\ \ l_3 \neq \texttt{NIL} \wedge pos > 0 \\ S(l_1', pos+1) = \texttt{NIL} \quad if\ \ l_3 = \texttt{NIL} \\ S' = S[(l_1', pos) \leftarrow elem] \end{array}}{E,S,C \vdash v_1.\texttt{write}(v_2, v_3) : \texttt{NIL}, S'}\ (v_1\texttt{pos\_adt})$$

$$\frac{\begin{array}{l} E(v) = l \quad S(l,0) = l' \\ \Delta = \quad [(k_1,\vartheta_1),\ldots,(k_n,\vartheta_n)] \quad where\ \ k_i < k_{i+1} \wedge \\ \qquad (k_i,\vartheta_i) \in \Delta \Leftrightarrow (S(l',k_i) = \vartheta_i \wedge \vartheta_i \neq \texttt{NIL}) \\ \Delta' = \begin{cases} \Delta & if\ \ v\ \texttt{pos\_adt} \\ Permutation(\Delta) & if\ \ v\ \texttt{val\_adt} \end{cases} \\ E' = E[v_1 \leftarrow l_k, v_2 \leftarrow l_v] \quad l_k, l_v \notin dom(S) \\ E', S, (0::C) \vdash process((v_1,v_2)\ \ in\ \ \Delta')\ \ do\ \ e : S' \end{array}}{E,S,C \vdash \texttt{foreach}\ (v_1,v_2)\ \texttt{in}\ v\ \texttt{do}\ e\ \texttt{od} : \texttt{nil}, S'\backslash\{l_k, l_v\}}$$

$$\frac{\begin{array}{l} i < Size(\Delta) \\ E(v_1) = l_k \quad E(v_2) = l_v \\ (k_i,\vartheta_i) = i\text{'th element of}\ \ \Delta \\ S' = S[l_k \leftarrow k_i, l_v \leftarrow \vartheta_i] \\ E, S', (i::C) \vdash e : l_e, S'' \\ E, S'', ((i+1)::C) \vdash process((v_1,v_2)\ \ in\ \ \Delta)\ \ do\ \ e : S''' \end{array}}{E,S,(i::C) \vdash process\ ((v_1,v_2)\ \ in\ \ \Delta)\ \ do\ e : S'''}$$

$$\frac{i = Size(\Delta)}{E,S,(i::C) \vdash process\ ((v_1,v_2)\ \ in\ \ \Delta)\ \ do\ e : S} \qquad \frac{\begin{array}{l} E,S,C \vdash e_1 : l_1, S_1 \\ E,S_1,C \vdash e_2 : l_2, S_2 \end{array}}{E,S,C \vdash e_1; e_2 : l_2, S_2}$$

$$\frac{\begin{array}{l} E(v) = l \quad S(l,0) = l' \\ E,S,C \vdash e_1 : l_r, S' \quad if\ \ l' \neq \texttt{NIL} \\ E,S,C \vdash e_2 : l_r, S' \quad if\ \ l' = \texttt{NIL} \end{array}}{E,S,C \vdash \texttt{if}\ v \neq \texttt{nil}\ \texttt{then}\ e_1\ \texttt{else}\ e_2\ \texttt{fi} : l_r, S'}$$

Figure 6.4: Operational Semantics

containers may be sparse. In particular, it is legal to use any key when writing to a value-dependent container, but for position-dependent containers, the write operation is only defined if it does not create "holes" in the container, i.e., all elements with indices in range $[0, size)$ are non-`nil` and elements with indices at least *size* are `nil`. To capture the second difference, observe, in the `foreach rule`, that Δ is an ordered list of (key, value) pairs, but we construct an arbitrary permutation Δ' of Δ when iterating over value-dependent containers. Also, observe that the (key, value) pairs are pre-computed; hence, any changes to the container during the iteration do not affect the (key, value) pairs that are visited.

6.3 Abstract Semantics

In this section, we describe the abstract semantics that form the basis of our analysis. We first describe our abstract domain (Section 6.3.1) and then discuss our abstract model of containers (Section 6.3.2). In Section 6.3.3, we present the analysis, and in Section 6.3.4, we state the soundness theorem.

6.3.1 Abstract Domain and Preliminaries

Our abstraction differentiates between two kinds of *abstract memory locations*: *Basic locations*, β, represent a single concrete element, and *indexed locations*, $\langle\alpha\rangle_i$, represent containers. Recall that although a single indexed location $\langle\alpha\rangle_i$ represents many concrete elements, our abstraction can reason about individual elements stored in the container by using constraints on the *index variable* i. The *abstract values* used in the analysis are:

$$\begin{array}{lcl} \textit{Abstract value } \pi & = & \mathit{NIL} \mid c \mid \delta \\ \textit{Abstract location } \delta & = & \beta^{\rho} \mid \langle\alpha\rangle_i \\ \textit{Allocation } \alpha & = & \eta_{\rho}\{\vec{i}\} \end{array}$$

Abstract values are *NIL*, integer constants c, basic locations β^{ρ} (where ρ indicates the program point where the location is introduced) and indexed abstract locations $\langle\alpha\rangle_i$. Allocations α are of the form $\eta_{\rho}\{\vec{i}\}$, where ρ is a label for the syntactic allocation

expression $\texttt{new}^{\rho}\tau$. More interestingly, allocations are also qualified by a (potentially empty) vector of index variables to distinguish allocations arising from the same syntactic expression in different loop iterations. Hence, just as index variables allow us to refer to distinct elements in a container, index variables also distinguish allocations arising from the same program expression. Since loops may be nested, the number of index variables in $\eta_{\rho}\{\vec{i}\}$ is equal to the loop nesting depth of a $\texttt{new}^{\rho}\ \tau$ expression.

Unlike the concrete store that maps each concrete location to exactly one concrete value, the *abstract store* necessarily maps each abstract location to a *set* of possible abstract values. An *abstract value set* θ is a set of abstract value (π), bracketing constraint (ϕ) pairs:

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

As before, bracketing constraints ϕ select particular elements from indexed locations. For example, if the abstract value set for a container $\langle\alpha\rangle_i$ is $\{(7, i = 0), (4, i = 1), (\mathit{NIL}, i \geq 2)\}$, the abstraction encodes that the values of elements at indices 0 and 1 are 7 and 4 respectively, but all the other elements are `nil`.

In the rest of this chapter, we assume that an abstract value set θ does not contain two pairs of the form (π, ϕ_1) and (π, ϕ_2); instead, θ contains π only once under $\phi_1 \vee \phi_2$.

6.3.2 Abstract Model of Containers

Our abstraction models any position- or value-dependent container as a mapping from a key to an abstract index to a value stored at this index of the container. In this section, we detail the key-to-index and the index-to-value mappings.

Index Selection: From Keys to Indices

The most important requirement for the key-to-index mapping M for containers is that it obeys the following axiom:

$$\forall i_1, i_2.\ i_1 = i_2 \Rightarrow M^{-1}(i_1) = M^{-1}(i_2)$$

This axiom states that if two abstract indices are equal, then the keys associated with these indices must also be equal. This requirement is necessary for soundness; otherwise, two keys may be mapped to the same index, causing a value associated with key k_1 to be erroneously overwritten through inserts using a different key k_2. Hence, M has the property that its inverse mapping is a function. However, the question remains whether M is itself a function. In this regard, there are two sensible design alternatives:

1. For some containers that allow multiple values for the same key, such as multimaps, we can allow the same key to map to multiple indices such that M itself is not a function.

2. We can require M to be a function and model containers that allow multiple values per key using nested containers.

Without loss of generality, we choose (2) because our model can express arbitrary nestings of containers. Since both M and M^{-1} are functions, the key-to-index mapping is always a bijection (i.e., an invertible function). However, the key-to-index mapping for value-dependent containers differs from that of position-dependent containers: In particular, for value-dependent containers, M is an invertible *uninterpreted* function, while for position-dependent containers, M is the (interpreted) identity function. The intuition behind this choice is that if we insert an element e with key j into a position-dependent container, then e is guaranteed to be the j'th element when iterating over the container. On the other hand, if we insert element e with key j to a value-dependent container, we have no guarantees about where e will appear in the iteration order. Thus, we model the key-to-index mapping of value-dependent containers as an invertible uninterpreted function.

Formally, we define two index selection operators, $\diamondsuit$ and $\clubsuit$, for mapping keys to index constraints for position- and value-dependent containers respectively.

Definition 12 (Index Selection $\diamondsuit$ for Position-Dependent Container) *Let θ_{key} be the set of possible abstract values associated with some key, and let i be an*

index variable. Then,

$$\theta_{\text{key}}\diamondsuit\ i = \bigvee_{(\pi_j,\phi_j)\in\theta_{\text{key}}} (i = \pi_j\ \wedge\ \phi_j)$$

Definition 13 (Index Selection ♣ for Value-Dependent Container) *Let* θ_{key} *be the set of possible abstract values associated with some key, and let* i *be an index variable. Then,*

$$\theta_{\text{key}}\clubsuit\ i = \bigvee_{(\pi_j,\phi_j)\in\theta_{\text{key}}} (i = \text{pos}(\pi_j)\ \wedge\ \phi_j)$$

where pos *is an invertible uninterpreted function.*

Given an abstract value set θ_{key} representing a set of possible keys, the index selectors $\diamondsuit$ and ♣ yield a constraint describing the possible indices associated with θ_{key}. In the definition of $\diamondsuit$, since the mapping M is the identity function, the index variable i is set equal to each possible value π_j of the key (i.e., $i = \pi_j$). On the other hand, in the definition of ♣, the index variable i is equal to an invertible uninterpreted function *pos* of each key (i.e., $i = pos(\pi_j)$). Since the abstract value set associated with the key may contain more than one element, we take the disjunction of the constraints associated with each possible value of the key.

Element Selection: From Indices to Values

We now consider the problem of determining the value associated with a given index. More specifically, given an abstract value set θ associated with a container, we want to determine which elements of θ are consistent with some index constraint ϕ.

We begin with a simple example: Suppose that the abstract value set θ for a container $\langle\alpha\rangle_i$ is $\{(8, i = 1), (5, i = 2), (NIL, i > 2))\}$ and we want to determine the possible values of the element at index 2 in the container. To do this, we can substitute 2 for index variable i and remove all unsatisfiable elements from θ, which yields 5 as the only possible value for this element. We formalize this concept using an *element selection* operation $\bowtie$:

Definition 14 (Element Selection $\bowtie$) *Let I denote the set of index variables mentioned in constraint ϕ, and let* QE *define a quantifier elimination procedure. Then,*

$$\theta \bowtie \phi = \left\{ (\pi_j, \phi'_j) \;\middle|\; \begin{array}{l} (\pi_j, \phi_j) \in \theta \wedge \mathrm{SAT}(\phi_j \wedge \phi) \wedge \\ \phi'_j = \mathrm{QE}(\exists I.\ (\phi_j \wedge \phi)) \end{array} \right\}$$

First, observe that the element selection operation $\bowtie$ filters out elements in θ inconsistent with ϕ because of the requirement $SAT(\phi_j \wedge \phi)$. Second, observe that the resulting constraint ϕ'_j is obtained by existentially quantifying and then subsequently eliminating all index variables used in ϕ from the constraint $\phi_j \wedge \phi$ because $\phi'_j = \mathrm{QE}(\exists I.\ (\phi_j \wedge \phi))$. Existential quantifier elimination generalizes the simple substitution mechanism we sketched out informally in the example: Since the index constraint ϕ is not always a simple equality, we may not be able to substitute concrete values for the index variables. Hence, similar to Definition 2 from Chapter 4, we need to use existential quantification in the general case.

Example 12 *Consider the abstract value set*

$$\theta = \left\{ \begin{array}{l} (0, \langle 0 \leq i \leq 10, \mathrm{false} \rangle), (1, \langle 0 \leq i \leq 10, \mathrm{false} \rangle), \\ (\mathrm{NIL}, \langle i > 10, i > 10 \rangle) \end{array} \right\}$$

associated with container $\langle \alpha \rangle_i$. To determine the possible values of those elements whose indices in the container are in the range $[0, 2]$, we compute:

$$\theta \bowtie (0 \leq i \leq 2) \;=\; \{(0, \langle \mathrm{true}, \mathrm{false} \rangle), (1, \langle \mathrm{true}, \mathrm{false} \rangle)\}$$

The resulting set encodes that the possible values of elements in the range $[0, 2]$ are either 0 or 1, but definitely not NIL.

6.3.3 The Analysis

We describe the analysis as deductive rules of the form:

$$\mathbb{E}, \mathbb{S}, \mathbb{C} \vdash e : \theta, \mathbb{S}'$$

$$(1)\quad \frac{\theta = \{NIL, true\}}{\mathbb{E}, \mathbb{S}, \mathbb{C} \vdash \texttt{nil} : \theta, \mathbb{S}} \qquad (2)\quad \frac{\theta = \{c, true\}}{\mathbb{E}, \mathbb{S}, \mathbb{C} \vdash c : \theta, \mathbb{S}}$$

$$(3)\quad \frac{\begin{array}{c}\mathbb{E}(v) = \beta \\ \mathbb{S}(\beta) = \theta\end{array}}{\mathbb{E}, \mathbb{S}, \mathbb{C} \vdash v : \theta, \mathbb{S}} \qquad (4)\quad \frac{\begin{array}{c}\mathbb{E}, \mathbb{S}, \mathbb{C} \vdash e_1 : \theta_1, \mathbb{S}_1 \\ \mathbb{E}, \mathbb{S}_1, \mathbb{C} \vdash e_2 : \theta_2, \mathbb{S}_2\end{array}}{\mathbb{E}, \mathbb{S}, \mathbb{C} \vdash e_1; e_2 : \theta_2, \mathbb{S}_2}$$

$$(5)\quad \frac{\begin{array}{c}\mathbb{E}, \mathbb{S}, \mathbb{C} \vdash e : \theta, \mathbb{S}' \\ \mathbb{E}[v \leftarrow \beta^\rho], \mathbb{S}'[\beta^\rho \leftarrow \theta], \mathbb{C} \vdash e' : \theta', \mathbb{S}''\end{array}}{\mathbb{E}, \mathbb{S}, \mathbb{C} \vdash \texttt{let}^\rho\ v : \tau = e\ \texttt{in}\ e' : \theta', \mathbb{S}'' \backslash \beta^\rho}$$

$$(6)\quad \frac{\begin{array}{c}\mathbb{E}(v) = \beta \quad \mathbb{S}(\beta) = \theta \\ \phi_{nil} = \bigvee_{(\pi_j, \phi_j) \in \theta} ((\pi_j = NIL)\ \wedge\ \phi_j) \\ \mathbb{E}, \mathbb{S}, \mathbb{C} \vdash e_1 : \mathbb{S}_1 \quad \mathbb{E}, \mathbb{S}, \mathbb{C} \vdash e_2 : \mathbb{S}_2 \\ \mathbb{S}' = (\mathbb{S}_1 \wedge \neg\phi_{nil}) \sqcup (\mathbb{S}_2 \wedge \phi_{nil})\end{array}}{\mathbb{E}, \mathbb{S}, \mathbb{C} \vdash \texttt{if}\ v \neq \texttt{nil}\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \mathbb{S}'}$$

Figure 6.5: Transformers not Directly Related to Containers

where $\mathbb{E}, \mathbb{S}$, and $\mathbb{C}$ are the abstract counterparts of the E, S, C environments used in the concrete semantics. In particular, the *abstract environment* $\mathbb{E}$ maps program variables to basic locations β^ρ, the *abstract store* $\mathbb{S}$ maps abstract memory locations δ to abstract value sets θ, and, finally, the counter vector $\mathbb{C}$ (a vector of integers) is used for distinguishing different loop iterations.

We present the analysis in three steps: First, we discuss the basic transformers not directly related to containers (Figure 6.5), then we describe the abstract semantics for reading from, writing to, and allocating containers (Figure 6.6), and, finally, we give the abstract semantics of the `foreach` construct (Figure 6.7).

Most of the transformers presented in Figure 6.5 are straightforward; we only discuss rule (6) in detail. In this rule, ϕ_{nil} describes under what condition v is *NIL*. After independently analyzing the then and else branches, we obtain the resulting abstract store $\mathbb{S}'$ by conjoining $\mathbb{S}_1$ and $\mathbb{S}_2$ with $\neg\phi_{nil}$ and ϕ_{nil} respectively and then taking their union.

In this rule, we use the notation $\mathbb{S} \wedge \phi$ as shorthand for the operation that conjoins

ϕ with every constraint in $\mathbb{S}$:

$$\forall \delta \in dom(\mathbb{S}).\ \ (\mathbb{S} \wedge \phi)(\delta) = \{(\pi_j, \phi_j \wedge \phi) \mid (\pi_j, \phi_j) \in \mathbb{S}(\delta)\}$$

Rule (6) also uses a join operation on abstract stores defined as:

$$\forall \delta \in (dom(\mathbb{S}_1) \cup dom(\mathbb{S}_2)).\ \ (\pi, \phi) \in (\mathbb{S}_1 \sqcup \mathbb{S}_2)(\delta) \ \Leftrightarrow$$
$$(\pi, \phi_1) \in \mathbb{S}_1(\delta) \wedge (\pi, \phi_2) \in \mathbb{S}_2(\delta) \wedge \phi = \phi_1 \vee \phi_2$$

In this definition, we require that every abstract value π that is present in either $\mathbb{S}_1$ or $\mathbb{S}_2$ is also present in the other; if it is not explicitly there, we add it under constraint *false*.

Abstract Semantics for Container Operations

We now consider the abstract semantics for reading from containers, presented in Figure 6.6. In the first two rules of Figure 6.6, θ_2 represents the abstract value set for the key v_2, and each of the elements $\langle\alpha\rangle_{i_j}$ in abstract value set θ_1 are containers that the read operation may be performed on. We perform the key-to-index mapping using the $\diamondsuit$ operator for position-dependent containers and the $\clubsuit$ operator for value-dependent containers, as described in Section 6.3.2. In these rules, the constraints $((\theta_2 \diamondsuit i_j) \wedge \phi_j)$ and $((\theta_2 \clubsuit i_j) \wedge \phi_j)$ describe the positions in container $\langle\alpha\rangle_{i_j}$ from which we read the value. Finally, we perform the index-to-value mapping using the $\bowtie$ operation; the abstract value set θ describes all possible elements that may be obtained as a result of the read.

The now consider the rules in Figure 6.6 that describe the abstract semantics for writing to containers. The helper rule Newval computes the new abstract value set associated with container $\langle\alpha\rangle_i$ after writing θ_w at those indices of $\langle\alpha\rangle_i$ described by constraint ϕ_w. Since θ_w is written to only those locations that satisfy the index constraint ϕ_w, we conjoin ϕ_w with each element in θ_w to obtain θ'_w. Now, those elements in container $\langle\alpha\rangle_i$ that do not satisfy the index constraint ϕ_w are not modified by the write; hence the existing values $\mathbb{S}(\langle\alpha\rangle_i)$ are preserved under condition $\neg\phi_w$. Thus, θ_p represents all values in $\langle\alpha\rangle_i$ that are not affected by the write. Finally, the

Read from Position Dependent Container

$$\frac{\mathbb{E}(v_1) = \beta_1 \quad \mathbb{E}(v_2) = \beta_2 \quad \mathbb{S}(\beta_1) = \theta_1 \quad \mathbb{S}(\beta_2) = \theta_2 \quad \theta = \{\mathbb{S}(\langle\alpha\rangle_{i_j}) \bowtie ((\theta_2 \diamondsuit i_j) \wedge \phi_j) \mid (\langle\alpha\rangle_{i_j}, \phi_j) \in \theta_1)\}}{\mathbb{E}, \mathbb{S}, \mathbb{C} \vdash v_1.\texttt{read}(v_2) : \theta, \mathbb{S}}$$

Read from Value Dependent Container

$$\frac{\mathbb{E}(v_1) = \beta_1 \quad \mathbb{E}(v_2) = \beta_2 \quad \mathbb{S}(\beta_1) = \theta_1 \quad \mathbb{S}(\beta_2) = \theta_2 \quad \theta = \{\mathbb{S}(\langle\alpha\rangle_{i_j}) \bowtie ((\theta_2 \clubsuit i_j) \wedge \phi_j) \mid (\langle\alpha\rangle_{i_j}, \phi_j) \in \theta_1)\}}{\mathbb{E}, \mathbb{S}, \mathbb{C} \vdash v_1.\texttt{read}(v_2) : \theta, \mathbb{S}}$$

Newval

$$\frac{\theta'_w = \{(\pi_j, \phi_w \wedge \phi_j) \mid (\pi_j, \phi_j) \in \theta_w\} \quad \theta_p = \{(\pi_k, \neg\phi_w \wedge \phi_k) \mid (\pi_k, \phi_k) \in \mathbb{S}(\langle\alpha\rangle_i)\}}{\mathbb{S} \vdash \mathit{newval}(\langle\alpha\rangle_i, \theta_w, \phi_w) : \theta'_w \cup \theta_p}$$

Update

$$\frac{\begin{array}{l}\theta_c = \{(\langle\alpha\rangle_{i_1}, \phi_1), \ldots, (\langle\alpha\rangle_{i_k}, \phi_k)\} \\ \mathbb{S} \vdash \mathit{newval}(\langle\alpha\rangle_{i_1}, \theta_{val}, (\theta_{key} \otimes i_1) \wedge \phi_1) : \theta_1 \\ \ldots \\ \mathbb{S} \vdash \mathit{newval}(\langle\alpha\rangle_{i_k}, \theta_{val}, (\theta_{key} \otimes i_k) \wedge \phi_k) : \theta_k \\ \mathbb{S}' = \mathbb{S}[\langle\alpha\rangle_{i_1} \leftarrow \theta_1, \ldots, \langle\alpha\rangle_{i_k} \leftarrow \theta_k]\end{array}}{\mathbb{S} \vdash \mathit{update}(\theta_c, \theta_{key}, \theta_{val}) \quad \mathit{with} \quad \otimes : \mathbb{S}'} (\otimes \in \{\diamondsuit, \clubsuit\})$$

Write to Value Dependent Container

$$\frac{\mathbb{E}(v_1) = \beta_1 \quad \mathbb{E}(v_2) = \beta_2 \quad \mathbb{E}(v_3) = \beta_3 \quad \mathbb{S}(\beta_1) = \theta_1 \quad \mathbb{S}(\beta_2) = \theta_2 \quad \mathbb{S}(\beta_3) = \theta_3 \quad \mathbb{S} \vdash \mathit{update}(\theta_1, \theta_2, \theta_3) \ \mathit{with} \ \clubsuit : \mathbb{S}'}{\mathbb{E}, \mathbb{S}, \mathbb{C} \vdash v_1.\texttt{write}(v_2, v_3) : \{(\mathit{NIL}, \mathit{true})\}, \mathbb{S}'}$$

Write to Position Dependent Container

$$\frac{\mathbb{E}(v_1) = \beta_1 \quad \mathbb{E}(v_2) = \beta_2 \quad \mathbb{E}(v_3) = \beta_3 \quad \mathbb{S}(\beta_1) = \theta_1 \quad \mathbb{S}(\beta_2) = \theta_2 \quad \mathbb{S}(\beta_3) = \theta_3 \quad \mathbb{S} \vdash \mathit{update}(\theta_1, \theta_2, \theta_3) \ \mathit{with} \ \diamondsuit : \mathbb{S}'}{\mathbb{E}, \mathbb{S}, \mathbb{C} \vdash v_1.\texttt{write}(v_2, v_3) : \{(\mathit{NIL}, \mathit{true})\}, \mathbb{S}'}$$

Container Allocation

$$\frac{\alpha = \eta_\rho\{\vec{i^\rho}\}, \quad \vec{i^\rho} = [i_1^\rho, \ldots, i_n^\rho] \ \mathit{where} \ \ n = |\mathbb{C}| \quad \mathbb{S} \vdash \mathit{newval}(\langle\alpha\rangle_{i_0^\rho}, \{(\mathit{NIL}, \mathit{true})\}, \vec{i^\rho} = \mathbb{C}) : \theta}{\mathbb{E}, \mathbb{S}, \mathbb{C} \vdash \texttt{new}^\rho \ \tau : \{(\langle\alpha\rangle_{i_0^\rho}, \vec{i^\rho} = \mathbb{C})\}, \mathbb{S}[\langle\alpha\rangle_{i_0^\rho} \leftarrow \theta]}$$

Figure 6.6: Abstract Semantics for Container Operations

set of new values stored in container $\langle\alpha\rangle_i$ is obtained by taking the union of θ'_w (i.e.,the new value for the updated indices) and θ_p (i.e., values stored at all other indices).

The second helper rule, Update, uses Newval to compute the new abstract store after a write. In this rule, each element $\langle\alpha\rangle_{i_j}$ in θ_c represents a container that may be written to. The value set θ_{val} describes the possible values that may be written, and the constraint $((\theta_{key} \otimes i_j) \wedge \phi_j)$ (where $\otimes$ is either $\clubsuit$ or $\diamondsuit$) describes those indices of $\langle\alpha\rangle_{i_j}$ that are modified. For each container $\langle\alpha\rangle_{i_j}$ in θ_c, the Newval rule is invoked to compute the new value set θ_j after the write, and a new store $\mathbb{S}'$ is obtained by binding each $\langle\alpha\rangle_{i_j}$ to its new value set θ_j. The write rules for position- and value-dependent containers use the Update rule to compute the new abstract store after the write. As expected, the rule for position-dependent containers uses the $\diamondsuit$ operator while the rule for value-dependent containters uses $\clubsuit$. [1]

The last rule in Figure 6.6 describes the abstract semantics for container allocations. The abstract location arising from the allocation is labeled with the expression identifier ρ to differentiate allocation sites, and the vector of index variables $\vec{i^\rho}$ differentiates allocations arising from the same syntactic expression in different loop iterations. Since the counter vector $\mathbb{C}$ has as many entries as the loop nesting depth of the allocation expression, the number of variables in $\vec{i^\rho}$ is equal to the number of entries in $\mathbb{C}$. Observe that, in this rule, the constraint $\vec{i^\rho} = \mathbb{C}$ stipulates that each index variable in $\vec{i^\rho}$ is equal to the appropriate counter describing the iteration number of a loop. Finally, recall that the concrete semantics initializes the entries in a freshly allocated container to `NIL`, hence, the Newval rule is invoked to compute the new value set associated with container $\langle\alpha\rangle_{i_0^\rho}$ after initializing its elements to *NIL*.

Example 13 *Consider the simple program:*

```
1:  let^a v:  val_adt(Int) = new^b val_adt(Int) in
2:    v.write(4, 87),
3:      let x = v.read(4) in
```

[1] Recall that the operational semantics are undefined if position-dependent containers are not used contiguously. Since checking this correct usage condition is an orthogonal problem to reasoning about container contents, the abstract semantics reason only about programs for which the operational semantics do not get "stuck".

```
4:        let y = v.read(3) in nil
```

Assume $\mathbb{E}(\mathtt{v}) = \beta^a$. *The abstract store after line 1 is given by:*

$$\mathbb{S} : [\beta^a \to \{\langle \eta_b \rangle_i, \text{true})\}, \quad \langle \eta_b \rangle_i \to \{(\text{NIL}, \text{true})\}]$$

Here, η_b *does not have any index variables because the allocation expression is not in a loop; the index variable* i *in* $\langle \eta_b \rangle_i$ *ranges over indices of the container. After the write at line 2, we have:*

$$\mathbb{S} : \left[\begin{array}{lcl} \beta^a & \to & \{\langle \eta_b \rangle_i, \text{true})\}, \\ \langle \eta_b \rangle_i & \to & \{(87, i = \text{pos}(4)), (\text{NIL}, i \neq \text{pos}(4))\} \end{array} \right]$$

At line 3, the abstract value set for `x` *is:*

$$\begin{array}{rcl} \mathbb{S}(\langle \eta_b \rangle_i) \bowtie (i \clubsuit 4) & = & \mathbb{S}(\langle \eta_b \rangle_i) \bowtie (i = \text{pos}(4)) \\ & = & \{(87, \text{true})\} \end{array}$$

Similarly, at line 4 the abstract value set for `y` *is:*

$$\mathbb{S}(\langle \eta_b \rangle_i) \bowtie (i \clubsuit 3) \quad = \quad \{(\text{NIL}, \text{true})\}$$

Abstract Semantics for Iteration

The main idea behind the abstract semantics for iterating over containers is that *the j'th iteration of the loop accesses the key and value pairs stored at the j'th index* of the container. It is easy to see that this strategy is correct for position-dependent containers because (i) the concrete semantics requires an element with key j to be accessed during the j'th iteration and (ii) in our abstraction, the key-to-index mapping for position-dependent containers is the identity function. For value-dependent containers, recall that the operational semantics stipulates an arbitrary iteration order. Now, although the abstraction models iteration by visiting the element at the j'th index during the j'th iteration, it does not impose any restrictions on which key may be visited during the j'th iteration because the key-to-index mapping is an

uninterpreted function (the constraint $j = pos(k)$ is satisfiable for any value of j and any key k). Furthermore, since *pos* is an invertible function, the abstraction encodes that for each different value of j, there is a different key k, indicating that no key may be visited multiple times.

Figure 6.7 gives the abstract semantics of the `foreach` construct. The first two rules compute the set of (key, value) pairs that may be visited during an arbitrary k'th iteration of the loop for position- and value-dependent containers respectively. Since the abstract semantics models iteration as visiting the k'th index during the k'th iteration, we retrieve the values stored in container $\langle\alpha\rangle_{i_j}$ under the index constraint $i_j = k$. Therefore, in the first two rules, the abstract value set θ_v describes the values that may be stored at index k. For position-dependent containers, the key during the k'th iteration of the loop is bound to k, as required by the operational semantics. In the first rule, we construct the set of possible key, value pairs for the k'th iteration as the set of all (k, π_v) such that π_v is non-nil and in θ_v. Observe that the (key, value) pairs in $\theta_{k\times v}$ respect the relationship between keys (i.e., positions) and values, as illustrated by the following example:

Example 14 *Consider a position-dependent container* $\langle\alpha\rangle_i$ *such that* $\mathbb{S}(\langle\alpha\rangle_i) = \{(44, i = 0), (3, i = 1), (\text{NIL}, i \geq 2)\}$. *We compute the set of (key, value) pairs during the k'th iteration as:*

$$\theta_{k\times v} = \{((k, 44), k = 0), ((k, 3), k = 1)\}$$

Observe that the abstraction respects the relationship between positions and values; for example, the pair (1, 44) is infeasible.

The second rule in Figure 6.7 computes the (key, value) pairs during the k'th iteration for value-dependent containers. In this rule, the key during the k'th iteration is bound to all integers π_k such that $k = pos(\pi_k)$, as stipulated by constraint ϕ_k.[2] As in the position-dependent case, the relationship between keys and values are preserved because the rule filters out infeasible (key, value) pairs by checking the satisfiability of ϕ_{kv}. Finally, observe that the *pos* function is renamed to pos^{ρ} because elements

[2]Observe that the set of all possible π_k's is finite for any given program in our language; hence candidates for π_k are drawn from a finite set.

Key, value pair at kth Iteration for pos_adt

$$\frac{\begin{array}{l}\mathbb{E}(v) = \beta \quad \mathbb{S}(\beta) = \theta_c \\ \theta_v = \{\mathbb{S}(\langle\alpha\rangle_{i_j}) \bowtie (i_j = k \wedge \phi_j) \mid (\langle\alpha\rangle_{i_j}, \phi_j) \in \theta_c\} \\ \theta_{k\times v} = \left\{((k, \pi_v), \phi_v) \;\middle|\; (\pi_v, \phi_v) \in \theta_v \wedge \pi_v \neq \mathit{NIL})\ \right\}\end{array}}{\vdash \mathit{elem}^{\rho}(v)@k : \theta_{k\times v}}$$

Key, value pair at kth Iteration for val_adt

$$\frac{\begin{array}{l}\mathbb{E}(v) = \beta \quad \mathbb{S}(\beta) = \theta_c \\ \theta_v = \{\mathbb{S}(\langle\alpha\rangle_{i_j}) \bowtie (i_j = k \wedge \phi_j) \mid (\langle\alpha\rangle_{i_j}, \phi_j) \in \theta_c\} \\ \theta_{k\times v} = \left\{((\pi_k, \pi_v), \phi_{kv}) \;\middle|\; \begin{array}{l}(\pi_v, \phi_v) \in \theta_v \wedge \pi_v \neq \mathit{NIL} \\ \wedge\ \phi_k = ((\mathit{pos}(\pi_k) = k) \\ \wedge\ (\phi_{kv} = (\phi_k \wedge \phi_v)[\mathit{pos}^{\rho}/\mathit{pos}]) \\ \wedge\ \mathit{SAT}(\phi_{kv})\end{array}\right\}\end{array}}{\vdash \mathit{elem}^{\rho}(v)@k : \theta_{k\times v}}$$

Foreach

$$\frac{\begin{array}{l}\vdash \mathit{elem}^{\rho_0}(v)@k^{\rho_0} : \theta_{kv} \\ \theta_{key} = \{(\pi_{key}, \phi) \mid ((\pi_{key}, \pi_{val}), \phi) \in \theta_{kv}\} \\ \theta_{val} = \{(\pi_{val}, \phi) \mid ((\pi_{key}, \pi_{val}), \phi) \in \theta_{kv}\} \\ \mathbb{E}' = \mathbb{E}[v_1 \leftarrow \beta^{\rho_1}, v_2 \leftarrow \beta^{\rho_2}] \qquad \mathbb{S}' = \mathbb{S}[\beta^{\rho_1} \leftarrow \theta_{key}, \beta_2^{\rho_2} \leftarrow \theta_{val}] \\ \mathbb{E}', \mathbb{S}', (0{::}\mathbb{C}) \vdash \mathit{fix}\ (e, k^{\rho}) : \mathbb{S}'' \qquad \theta = \{(\mathit{NIL}, \mathit{true})\}\end{array}}{\mathbb{E}, \mathbb{S}, \mathbb{C} \vdash \texttt{foreach}^{\rho_0}(v_1^{\rho_1}, v_2^{\rho_1})\ \texttt{in}\ v\ \texttt{do}\ e\ \texttt{od} : \theta, \mathbb{S}''\backslash\{\beta^{\rho_1}, \beta^{\rho_2}\}}$$

Fix

$$\frac{\begin{array}{l}\mathbb{E}, \mathbb{S}[c/k], (c{::}\mathbb{C}) \vdash e : \theta, \mathbb{S}', \qquad \mathbb{S}' \sqsubseteq \mathbb{S}^* \\ \mathbb{E}, \mathbb{S}^*, ((c+1){::}\mathbb{C}) \vdash \mathit{fix}\ (e, k) : \mathbb{S}^*\end{array}}{\mathbb{E}, \mathbb{S}, (c{::}\mathbb{C}) \vdash\ \mathit{fix}\ (e, k) : \mathbb{S}^*}$$

Figure 6.7: Abstract Semantics for Iterating over Containers

may be visited in a different order in each loop.

The `foreach` rule first invokes the appropriate helper *elem* rule for computing the set of (key, value) pairs during an arbitrary k^{ρ_0}'th iteration. (The variable k is superscripted with the expression identifier ρ_0 for this loop in order to avoid naming conflicts.) The set θ_{kv} therefore describes the set of possible (key, value) pairs during an arbitrary iteration. The abstract value sets θ_{key} and θ_{val} are obtained by selecting the keys and the values from θ_{kv} respectively. The abstract environment $\mathbb{E}'$ binds variables v_1, v_2 to fresh locations β^{ρ_1} and β^{ρ_2}, and the abstract store $\mathbb{S}'$ binds β^{ρ_1} and β^{ρ_2} to θ_{key} and θ_{val}, since the operational semantics requires the (key, value) pairs to be computed before executing the body of the `foreach` construct. The `foreach` rule uses the helper *fix* rule to obtain the final store $\mathbb{S}''$.

In the *fix* (e, k) rule, c represents the current iteration number of the loop. Since the bindings for v_1 and v_2 are parametric on variable k, the rule replaces occurrences of k in $\mathbb{S}$ with concrete value c when evaluating the loop body e. In this rule, $\mathbb{S}^*$ is a sound store describing the cumulative effect of the loop, as $\mathbb{S}^*$ overapproximates the store after any loop iteration. Here, an abstract store $\mathbb{S}'$ overapproximates another abstract store $\mathbb{S}$, written $\mathbb{S} \sqsubseteq \mathbb{S}'$ according to Definition 16:

Definition 15 *(***Domain Extension** $\mathbb{S}_{\mapsto\mathbb{S}'}$*) 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: Let δ be any binding in $\mathbb{S}'$ and let (π_i, ϕ_i) be any element of $\mathbb{S}'(\delta)$.*

1. *If $\delta \in \mathbb{S} \wedge (\pi_i, \phi_i') \in \mathbb{S}(\delta)$, then $\delta \in \mathbb{S}_{\mapsto\mathbb{S}'} \wedge (\pi_i, \phi_i') \in \mathbb{S}_{\mapsto\mathbb{S}'}(\delta)$*
2. *Otherwise, $\delta \in \mathbb{S}_{\mapsto\mathbb{S}'} \wedge (\pi_i, \text{false}) \in \mathbb{S}_{\mapsto\mathbb{S}'}(\delta)$*

Definition 16 *(***Abstract Store Overapproximation** $\mathbb{S} \sqsubseteq \mathbb{S}'$*) Let $\mathbb{S}_1$ be the domain extension $\mathbb{S}_{\mapsto\mathbb{S}'}$, and let $\mathbb{S}_2$ be the domain extension $\mathbb{S}'_{\mapsto\mathbb{S}}$. Then, $\mathbb{S} \sqsubseteq \mathbb{S}'$ if for all $\delta \in \mathbb{S}_1$ and for all π_i such that $(\pi_i, \langle\varphi_{\text{may}}, \varphi_{\text{must}}\rangle) \in \mathbb{S}_1(\delta)$ and $(\pi_i, \langle\varphi'_{\text{may}}, \varphi'_{\text{must}}\rangle) \in \mathbb{S}_2(\delta)$:*

$$\varphi_{\text{may}} \Rightarrow \varphi'_{\text{may}} \quad \wedge \quad \varphi'_{\text{must}} \Rightarrow \varphi_{\text{must}}$$

According to this definition, a store $\mathbb{S}'$ overapproximates another abstract store $\mathbb{S}$ if, when they are extended to the same domain, any may constraint in $\mathbb{S}$ implies the

corresponding may constraint in $\mathbb{S}'$, and any must constraint in $\mathbb{S}$ is implied by the corresponding must constraint in $\mathbb{S}'$. In other words, the overapproximation encoded in $\mathbb{S}'$ through the may constraints is *more* permissive than $\mathbb{S}$, and the underapproximation encoded by $\mathbb{S}'$ through the must constraints is *less* permissive than $\mathbb{S}$.

In the *fix* rule, it is easy to see that a trivial invariant store $\mathbb{S}^*$ always exists since the analysis creates a finite number of abstract locations for any given program, and an abstract store $\mathbb{S}_{triv}$ with constraint $\langle true, false \rangle$ mapping each possible abstract location to any other abstract location has the property $\forall \mathbb{S}.\ \mathbb{S} \sqsubseteq \mathbb{S}_{triv}$. To find a more useful invariant store than the trivial $\mathbb{S}_{triv}$, it is necessary to infer numeric invariants relating index variables associated with different containers or allocation sites. Since the focus of this chapter is not invariant generation, we do not go into the details of how to find a "good" invariant store; various techniques based on abstract interpretation [31, 58] and quantifier elimination [38, 59] can be used for finding invariants. In particular, Section 4.4 from Chapter 4 presents an algorithmic way of finding such invariants in this domain.

An Example Illustrating Key Features of the Analysis

In this section, we consider a small, but realistic, example illustrating some important features of the analysis. Consider the following program fragment:

```
1:  let^a paper_scores:  val_adt(pos_adt(Int)) =
2:    new^b val_adt(pos_adt(Int)) in
3:      foreach^c (pos, cur_paper) in papers
4:      do
5:        let^d scores:  pos_adt(Int) = new^e pos_adt(Int) in
6:          paper_scores.write(cur_paper, scores)
7:      od;
8:  let^f reviewed_paper = paper_scores.read(45) in
9:    if(reviewed_paper != nil)
10:     then reviewed_paper.write(0, 5) else nil
```

In this program fragment, `papers` is a position-dependent container whose elements are identifiers for all submitted papers to a conference. The code above creates a new value-dependent container `paper_scores` that maintains a mapping from each paper identifier to a list of scores associated with this paper. The code iterates over `papers` and, for each paper, allocates a new position-dependent container, `scores`, and inserts the (key, value) pair, `(cur_paper, scores)` into the map.

For simplicity, let us assume this particular conference was unpopular this year and had only 3 submissions with identifiers 21, 45, and 32, which are placed in `papers` in this order. Let us also assume that $\mathbb{E}(\texttt{papers})$ is β^p and $\mathbb{S}(\beta^p) = \{(\langle\eta_p\rangle_{i_1}, \mathit{true})\}$. After the allocation at line 5 during some arbitrary k'th iteration of the loop, the abstract environment and stores are:

$$\mathbb{E}(\texttt{papers}) = \beta^p \quad \mathbb{E}(\texttt{paper\_scores}) = \beta^a$$
$$\mathbb{E}(\texttt{scores}) = \beta^d \quad \mathbb{E}(\texttt{cur\_paper}) = \beta^c$$

$$\begin{array}{lcl}
\mathbb{S}(\beta^p) & = & \{(\langle\eta_p\rangle_{i_1}, \mathit{true})\} \\
\mathbb{S}(\beta^a) & = & \{(\langle\eta_b\rangle_{i_2}, \mathit{true})\} \\
\mathbb{S}(\langle\eta_p\rangle_{i_1}) & = & \{(21, i_1 = 0), (45, i_1 = 1), \\
 & & \quad (32, i_1 = 2), (\mathit{NIL}, i_1 \geq 3)\} \\
\mathbb{S}(\beta^c) & = & \{(21, k = 0), (45, k = 1), (32, k = 2)\} \\
\mathbb{S}(\langle\eta_b\rangle_{i_2}) & = & \{(\mathit{NIL}, \mathit{true})\} \\
\mathbb{S}(\beta^d) & = & \{(\langle\eta_e\{i_3\}\rangle_{i_4}, i_3 = k)\} \\
\mathbb{S}(\langle\eta_e\{i_3\}\rangle_{i_4}) & = & \{(\mathit{NIL}, i_3 = k)\}
\end{array}$$

Consider the write at line 6, which uses `cur_paper` as the key and the freshly allocated container `scores` as the value. Here, the possible values of `cur_paper` during the k'th loop iteration are given by $\mathbb{S}(\beta^c)$ above, which encodes that the value of `cur_paper` is 21 during the first iteration ($k = 0$), 45 during the second iteration ($k = 1$), and 32 during the third iteration ($k = 2$). The abstract value set for the value `scores` is given by $\mathbb{S}(\beta^d) = \{(\langle\eta_e\{i_3\}\rangle_{i_4}, i_3 = k)\}$. Here, the freshly allocated container is represented by $\langle\eta_e\{i_3\}\rangle_{i_4}$, which has two index variables i_3 and i_4, where i_3 distinguishes allocations from different loop iterations and i_4 differentiates elements stored in the container. The constraint $i_3 = k$ in $\mathbb{S}(\beta^d)$ encodes that we are considering

the allocation that happened during the k'th iteration. Hence, the set of all possible (key, value) pairs that are written at line 6 in any k'th iteration are:

$$\left\{ \begin{array}{l} ((21, \langle \eta_e\{i_3\}\rangle_{i_4}), i_3 = k \wedge k = 0), \\ ((45, \langle \eta_e\{i_3\}\rangle_{i_4}), i_3 = k \wedge k = 1), \\ ((32, \langle \eta_e\{i_3\}\rangle_{i_4}), i_3 = k \wedge k = 2) \end{array} \right\}$$

Now, if we eliminate the dependence on a particular iteration k, we obtain the set of all possible (key, value) pairs that may be written during any iteration of the loop:

$$W = \left\{ \begin{array}{l} ((21, \langle \eta_e\{i_3\}\rangle_{i_4}), i_3 = 0), \\ ((45, \langle \eta_e\{i_3\}\rangle_{i_4}), i_3 = 1), \\ ((32, \langle \eta_e\{i_3\}\rangle_{i_4}), i_3 = 2) \end{array} \right\}$$

Observe that, while all the allocations at line 5 are represented by a single abstract container $\langle \eta_e\{i_3\}\rangle_{i_4}$, the index constraints stipulate that the allocations associated with each key are distinct from each other, since the values of i_3 are different for the keys 21, 45, and 32. Now, to process the write at line 6, we use the *update* rule from Figure 6.6 for each entry in W with $\theta_c = \{(\langle \eta_b\rangle_{i_2}, \mathit{true})\}$ (the location associated with container `paper_scores`), the key, value sets $\theta_{key}, \theta_{val}$ given by each entry in W ($\theta_{key} = \{(21, \mathit{true})\}$, $\theta_{val} = \{(\langle \eta_e\{i_3\}\rangle_{i_4}), i_3 = 0)\}$ etc.), and using the index selector ♣ since `paper_scores` is value-dependent. This yields:

$$\mathbb{S}(\langle \eta_b\rangle_{i_2}) \quad = \quad \left\{ \begin{array}{ll} (\langle \eta_e\{i_3\}\rangle_{i_4}, & ((i_3 = 0 \wedge i_2 = \mathit{pos}(21)) \vee \\ & (i_3 = 1 \wedge i_2 = \mathit{pos}(45)) \vee \\ & (i_3 = 2 \wedge i_2 = \mathit{pos}(32))), \\ & \\ (\mathit{NIL}, & i_2 \neq \mathit{pos}(21) \wedge \\ & i_2 \neq \mathit{pos}(45) \wedge i_2 \neq \mathit{pos}(32)) \end{array} \right\}$$

The new abstract value set $\mathbb{S}(\langle \eta_b\rangle_{i_2})$ expresses that all containers stored in `paper_scores` are unique because the value of i_3 is different for each key. Now, let us consider lines 8-10 in the program fragment. To determine the result of the read at line 8, we compute:

$$\left\{\begin{array}{ll} (\langle \eta_e\{i_3\}\rangle_{i_4}, & ((i_3 = 0 \wedge i_2 = pos(21)) \vee \\ & (i_3 = 1 \wedge i_2 = pos(45)) \vee \\ & (i_3 = 2 \wedge i_2 = pos(32))), \\ & \\ (NIL, & i_2 \neq pos(21) \wedge \\ & i_2 \neq pos(45) \wedge i_2 \neq pos(32)) \end{array}\right\} \bowtie (i_2 = pos(45))$$

which, when simplified, yields $\{(\langle \eta_e\{i_3\}\rangle_{i_4}, i_3 = 1)\}$. Hence, if $\mathbb{S}(\texttt{reviewed\_paper}) = \beta^f$, then:

$$\mathbb{S}(\beta_f) = \{(\langle \eta_e\{i_3\}\rangle_{i_4}, i_3 = 1)\}$$

For the `if` expression at line 9, only the then branch is satisfiable since $\langle \eta_e\{i_3\}\rangle_{i_4}$ is not *NIL*. Finally, after the write at line 11, the values for the nested containers are given by:

$$\mathbb{S}(\langle \eta_e\{i_3\}\rangle_{i_4}) = \{(5, i_4 = 0 \wedge i_3 = 1), (NIL, i_4 \neq 0 \vee i_3 \neq 1)\}$$

Hence, this abstract store encodes that only the score at position 0 of the scores list associated with key 45 in `paper_scores` has been changed to 5, but the score lists associated with all other keys are unchanged.

6.3.4 Soundness of the Abstraction

In order to state the soundness theorem, we first need to define an abstraction function from concrete to abstract memory locations. Observe that if $\vec{i}$ denotes a vector of index variables used in some abstract location δ and σ is a concrete assignment to each of the index variables in $\vec{i}$, then the pair (δ, σ) represents one concrete memory location. Therefore, the abstraction function is a mapping from concrete locations to a pair consisting of an abstract memory location δ and a full assignment σ to all index variables used in δ:

$$\textit{Abstraction function } \alpha = \textit{Concrete loc } (l, i) \rightarrow (\delta, \sigma)$$

To make this abstraction function precise, it is necessary to augment the operational semantics with some additional bookkeeping machinery that was omitted from Figure 6.4 to avoid complicating the language semantics. First, for each concrete location (l, i) in store S, we need to determine the program point ρ that results in the binding of (l, i) in S; we write $id(l, i)$ to denote the program point ρ associated with the introduction of (l, i). Second, to be able to give a full assignment to the index variables in an abstract location, we need to determine the counter vector C when a concrete location was introduced. Hence, we assume an environment A maps each concrete location (l, i) to the counter vector C present when (l, i) was introduced in concrete store S. Since it is trivial to extend the operational semantics from Figure 6.4 to track $id(l, i)$ and $A(l, i)$, we assume this additional bookkeeping information is available. We can now define the abstraction function as follows:

Definition 17 (Abstraction Function) *Let (l, k) be a concrete memory location, and let $\mathrm{id}(l, k) = \rho$ such that ρ labels expression e^ρ, and $A(l, k) = C$. Then, the abstraction of (l, k), written $\alpha(l, k)$, is:*

$$
\begin{array}{lll}
1. & (\langle \eta_\rho \{\vec{i_\rho}\} \rangle_{i_0^\rho}, \vec{i_\rho} = C \wedge i_0^\rho = k) & \text{if } e^\rho = \texttt{new}^\rho \texttt{ pos\_adt } \tau \\
2. & (\langle \eta_\rho \{\vec{i_\rho}\} \rangle_{i_0^\rho}, \vec{i_\rho} = C \wedge i_0^\rho = \mathrm{pos}(k)) & \text{if } e^\rho = \texttt{new}^\rho \texttt{ val\_adt } \tau \\
3. & (\beta^\rho, \mathrm{true}) & \text{otherwise}
\end{array}
$$

We extend this abstraction function from all concrete values v to all abstract values π in the following obvious way:

$$
\alpha(v) = \begin{cases}
(NIL, true) & \text{if } v = \texttt{NIL} \\
(c, true) & \text{if } v \text{ is integer constant } c \\
\alpha(l, k) & \text{if } v \text{ is a memory location} (l, k)
\end{cases}
$$

We write $\sigma(\phi)$ to denote the result of substituting each of the variables in ϕ with their concrete assignment specified by σ. In addition, we assume the substitution $\sigma(\phi)$ gives an interpretation to all function symbols pos^ρ in ϕ by replacing pos^ρ with the particular permutation it stands for in a given execution. Since it is trivial to extend the operational semantics to track which permutation was used for which loop,

we assume this information is available.

Definition 18 (Value Agreement) *Let v be a concrete value with $\alpha(v) = (\pi, \sigma)$, and let θ be an abstract value set. We say concrete value v agrees with abstract value set θ, written $v \sim \theta$, if:*

$$1.\quad (\pi, \langle \varphi_{\text{may}}, \varphi_{\text{must}} \rangle) \in \theta \wedge \text{VALID}(\sigma(\varphi_{\text{may}}))$$
$$2.\quad \forall (\pi', \langle \varphi'_{\text{may}}, \varphi'_{\text{must}} \rangle) \in \theta.\ \text{UNSAT}(\sigma(\varphi'_{\text{must}})) \quad (\pi' \neq \pi)$$

In this definition, the first condition states the correctness of the overapproximation encoded by θ, and the second condition states the correctness of the underapproximation. If the abstract representation of v is (π, σ), then, for the overapproximation to be correct, π must be in θ under some constraint $\langle \varphi_{may}, \varphi_{must} \rangle$ and the may constraint φ_{may} must evaluate to *true* under the index assignment σ. (Recall that since the language from Section 6.2 has no inputs, the only variables in constraints are index variables; thus, φ_{may} always evaluates to a constant under σ.) The second condition of value agreement states the correctness of the underapproximation, requiring at most the abstract representation of v to be in θ, i.e., all other elements in θ should be infeasible under index assignment σ.

Definition 19 (State Agreement) Let (v, E, S, C) be a *concrete state*, consisting of a concrete value v, concrete environment E, concrete store S and counter vector C, and let $(\theta, \mathbb{E}, \mathbb{S}, \mathbb{C})$ be an *abstract state* with abstract value set θ and abstract environment and store $\mathbb{E}, \mathbb{S}$ and counter vector $\mathbb{C}$. We say concrete state (v, E, S, C) agrees with abstract state $(\theta, \mathbb{E}, \mathbb{S}, \mathbb{C})$, written $(v, E, S, C) \sim (\theta, \mathbb{E}, \mathbb{S}, \mathbb{C})$, if the following conditions hold:

1. $v \sim \theta$ (according to Definition 18)
2. $\forall v \in dom(E).\ (v \in dom(\mathbb{E}) \ \wedge\ \mathbb{E}(v) = \alpha(E(v), 0))$
3. $\forall (l, k) \in dom(S).\ \ S(l, k) = l' \Rightarrow$
$$(\alpha(l, k) = (\delta, \sigma) \ \wedge\ l' \sim (\mathbb{S}(\delta) \bowtie \sigma))$$
4. $\mathbb{C} = C$

Theorem 2 (Soundness) Let P be any program. If $(E, S, C) \sim (\mathbb{E}, \mathbb{S}, \mathbb{C})$, then

$$E, S, C \vdash P : l, S'$$
$$\Rightarrow$$
$$\mathbb{E}, \mathbb{S}, \mathbb{C} \vdash P : \theta, \mathbb{S}' \ \wedge \ (l, E, S', C) \sim (\theta, \mathbb{E}, \mathbb{S}', \mathbb{C})$$

The proof is given in Section 6.7.

6.4 Extensions

The language we have used for the formal development requires keys of value-dependent containers to be integers, but, in real languages, keys may have arbitrary types. The techniques we have described so far are directly applicable when pointer values are used as keys because pointer equality is a form of integer equality. However, it is common to define custom equality predicates on some types, and determining whether two keys are equal may be more involved than simple integer equality. Consider the following C++ code snippet:

```
class Point {
  int x; int y; color c;
  Point(int x, int y, color c){
    this->x=x; this->y=y; this->c=c;
  };
  bool operator==(const Point & other) {
    return x == other.x && y == other.y;
  }
}
unordered_set<Point> points;
Point p1 = Point(5, 34, RED);  points.insert(p1);
Point p2 = Point(5, 34, BLUE); points.insert(p2);
```

Here, the type `Point` defines a custom equality operator that only checks the x and y coordinates for a point but disregards its color. In the above program, after the

last insertion operation, there is only one element in the set even though two points with different colors are inserted. If we treat `p1` and `p2` as variables in the constraint language, our technique would conclude that the size of the set is 2 under constraint $\texttt{p2} \neq \texttt{p1}$ and 1 under $\texttt{p2} = \texttt{p1}$. To be more precise in the presence of custom equality predicates, we infer axioms characterizing when two objects are equal. Specifically, by analyzing the implementations of the custom equality predicates, we infer necessary and sufficient conditions $\langle \varphi^{=}_{may}, \varphi^{=}_{must} \rangle$ characterizing when two objects o_1 and o_2 may and must be equal. (Observe that treating `p1` and `p2` as variables in the constraint language as above is equivalent to the trivial and always sound equality condition $\langle \mathit{true}, \mathit{false} \rangle$). Now, in order to take into account what we know about the custom equality predicate, we add the axioms $\forall o_1, o_2.\ o_1 = o_2 \Rightarrow \varphi^{=}_{may}$ and $\forall o_1, o_2.\ \varphi^{=}_{must} \Rightarrow o_1 = o_2$ to the constraint solver. For instance, for the simple equality predicate for `Point`, we could utilize the axiom $\forall p_1, p_2.\ p_1 = p_2 \Leftrightarrow (p_1.x = p_2.x \wedge p_1.y = p_2.y)$, allowing the technique to conclude that the size of the set after the second insertion is 1.

In the technical development, we also assumed that the iteration order over value-dependent containers is arbitrary. While this is true in most cases, some value-dependent containers (such as a red-black-tree based map) may visit keys in a certain order during an iteration. We can encode such restrictions in the iteration order by analyzing the custom less than operators and inferring appropriate axioms about the *pos* function in a similar way as above.

6.5 Implementation

We have implemented the ideas presented in this chapter in our Compass program verification framework for analyzing C and C++ programs. Compass utilizes a gcc and g++ based front-end called SAIL [37] which translates C and C++ code to a low-level representation, similar to 3-address code. Compass uses the Mistral SMT solver [36, 39] for solving and simplifying constraints generated during the analysis. Compass supports most features of C++, including classes, arrays, dynamic memory allocation, pointer arithmetic, references, single and multiple inheritance, and virtual method

calls. Compass performs path-sensitive analysis and achieves context-sensitivity by computing polymorphic summaries of functions (and loops) and instantiates them in calling contexts [40].

6.6 Experimental Evaluation

To demonstrate the usefulness of the ideas presented in this chapter, we evaluate the proposed technique in two different ways: In a first set of experiments, we perform a case study and prove the functional correctness of a set of small, but challenging programs manipulating containers. In a second set of experiments, we use this technique to prove memory safety properties of real C++ applications that heavily use containers, and we show that a precise understanding of data structure contents is beneficial in improving analysis results. For both sets of experiments, we annotated the containers provided by the C++ standard template library [1], either directly as position- or value-dependent containers or by nesting them inside already annotated STL containers.

6.6.1 Case Study

In our case study, we analyze fifteen small, but challenging, example programs totaling close to 1000 lines of code. All benchmarks are available at:
`http://www.stanford.edu/~tdillig/cont.txt`.

The results of the case study are presented in Figure 6.8; we briefly discuss each of the programs from this table. The first two programs copy the contents of a vector and a map into another container of the same type and assert their element-wise equality. Program 3 builds the reverse map `r` of map `m` by inserting each (k, v) pair in `m` as the key-value pair (v, k) of `r`. Program 4 is modeled after the example in Figure 6.1 and asserts the correctness of the entries in `exam_scores`. Program 5 inserts all the keys in a map `m` into a set `s` and asserts that `s` contains exactly the keys in `m`. Programs 6-8 illustrate nested containers by asserting properties about the composed data structures. Program 9 inserts numbers $[0, size)$ into a stack and a queue and

	Program	**Time**	**Memory**
1	Vector copy	0.22s	<2 MB
2	Map copy	0.33s	<2 MB
3	Reverse mapping	0.14s	<2 MB
4	Example from Introduction	0.22s	<4 MB
5	Set containing map keys	0.62s	<2 MB
6	Map of lists	0.21s	<2 MB
7	Vector of sets	0.11s	<2 MB
8	Multimap	0.33s	<2 MB
9	Stack-queue relationship	0.19s	<2 MB
10	Singleton pattern correctness	0.23s	<5 MB
11	Prove map values non-null	0.30s	<2 MB
12	Prove non-aliasing between vector elements	0.31s	<2 MB
13	List containing key,value pairs of a map	1.14s	<2 MB
14	Set containing map keys with non-null values	0.44s	<2 MB
15	Relationship between keys and values in map	0.31s	<2 MB

Figure 6.8: Experimental Results of the Case Study

asserts that the top of the stack is the last element in the queue. Program 10 emulates the singleton pattern through a `get_shared` method that uniquifies objects that are the same according to their custom equality predicate by using a set, and asserts the correctness of `get_shared`. Program 11 asserts that the values in a map are non-null, and Program 12 asserts that there is no aliasing between elements in a vector. Program 13 builds a list containing (key, value) pairs in a map and asserts that the list contains exactly the key, value pairs in the map. Program 14 builds a set containing all map keys with non-null values and asserts that the set contains exactly the keys with non-null values. Program 15 asserts various properties about the relationship between keys and values in a map. Compass is able to fully automatically verify all of these examples, while reporting errors in slightly modified, buggy versions of these programs. As shown in Figure 6.8, the running times for most of these examples are under a second and maximum memory consumption is consistently below 4 MB. We believe these examples illustrate that Compass can automatically verify interesting properties about the functional correctness of client programs using containers and their nestings.

6.6.2 Proving Memory Safety Properties

In a second set of experiments, we investigate the added benefits of precise reasoning about container contents when checking for memory safety properties in real C++ applications. Using Compass, we analyzed three applications ranging from 16,030 to 128,318 lines of C++ code: The first application is LiteSQL [7], which integrates C++ objects tightly with a database. The second application we analyzed is the widget library of the vector graphics program, Inkscape (which was used for the drawings in this submission) [6]. We chose this component of Inkscape because it illustrates how more complex abstract data types are implemented using standard containers as building blocks. The third application is Digikam, a stand-alone, fairly large, open-source photo management program [3].

Both LiteSQL and the Inkscape widget library use the C++ standard template library (STL), while Digikam uses container libraries of the QT framework [10]. Fortunately, since the containers in QT are interface-compatible with the ones in the STL, we were able to use the same set of container interface annotations for all three applications. As typical of many programs written in an object-oriented style, all of these applications make heavy use of containers, such as vectors, lists, maps, and their combinations.

To demonstrate the importance of precise, per-element reasoning about containers when checking for memory safety properties, we analyzed these applications in two different configurations: In the first configuration, we use the technique described in this chapter, while in the second configuration, we track which set of elements a container may store, but we do not reason about the relationship between positions and values for position-dependent containers, and we do not track the key-value relationships for value-dependent containers (i.e., we "smash" containers into a set of values).

Figure 6.9 summarizes the results of our experiments. For each of the three applications, we check the following memory safety properties: Null pointer dereferences, memory leaks (i.e., lack of unreachable memory), and accessing deleted memory. All of our experiments were performend on an 8-core 2.66 GHz Xeon workstation with 24GB of memory. In Figure 6.9, we provide the running times of the analyses both on

LiteSQL 0.3.8		
Number of lines	16,030	
	Our technique	**Containers as sets**
Running time 1 CPU	4.5 min	5.8 min
Running time 8 CPUs	1.6 min	1.6 min
Maximum Memory	1.3 GB	1.5 GB
Null Dereference Errors		
Actual errors	2	2
False positives	2	68
Memory Leak Errors		
Actual errors	3	3
False positives	0	7
Access to Deleted Memory		
Actual errors	0	0
False positives	0	4
Total FP to error ratio	*0.75*	*15.8*
Inkscape 0.47 Widget Library		
Number of lines	37,211	
	Our technique	**Containers as sets**
Running time 1 CPU	7.2 min	6.1 min
Running time 8 CPUs	2.3 min	2.1 min
Maximum Memory	1.9 GB	1.8 GB
Null Dereference Errors		
Actual errors	1	1
False positives	0	24
Memory Leak Errors		
Actual errors	1	1
False positives	1	18
Access to Deleted Memory		
Actual errors	2	2
False positives	2	22
Total FP to error ratio	*0.75*	*16*
Digikam 1.2.0		
Number of lines	128,318	
	Our technique	**Containers as sets**
Running time 1 CPU	45.1 min	44.3 min
Running time 8 CPUs	8.7 min	10.3 min
Maximum Memory	12.0 GB	10.6 GB
Null Dereference Errors		
Actual errors	17	17
False positives	8	220
Memory Leak Errors		
Actual errors	8	8
False positives	1	45
Access to Deleted Memory		
Actual errors	3	3
False positives	0	6
Total FP to error ratio	*0.32*	*9.68*

Figure 6.9: Proving memory safety properties

a single core as well as on all eight cores. Since the analysis is summary-based, many functions can be analyzed in parallel to yield much better running times, ranging from 1.6 to 8.7 minutes on eight cores.

In Figure 6.9, observe that the technique presented in this chapter improves the precision of the analysis over the second configuration which treats the container's contents as a set, in many cases by an order of magnitude. For example, the total false positive to error ratio for Digikam is 0.32 if the technique presented in this chapter is used for the analysis, while this ratio increases to 9.68 with the second analysis configuration. This statistic means that there are roughly three actual error reports per false positive using our technique, while there is less than one actual error per nine false positives using the second, less precise configuration. We believe that this dramatic reduction in false positives illustrates the usefulness of our technique for analyzing real-world C++ applications.

Also, observe in Figure 6.9 that there are no significant differences in running time and memory consumption between the two analysis configurations. We believe that the statistics provided in Figure 6.9 illustrate that our technique adds useful precision without incurring significant extra computational resources.

6.7 Proof of Soundness

In this section, we sketch the proof of soundness of the key rules from Section 6.3.3. The proof is a standard induction on the inference rules from Figures 6.5, 6.6, and 6.7. We only focus on the rules that involve containers.

6.7.1 Preliminaries

We first introduce some notation that is convenient to use in the proofs and state some assumptions.

Definition 20 *($\sigma(\theta)$) Let θ be an abstract value set, and let σ be an assignment to*

(at least) the index variables in θ. Then:

$$\sigma(\theta) = \{(\pi_j, \sigma(\phi_j) \mid (\pi_j, \phi_j) \in \theta \wedge \mathrm{SAT}(\sigma(\phi_j)\}$$

Definition 21 *($\lceil\phi\rceil$, $\lfloor\phi\rfloor$) Let ϕ be the bracketing constraint $\langle\varphi_{\mathrm{may}}, \varphi_{\mathrm{must}}\rangle$. Then, $\lceil\phi\rceil = \varphi_{\mathrm{may}}$ and $\lfloor\phi\rfloor = \varphi_{\mathrm{must}}$.*

Throughout the proof, we assume that every abstract value π that can arise for a given program is present in every abstract value set θ; for values that have not been explicitly added to θ, we assume $(\pi, \mathit{false}) \in \theta$.

6.7.2 Proof of Key Rules

We first consider the read rule for position-dependent containers:

Let (l_c, k) denote the concrete location that the read is performed on (i.e, the result is obtained from $S(l_c, k)$). Let $\alpha(l_c, k) = (\delta, \sigma_c)$, and let $\alpha(S(l_c, k)) = (\pi, \sigma_\pi)$. In the rule, suppose:

$$\begin{array}{lcl} \theta_1 & = & \{\ldots, (\langle\alpha\rangle_{i_j}, \phi_j), \ldots\} \\ \theta_2 & = & \{\ldots, (\pi_k, \phi_k), \ldots\} \\ S(\langle\alpha\rangle_{i_j}) & = & \{\ldots, (\pi_{l_j}, \phi_{l_j}), \ldots\} \end{array}$$

By the assumption that the abstraction is correct before the read (i.e., $E, S, C \sim \mathbb{E}, \mathbb{S}, \mathbb{C}$), we have:

$$\begin{array}{lcl} (\langle\alpha\rangle_{i_j} = \delta) & \Rightarrow & \sigma_c(\lceil\phi_j\rceil) = \mathit{true} \\ (\langle\alpha\rangle_{i_j} \neq \delta) & \Rightarrow & \sigma_c(\lfloor\phi_j\rfloor) = \mathit{false} \\ \pi_k = k & \Rightarrow & \lceil\phi_k\rceil = \mathit{true} \\ \pi_k \neq k & \Rightarrow & \lfloor\phi_k\rfloor = \mathit{false} \\ \pi_{l_j} = \pi & \Rightarrow & \sigma_\pi(\sigma_c(\lceil\phi_{l_j}\rceil)) = \mathit{true} \\ \pi_{l_j} \neq \pi & \Rightarrow & \sigma_\pi(\sigma_c(\lfloor\phi_{l_j}\rfloor)) = \mathit{false} \end{array} \quad (*)$$

The resulting abstract value set θ is computed by the rule as:

$$\theta = S(\langle\alpha\rangle_{i_j}) \bowtie \ (\bigvee((i_j = \pi_k) \wedge \phi_k) \wedge \phi_j)$$

Now, assume, for contradiction, that $S(l_c, k) \not\sim \theta$. Then, either (i) $(\pi, \langle \mathit{true}, * \rangle) \notin \sigma_\pi(\theta)$, or (ii) $\exists \pi' \neq \pi.\ (\pi', \langle *, \mathit{true} \rangle) \in \sigma_\pi(\theta)$.

Assume (i). By (*), for $\delta = \langle\alpha\rangle_{i_j}$ and $\pi_k = k$, we have $\sigma_c(\lceil\phi_j\rceil) = \mathit{true}$ and $\lceil\phi_k\rceil = \mathit{true}$; thus,

$$\theta = S(\langle\alpha\rangle_{i_j}) \bowtie \ \langle(i_j = k \wedge \phi_j), *\rangle$$

By correctness of the abstraction before the read, we have:

$$(\pi, \langle\varphi_{may}, \varphi_{must}\rangle) \in \mathbb{S}(\langle\alpha\rangle_{i_j})$$

Furthermore since $\sigma_\pi(\sigma_c(\varphi_{may})) = \mathit{true}$ by (*) and since σ_c must assign i_j to k and $\sigma_c(\phi_j) = \mathit{true}$,

$$\sigma_\pi(\sigma_c(\varphi_{may} \wedge (i_j = k) \wedge \phi_j)) = \mathit{true}$$

Hence, assumption (i) is not possible.

Now, assume (ii). First, observe that if $\langle\alpha\rangle_{i_j} = \delta$ and $\pi_k = k$, then, from the last identity in (*), it follows that (ii) cannot hold. Now, if $\langle\alpha\rangle_{i_j} \neq \delta$, then $\sigma_c(\lfloor\phi_j\rfloor) = \mathit{false}$, and $\forall(\pi', \phi').\ \in \sigma_c(S(\langle\alpha\rangle_{i_j}) \bowtie \ (\bigvee((i_j = \pi_k) \wedge \phi_k) \wedge \phi_j))$, we have $\lfloor\phi'\rfloor = \mathit{false}$. Now, if $\pi_k \neq k$, we know from (*) that $\lfloor\phi_k\rfloor = \mathit{false}$, hence (ii) is again not possible.

An almost identical argument also applies to value-dependent containers; the only difference is that σ_c now assigns i_j to $\mathit{pos}(k)$ and θ is computed as:

$$\theta = S(\langle\alpha\rangle_{i_j}) \bowtie \ (\bigvee((i_j = \mathit{pos}(\pi_k) \wedge \phi_k) \wedge \phi_j)$$

Now, we consider a write $v_1.\texttt{write}(v_2, v_3)$ to position-dependent container v_1. Let (l, k) denote the concrete memory location that is modified, and let v denote the

concrete value that is written. From the operational semantics, we have:

$$S'(l,i) = \begin{cases} v & \textit{if } \ i = k \\ S(l,i) & \textit{otherwise} \end{cases} \qquad (1)$$

Let:

$$\begin{aligned} &\alpha(l,i) = (\delta, \sigma_i) \\ &\alpha(k) = (k, \textit{true}) \\ &\alpha(v) = (\pi_v^*, \sigma_v^*) \\ &\alpha(S(l,i)) = (\pi_{e_i}, \sigma_{e_i}) \end{aligned}$$

In the write rule from Figure 6.6, let:

$$\begin{array}{lcl} \theta_1 & = & \{\ldots, (\langle\alpha\rangle_{i_j}, \phi_j), \ldots\} \\ \theta_2 & = & \{\ldots, (\pi_k, \phi_k), \ldots\} \\ \theta_3 & = & \{\ldots, (\pi_v, \phi_v), \ldots\} \\ \mathbb{S}(\langle\alpha\rangle_{i_j}) & = & \{\ldots, (\pi_{l_j}, \phi_{l_j}), \ldots\} \end{array}$$

By the assumption that the abstraction is correct before the write (i.e., $E, S, C \sim \mathbb{E}, \mathbb{S}, \mathbb{C}$), we know:

$$\begin{array}{lcl} (\pi_k = k) & \Rightarrow & \lceil \phi_k \rceil = \textit{true} \\ (\pi_k \neq k) & \Rightarrow & \lfloor \phi_k \rfloor = \textit{false} \\ (\pi_v = \pi_v^*) & \Rightarrow & \sigma_v^*(\lceil \phi_v \rceil) = \textit{true} \\ (\pi_v \neq \pi_v^*) & \Rightarrow & \sigma_v^*(\lfloor \phi_v \rfloor) = \textit{false} \\ (\delta = \langle\alpha\rangle_{i_j}) & \Rightarrow & \sigma_i(\lceil \phi_j \rceil) = \textit{true} \\ (\delta \neq \langle\alpha\rangle_{i_j}) & \Rightarrow & \sigma_i(\lfloor \phi_j \rfloor) = \textit{false} \\ \pi_{l_j} = \pi_{e_i} & \Rightarrow & \sigma_{e_i}(\sigma_i(\lceil \phi_{l_j} \rceil)) = \textit{true} \\ \pi_{l_j} \neq \pi_{e_i} & \Rightarrow & \sigma_{e_i}(\sigma_i(\lfloor \phi_{l_j} \rfloor)) = \textit{false} \end{array} \qquad (*)$$

In the write rule, for each $\langle\alpha\rangle_{i_j}$, the new entry in new abstract store $\mathbb{S}'$ is computed as:

$$\begin{array}{lcl} \mathbb{S}'(\langle\alpha\rangle_{i_j}) & = \mathbb{S}[\langle\alpha\rangle_{i_j} \leftarrow & (\theta_3 \wedge (\theta_2 \diamondsuit i_j) \wedge \phi_j) \cup \\ & & \mathbb{S}(\langle\alpha\rangle_{i_j}) \wedge (\neg(\theta_2 \diamondsuit i_j) \vee \neg\phi_j)] \end{array} \qquad (**)$$

where $\theta \wedge \phi$ is shorthand for conjoining ϕ with every constraint in θ, as described in

Section 6.3.3.

Assume that the write rule is not sound. Then, using (1), either:

$$\begin{array}{ll} \text{(i)} & v \not\sim (\mathbb{S}'(\delta) \bowtie \sigma_i[k/i]) \quad \textit{or} \\ \text{(ii)} & S(l,i) \not\sim (\mathbb{S}'(\delta) \bowtie \sigma_i) \quad \textit{for} \;\; i \neq k \end{array}$$

We first consider (i). Suppose $v \not\sim (\mathbb{S}'(\delta) \bowtie \sigma_i[k/i])$. Then, there are two possibilities:

$$\begin{array}{ll} 1a. & (\pi_v^*, \langle \mathit{true}, * \rangle) \notin \sigma_v^*(\mathbb{S}'(\delta) \bowtie \sigma_i[k/i]) \\ 1b. & \exists \pi_v' \neq \pi_v^*.\ (\pi_v', \langle *, \mathit{true} \rangle) \in \sigma_v^*(\mathbb{S}'(\delta) \bowtie \sigma_i[k/i]) \end{array}$$

Now, assume 1a and consider evaluating $\sigma_v^*(\mathbb{S}'(\delta) \bowtie \sigma_i[k/i])$. Under σ_v^*, we know from (*) that in (**), $\sigma_v^*(\theta_2)$ contains the pair $(\pi_v^*, \langle \mathit{true}, * \rangle)$. Observe that $\sigma_i[k/i]$ must assign i_j to k. Hence, by using (*), we know that the constraint

$$\theta_2 \diamondsuit i_j = \bigvee ((i_j = \pi_k) \wedge \phi_k) \qquad (***)$$

evaluates to $\langle \mathit{true}, * \rangle$ under assignment $\sigma_i[k/i]$. Furthermore, under assignment $\sigma_i[k/i]$, (*) implies that $\lceil \phi_j \rceil$ is *true*; hence it follows that $(\pi_v^*, \langle \mathit{true}, * \rangle) \in \sigma_v^*(\mathbb{S}'(\delta) \bowtie \sigma_i[k/i])$, contradicting assumption 1a.

Now assume 1b. Under assignment σ_v^*, we know from (*) that for any $(\pi_v', \phi_v') \in \theta_2$ such that $\pi_v' \neq \pi_v^*$, $\lfloor \phi_v' \rfloor$ is *false*. Since conjoining additional constraints cannot weaken *false*, it follows that $\forall \pi_v' \neq \pi_v^*.\ (\pi_v', \langle *, \mathit{false} \rangle) \in \sigma_v^*(\mathbb{S}'(\delta) \bowtie \sigma_i[k/i])$, contradicting assumption 1b.

We know consider (ii), i.e., $S(l,i) \not\sim (\mathbb{S}'(\delta) \bowtie \sigma_i)$ for some i such that $i \neq k$. This corresponds to the case where the abstract semantics for `write` overwrites the existing value of the wrong key. Again, there are two possibilities:

$$\begin{array}{ll} 2a. & (\pi_{e_i}, \langle \mathit{true}, * \rangle) \notin \sigma_{e_i}(\mathbb{S}'(\delta) \bowtie \sigma_i) \\ 2b. & \exists \pi_{e_i'} \neq \pi_{e_i}.\ (\pi_{e_i}, \langle *, \mathit{true} \rangle) \in \sigma_{e_i}(\mathbb{S}'(\delta) \bowtie \sigma_i) \end{array}$$

Now, assume 2a. From (*), we know that under assignment σ_{e_i}, $(\pi_{e_i}, \langle \mathit{true}, * \rangle)$. Now, by (**), we need to show that conjoining the constraint $(\neg(\theta_2 \diamondsuit i_j) \vee \neg \phi_j)$ cannot

strengthen *true*. Observe that σ_i assigns i_j to i where $i \neq k$, and observe that $\pi_k \neq i \Rightarrow \sigma_i(\lfloor \phi_k \rfloor) = \mathit{false}$; hence the sufficient condition for $\theta_2 \diamondsuit i_j$ is *false*. Thus, $\lceil \neg(\theta_2 \diamondsuit i_j) \rceil = \mathit{true}$, which implies $(\pi_{e_i}, \langle \mathit{true}, * \rangle) \in \sigma_{e_i}(\mathbb{S}'(\delta) \bowtie \sigma_i)$, contradicting 2a.

Finally, assume 2b. Under assignment σ_{e_i}, we know from (*) that for any $(\pi_{l_j}, \phi_{l_j}) \in \mathbb{S}(\langle \alpha \rangle_{i_j})$ such that $\pi_{l_j} \neq \pi_{e_i}$, $\lfloor \phi_{l_j} \rfloor$ is *false*. Since conjoining additional constraints cannot weaken *false*, assumption 2b is also infeasible.

The proof for value-dependent containers is almost identical. The only differences are that σ_i now assigns i_j to $\mathit{pos}(i)$ according to the definition of the abstraction function, and $\mathbb{S}'$ is computed as:

$$\begin{aligned} \mathbb{S}'(\langle \alpha \rangle_{i_j}) \quad = \mathbb{S}[\langle \alpha \rangle_{i_j} \leftarrow \quad & (\theta_3 \wedge (\theta_2 \clubsuit i_j) \wedge \phi_j) \cup \\ & \mathbb{S}(\langle \alpha \rangle_{i_j}) \wedge (\neg(\theta_2 \clubsuit i_j) \vee \neg \phi_j)] \end{aligned}$$

We now consider the `foreach` rule. Since the *fix* rule stipulates that $\mathbb{S}^*$ is a correct invariant without giving a constructive algorithm, we only argue about the loop initialization, i.e., (key, value) pairs bound in the `foreach` rule of the abstract semantics correctly model the concrete execution. We focus on value-dependent containers. Let key_k be the concrete key visited during the k'th loop iteration such that $\alpha(\mathit{key}_k) = (\mathit{key}_k, \sigma_k)$ where σ_k gives interpretation to function symbols pos^ρ. In the abstract semantics, the value set θ_{key} during the k'th iteration is given by

$$\theta_{key} = \{\ldots, (\pi_k, k = \mathit{pos}^\rho(\pi_k)), \ldots\}$$

Suppose $\mathit{key}_k \not\sim \theta_{key}$. Then, either $(\mathit{key}_k, \langle \mathit{true}, * \rangle) \notin \sigma_k(\theta_{key})$ or $\exists \pi_k \neq \mathit{key}_k.\ (\pi_k, \langle *, \mathit{true} \rangle) \in \sigma_k(\theta_{key})$

But, under interpretation σ_k, we have:

$$\begin{aligned} \pi_k = \mathit{key}_k &\Rightarrow \sigma_k(\mathit{pos}^\rho) = k \\ \pi_k \neq \mathit{key}_k &\Rightarrow \sigma_k(\mathit{pos}^\rho) \neq k \end{aligned}$$

Hence, $\mathit{key}_k \sim \theta_{key}$.

Chapter 7

Previous Work on Data Structure Analysis

In this chapter, we survey existing techniques for reasoning about various aspects of heap data structures and arrays.

7.1 Shape Analysis

Reasoning about unbounded data structures has a long history. Jones et al. first propose *summary nodes* to finitely represent lists in LISP [57], and [27] extends this work to languages with updates and introduces strong and weak updates. Representation of access paths qualified by indices is first introduced in Deutsch [35], which uses a combination of *symbolic access paths* and numeric abstract domains to represent may-alias pairs for recursive data structures, such as lists. Symbolic access paths proposed by Deutsch are similar to the indexed locations used in this thesis in that they both employ a symbolic index to differentiate between elements in data structures. However, Deutsch's technique is mainly limited to may-alias relations between list elements and can only perform strong updates in a limited number of situations, as this technique does not make use of underapproximations.

Most of the existing work on data structure analysis, most notably three-valued logic analysis [73, 22] and techniques based on separation logic [41, 74, 81], focus on the

inference of so-called *shape invariants*, which address questions such as "Is this data structure a singly-linked list, a doubly linked-list, a tree, a DAG, etc.?". Unlike such shape analyses, the techniques we consider in this thesis are not meant for verifying the implementations of data structures, but rather for analyzing the contents and client-side uses of these data structures. We believe the techniques addressed in this thesis are both orthogonal and complementary to shape analysis. For example, while we can use shape analysis to verify the implementation of a map as a red-black tree, we can use our techniques to precisely reason about a client program using this map library.

7.2 Array Analysis

The most basic technique for reasoning about array contents is *array smashing*, which represents all array elements with one summary node and only allows weak updates [20]. As a result, techniques that employ array smashing are imprecise and cannot reason about position-value correlations.

Gopan et al. propose a three-valued logic based framework to discover relationships among values of array elements [48]. This technique isolates individual elements to perform strong updates and places elements that share a common property into a partition, which is commonly a contiguous range, and a heuristic is used to automatically infer relevant partitions. In contrast, our approach to array analysis does not need to distinguish between strong and weak updates or concretize individual elements. Furthermore, our approach obviates the need for explicit partitioning and therefore does not suffer from the same state space explosion problem. It can also naturally express invariants about non-contiguous array elements.

Jhala and McMillan propose a technique for reasoning about arrays based on counterexample-guided abstraction refinement [56]. This technique discovers properties similar to those in [48], but does so in a demand-driven way using interpolation. This approach also only reasons about contiguous ranges and also constructs explicit partitions. Furthermore, the predicates used in the abstraction belong to a finite language to guarantee convergence.

Many techniques have been proposed for inferring complex invariants, such as sortedness, about array elements [54, 59, 16, 50, 76, 45]. While the techniques we have described do not attempt to reason about such invariants, our approach goes beyond scalar arrays and, unlike the afore-mentioned techniques, is capable of precisely reasoning about heap objects that are manipulated through arrays and containers.

7.3 Client-Side Use of Heap Data Structures

Similar to the container analysis techniques we have presented in this thesis, the Hob verification framework also separates the verification of data structure implementations from the verification of their client-side use [63, 62, 61]. Hob's main focus is to verify that the implementation of a data structure obeys its specification; on the client-side, Hob can be used to check that custom data structure invariants are obeyed by the client, such as the requirement that the data structure has no content before a certain method is called. While Hob addresses a more general class of abstract data structures than containers, the client-side abstraction of Hob is a *set abstraction* of data structures, which is less precise than the abstraction we consider. For instance, Hob's client-side reasoning about a map does not track the relationship between keys and values in a map or between positions and elements in a vector [61]. In contrast, we only consider the client-side use of a special, yet fundamental, class of data structures, and our focus is a fully automatic technique to improve analysis precision when analyzing real C++ programs that use containers.

Another work that addresses the client-side use of data structures is [72], which focuses on verifying that the client of a software component obeys the requirements of that component, such as the requirement that a data structure d is not modified during an iteration over d. Another work with a similar focus is [19], which uses predicate abstraction to verify that clients of the C++ standard template library (STL) obey the requirements for correct use of this library. Yet another work that is focused on usage of STL data structures is [49], which is an unsound bug finding tool for discovering incorrect usage of STL primitives. None of these efforts consider properties which require reasoning about the contents of containers, which is our

focus.

The work described in [44] addresses the typestate verification problem for real-world Java programs, which make heavy use of containers. This work also reports on the challenge of achieving sufficient precision as objects flow in and out of containers; they utilize techniques such as *focus* and *blur*, developed by the shape analysis community based on 3-valued logic [73]. In contrast, our approach never performs explicit case splits on abstract containers and instead uses constraints to both specify different elements in the container as well as to perform updates on individual elements.

7.4 Relational Analysis of Data Structures

While analyses based on three-valued logic [73, 22] can in principle be fully relational, heuristics necessary to ensure termination and scalability often lead to a heap abstraction that does not enforce existence and uniqueness of memory invariants, for example, after a blur operation is performed. To mitigate the state-space explosion that arises from analyzing the set of all possible heaps in the TVLA framework, Manevitch [68] proposes an interesting technique called *partial isomorphic heap abstraction*, which merges 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. Unlike heap analysis techniques based on three-valued logic, 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 often 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.

Unlike the graph-based heap abstraction we consider in this thesis, some approaches for heap analysis represent the heap purely as logical formulas by making use of combinations of various logics, such as the theory of arrays [66, 23, 79, 53] and

pointer logic [60] 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 [48, 38, 73], 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 relational heap analysis technique we have presented combines aspects of both approaches by allowing relational reasoning in a practical and scalable way 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.

Chapter 8

Cuts-from-Proofs

All the techniques we have described so far in this thesis effectively reduce much of the difficulty of reasoning about container- and heap-manipulating programs to solving boolean combinations of linear inequalities over integers. Since modern DPLL($\mathcal{T}$)-based solvers for deciding the satisfiability of boolean combinations of linear integer inequalities employ a decision procedure for deciding conjunctions of linear integer inequalities, a practical technique for solving conjunctive linear integer inequalities is of paramount importance. However, as described in Section 8.1, existing techniques for solving linear integer inequalities do not perform well for many systems that arise in practice. In this chapter, we present a new algorithm called *Cuts-from-Proofs* that substantially outperforms existing approaches.

8.1 Introduction

A quantifier-free system of linear inequalities over integers is defined by $A\vec{x} \leq \vec{b}$ where A is an $m \times n$ matrix with only integer entries, and $\vec{b}$ is a vector in Z^n. This system has a solution if and only if there exists a vector $\vec{x^*} \in Z^n$ that satisfies $A\vec{x^*} \leq \vec{b}$. Determining the satisfiability of such a system of inequalities is a recurring theme in program analysis and verification. This problem arises not only in the symbolic heap analysis described in the previous chapters, but also in many other contexts such as array dependence analysis, buffer overrun analysis, and integer overflow checking

[31, 71] as well as in RTL datapath and symbolic timing verification [24, 17]. For this reason, many modern SMT solvers incorporate a dedicated linear arithmetic module for solving this important subclass of constraints [43, 34, 18, 25, 21].

While practical algorithms, such as Simplex, exist for solving linear inequalities over the reals [33], solving linear inequalities over integers is known to be an NP-complete problem, and existing algorithms do not scale well in practice. There are three main approaches for solving linear inequalities over integers. One approach first solves the *LP-relaxation* of the problem to obtain a rational solution and adds additional constraints until either an integer solution is found or the LP-relaxation becomes infeasible. The second approach is based on the Omega Test, an extension of the Fourier-Motzkin variable elimination for integers [71]. Yet a third class of algorithms utilize finite-automata theory [80, 46].

The algorithm presented in this chapter falls into the first class of techniques described above. Existing algorithms in this class include *branch-and-bound*, *Gomory's cutting planes* method, or a combination of both, known as *branch-and-cut* [75]. Branch-and-bound searches for an integer solution by solving the two subproblems $A\vec{x} \leq \vec{b} \cup \{x_i \leq \lfloor f_i \rfloor\}$ and $A\vec{x} \leq \vec{b} \cup \{x_i \geq \lceil f_i \rceil\}$ when the LP-relaxation yields a solution with fractional component f_i. The original problem has a solution if at least one of the subproblems has an integer solution. Even though upper and lower bounds can be computed for each variable to guarantee termination, this technique is often intractably slow on its own. Gomory's cutting planes method computes *valid inequalities* that exclude the current fractional solution without excluding feasible integer points from the solution space. Unfortunately, this technique has also proven to be impractical on its own and is often only used in conjunction with branch-and-bound [69].

All of these techniques suffer from a common weakness: While they exclude the current fractional assignment from the solution space, they make no systematic effort to exclude the cause of this fractional assignment. In particular, if the solution of the LP-relaxation lies at the intersection of n planes defined by the initial set of inequalities, and $k \leq n$ of these planes have an intersection that contains no integer points, then it is desirable to exclude at least this entire $n - k$ dimensional subspace.

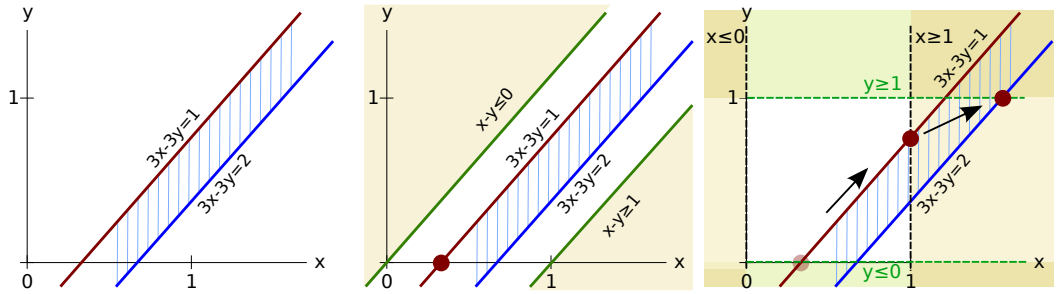


Figure 8.1: (a) The projection of Equation 8.1 onto the xy plane. (b) The green lines indicate the closest lines parallel to the proof of unsatisfiability; the red point marks the solution of the LP-relaxation. (c) Branch-and-bound first adds the planes $x = 0$ and $x = 1$, then the planes $y = 0$ and $y = 1$, and continues to add planes parallel to the coordinate axes.

The key insight underlying our approach is to systematically discover and exclude exactly this $n - k$ dimensional subspace rather than individual points that lie on this space. To be concrete, consider the following system with no integer solutions:

$$\begin{aligned} -3x + 3y + z &\leq -1 \\ 3x - 3y + z &\leq 2 \\ z &= 0 \end{aligned} \tag{8.1}$$

The projection of this system onto the xy plane is shown in Figure 8.1a. Suppose the LP-relaxation of the problem yields the fractional assignment $(x, y, z) = (\frac{1}{3}, 0, 0)$. The planes

$$\begin{aligned} z &= 0 \\ -3x + 3y + z &= -1 \end{aligned} \tag{8.2}$$

are the *defining constraints* of this vertex because the point $(\frac{1}{3}, 0, 0)$ lies at the intersection I of these planes. Since I contains no integer points, we would like to exclude exactly I from the solution space. Our technique discovers such intersections with no integer points by computing *proofs of unsatisfiability* for the defining constraints. A

proof of unsatisfiability is a single equality that (i) has no integer solutions and (ii) is implied by the defining constraints. In our example, a proof of unsatisfiability for I is $-3x + 3y + 3z = -1$ since it has no integer solutions and is implied by Equation 8.2. Such proofs can be obtained from the *Hermite normal form* of the matrix representing the defining constraints.

Once we discover a proof of unsatisfiability, our algorithm proceeds as a semantic generalization of branch-and-bound. In particular, instead of branching on a fractional component of the solution, our technique branches around the proof of unsatisfiability, if one exists. In our example, once we discover the equation $-3x+3y+3z = -1$ as a proof of unsatisfiability, we construct two new subproblems:

$$\begin{array}{rcl} -3x + 3y + z & \leq & -1 \\ 3x - 3y + z & \leq & 2 \\ z & = & 0 \\ -x + y + z & \leq & -1 \end{array} \qquad\qquad \begin{array}{rcl} -3x + 3y + z & \leq & -1 \\ 3x - 3y + z & \leq & 2 \\ z & = & 0 \\ -x + y + z & \geq & 0 \end{array}$$

where $-x+y+z = -1$ and $-x+y+z = 0$ are the closest planes parallel to and on either side of $-3x+3y+3z = -1$ containing integer points. As Figure 8.1b illustrates, neither of these systems have a real-valued solution, and we immediately determine the initial system to be unsatisfiable. In contrast, as shown Figure 8.1c, branch-and-bound only adds planes parallel to the coordinate axes, repeatedly yielding points that lie on either $3x - 3y = 1$ or $3x - 3y = 2$, neither of which contains integer points. On the other hand, Gomory's cutting planes technique first derives the valid inequality $y \geq 1$ before eventually adding a cut that makes the LP-relaxation infeasible. Unfortunately, this technique becomes much less effective in identifying the cause of unsatisfiability in higher-dimensions.

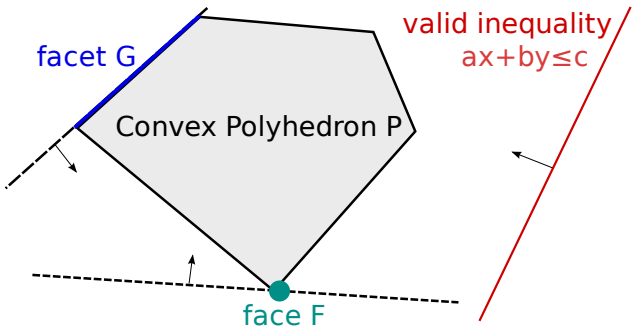


Figure 8.2: A convex polyhedron of dimension 2

8.2 Technical Background

8.2.1 Polyhedra, Faces, and Facets

In this section, we review a few standard definitions from polyhedral theory. The interested reader can refer to [69] for an in-depth discussion.

Definition 1 (Convex Polyhedron) *The set of (real-valued) solutions satisfying* $A\vec{x} \leq \vec{b}$ *describes a convex polyhedron* P*. The dimension* $dim(P)$ *of* P *is one less than the maximal number of affinely independent points in* P*.*

Definition 2 (Valid Inequality) *An inequality* $\pi\vec{x} \leq \pi_0$ *defined by some row vector* π *and a constant* π_0 *is a valid inequality for a polyhedron* P *if it is satisfied by all points in* P*.*

Definition 3 (Faces and Facets) F *is a face of polyhedron* P *if* $F = \{\vec{x} \in P : \pi\vec{x} = \pi_0\}$ *for some valid inequality* $\pi\vec{x} \leq \pi_0$*. A facet is a face of dimension* $dim(P) - 1$*.*

In Figure 8.2, polyhedron P has dimension 2 because there exist exactly 3 affinely independent points in P. The equation $ax + by \leq c$ is a valid inequality since all points in P satisfy this inequality. The point F is a face with dimension 0 since it is the intersection of P with the valid inequality represented by the dashed line. The line segment G is a facet of P since it is a face of dimension 1.

8.2.2 Linear Diophantine Equations

Definition 4 (Linear Diophantine Equation) *A linear equation of the form* $\sum a_i x_i = c$ *is diophantine if all coefficients* a_i *are integers and* c *is an integer.*

We state the following well-known result [69]:

Lemma 5 *A linear diophantine equation* $\sum a_i x_i = c$ *has a solution if and only if* c *is an integral multiple of the greatest common divisor* $gcd(a_1, \ldots, a_n)$.

Example 15 *The equation* $3x + 6y = 1$ *has no integer solutions since* 1 *is not evenly divisible by* $3 = gcd(3, 6)$. *However,* $3x + 6y = 9$ *has integer solutions.*

Corollary 1 *Let* E *be a plane defined by* $\sum a_i x_i = c$ *with no integer solutions and let* $g = gcd(a_1, \ldots, a_n)$. *Then, the two closest planes parallel to and on either side of* E *containing integer points are* $\lfloor E \rfloor$ *and* $\lceil E \rceil$, *given by* $\sum \frac{a_i}{g} x_i = \lfloor c/g \rfloor$ *and* $\sum \frac{a_i}{g} x_i = \lceil c/g \rceil$ *respectively.*

This corollary follows immediately from Lemma 5 and implies that there are no integer points between E and $\lfloor E \rfloor$ as well as between E and $\lceil E \rceil$.

8.2.3 Proofs of Unsatisfiability and the Hermite Normal Form

Given a system $A\vec{x} = \vec{b}$ of linear diophantine equations, we can determine in polynomial time whether this system has any integer solutions using the *Hermite normal form* of A. [1] Below we briefly review key properties of the Hermite normal form; the interested reader is referred to [69] for a more in-depth discussion.

Definition 5 (Hermite Normal Form) *An* $m \times m$ *integer matrix* H *is said to be in Hermite normal form (HNF) if (i)* H *is lower triangular, (ii)* $h_{ii} > 0$ *for* $0 \leq i < m$, *and (iii)* $h_{ij} \leq 0$ *and* $|h_{ij}| < h_{ii}$ *for* $i > j$.[2]

[1] While it is possible to determine the satisfiability of a system of linear diophantine equalities in polynomial time, determining the satisfiability of a system of linear integer *inequalities* is NP-complete.

[2] There is no agreement in the literature on the exact definition of the Hermite Normal Form. The one given here follows the definition in [69].

Definition 6 (Unimodular Matrix) *An $n \times n$ matrix U is* unimodular *if it has only integer entries and* $|\det(U)|$ *is* 1.

We review the following well-known lemmas:

Lemma 6 *For any $m \times n$ matrix A with* $\text{rank}(A) = m$*, there exists an $n \times n$ unimodular matrix U such that*

$$AU = \left[\begin{array}{c|c} H & \vec{0} \end{array} \right]$$

and the matrix H is the unique Hermite normal form of A.

While we do not describe the algorithm for computing the Hermite normal form of A, we remark that there exists an efficient polynomial time for computing the Hermite normal form of any matrix A (see [42]). Finally, we also recall the following two well-known results [69]:

Lemma 7 *If H is the Hermite normal form of A, then $H^{-1}A$ contains only integer entries.*

Lemma 8 (Proof of Unsatisfiability) *The system $A\vec{x} = \vec{b}$ has an integer solution if and only if $H^{-1}\vec{b} \in Z^m$. If $A\vec{x} = \vec{b}$ has no integer solutions, there exists a row vector $\vec{r_i}$ of the matrix $H^{-1}A$ such that the corresponding entry $\frac{n_i}{d_i}$ of $H^{-1}b$ is not an integer. We call the linear diophantine equation $d_i\vec{r_i}\vec{x} = n_i$ with no integer solutions a proof of unsatisfiability of $A\vec{x} = \vec{b}$.*

If the equation $d_i\vec{r_i}\vec{x} = n_i$ is a proof of unsatisfiability of $A\vec{x} = \vec{b}$, then it is implied by the original system and does not have integer solutions.

Example 16 *Consider the defining constraints from the example in Section 8.1:*

$$\begin{aligned} z &= 0 \\ -3x + 3y + z &= -1 \end{aligned}$$

Here, we have:

$$A = \begin{bmatrix} 0 & 0 & 1 \\ -3 & 3 & 1 \end{bmatrix} \qquad \vec{b} = \begin{bmatrix} 0 \\ -1 \end{bmatrix} \qquad H = \begin{bmatrix} 1 & 0 \\ -2 & 3 \end{bmatrix}$$

$$H^{-1}A = \begin{bmatrix} 0 & 0 & 1 \\ -1 & 1 & 1 \end{bmatrix} \qquad H^{-1}\vec{b} = \begin{bmatrix} 0 \\ -\frac{1}{3} \end{bmatrix}$$

This system does not have an integer solution because $H^{-1}\vec{b}$ contains a fractional component, and the equation $-3x+3y+3z=-1$ is a proof of unsatisfiability for this system.

8.3 The Cuts-from-Proofs Algorithm

In this section, we present our algorithm for determining the satisfiability of the system $A\vec{x} \leq \vec{b}$ over integers. In the presentation of the algorithm, we assume that there is a procedure `lp_solve` that determines the satisfiability of $A\vec{x} \leq \vec{b}$ over the reals, and if satisfiable returns a vertex v at an extreme point of the polyhedron induced by $A\vec{x} \leq \vec{b}$. This assumption is fulfilled by standard exterior-point algorithms for linear programming, such as Simplex [33].

Definition 7 (Defining Constraint) *An inequality $\pi\vec{x} \leq \pi_0$ is a defining constraint of vertex $\vec{v}$ of the polyhedron induced by $A\vec{x} \leq \vec{b}$ if $\vec{v}$ satisfies the equality $\pi\vec{v} = \pi_0$ where π is a row of A and π_0 is the corresponding entry in $\vec{b}$.*

With slight abuse of terminology, we call $\pi\vec{x} = \pi_0$ a defining constraint whenever $\pi\vec{x} \leq \pi_0$ is a defining constraint.

8.3.1 Algorithm

Let A be the initial $m \times n$ matrix and let a_{max} be the entry with the maximum absolute value in A. Then, choose any α such that $\alpha \geq n \cdot |a_{max}|$.

1. Invoke `lp_solve`. If the result is unsatisfiable, return unsatisfiable. Otherwise, if vertex v returned by `lp_solve` is integral, return v.

2. Identify the defining constraints $A'\vec{x'} \leq \vec{b'}$ of v.

3. Determine if the system $A'\vec{x'} = \vec{b'}$ has any integer solutions, and, if not, obtain a proof of unsatisfiability as described in Section 8.2.3. [3]

4. There are two cases:

   Case 1: (Conventional branch-and-bound) If a proof of unsatisfiability does not exist (i.e., $A'\vec{x'} = \vec{b'}$ has integer solutions) or if the proof of unsatisfiability contains a coefficient greater than $\alpha \cdot gcd(a_1, \ldots, a_n)$, pick a fractional component f_i of v and solve the two subproblems:

$$\begin{array}{rcl} A\vec{x} & \leq & \vec{b} \\ v_i & \leq & \lfloor f_i \rfloor \end{array} \qquad\qquad \begin{array}{rcl} A\vec{x} & \leq & \vec{b} \\ -v_i & \leq & -\lceil f_i \rceil \end{array}$$

   Case 2: (Branch around proof of unsatisfiability) Otherwise, consider the proof of unsatisfiability $\Sigma a_i x_i = c$ of $A'\vec{x'} = \vec{b'}$ and let g be $gcd(a_1, \ldots, a_n)$. The system $A\vec{x} \leq \vec{b}$ has a solution if either of the two subproblems has a solution:

$$\begin{bmatrix} A \\ \frac{a_1}{g} \cdots \frac{a_n}{g} \end{bmatrix} \vec{x} \leq \begin{bmatrix} \vec{b} \\ \lfloor \frac{c}{g} \rfloor \end{bmatrix} \qquad\qquad \begin{bmatrix} A \\ -\frac{a_1}{g} \cdots -\frac{a_n}{g} \end{bmatrix} \vec{x} \leq \begin{bmatrix} \vec{b} \\ -\lceil \frac{c}{g} \rceil \end{bmatrix}$$

8.3.2 Discussion of the Algorithm

In the above algorithm, if `lp_solve` yields a fractional assignment, then either

(i) the intersection of the defining constraints does not have an integer solution or

(ii) the defining constraints do have an integer solution but `lp_solve` did not pick an integer assignment

[3] Recall that Lemma 6 defines the Hermite normal form of an $m \times n$ matrix A' when A' has full rank. If A' does not have full rank, observe that we can still compute a proof of unsatisfiability of $A'\vec{x'} = \vec{b'}$ by dropping redundant rows of the system.

In the latter case (i.e., (ii)), we simply perform conventional branch-and-bound around any fractional component of the assignment to find an integer point on this intersection. Observe that while the current intersection $A'\vec{x'} = \vec{b'}$ is guaranteed to contain an integer point, this integer point may or may not lie inside the polyhedron defined by $A\vec{x} \leq \vec{b}$. Thus, the existence of an integer solution to the system $A'\vec{x'} = \vec{b'}$ in case (1) of the algorithm does not guarantee the existence of an integer solution to the original system $A\vec{x} \leq \vec{b}$.

On the other hand (i.e., (i)), if the defining constraints do not admit an integer solution, the algorithm obtains a proof of unsatisfiability with maximum coefficient less than α, if one exists, and constructs two subproblems that exclude this intersection without missing any integer points in the solution space. The constant α ensures that case 2 in step 4 of the algorithm is invoked a finite number of times and guarantees that there is a minimum bound on the volume excluded from the polyhedron at each step of the algorithm. (See Section 8.3.3 for the relevance of α for termination.)

Branching around the two planes in case 2 of the algorithm guarantees that the intersection $A'\vec{x'} = \vec{b'}$ of the defining constraints is no longer in the polyhedra defined by the two new subproblems. However, there *may* still exist a strict subset of these defining constraints (i.e., a higher-dimensional subspace) whose intersection contains no integer points but is not excluded from the solution space of the new subproblems. The following example illustrates such a situation.

Example 17 *Consider the defining constraints* $x + y \leq 1$ *and* $2x - 2y \leq 1$. *Using Hermite normal forms to compute a proof of unsatisfiability for the system*

$$\begin{aligned} x + y &= 1 \\ 2x - 2y &= 1 \end{aligned}$$

yields $4x = 3$. *While* $4x = 3$ *is a proof of unsatisfiability for the intersection of* $x + y = 1$ *and* $2x - 2y = 1$, *the strict subset* $2x - 2y = 1$ *has a proof of unsatisfiability on its own (namely itself), and it is not implied by* $4x = 3$.

As this example illustrates, the proof of unsatisfiability of a set of constraints does not necessarily imply the proof of unsatisfiability of any subset of these constraints.

At first glance, this seems problematic because if the intersection of any subset of the defining constraints contains no integer solutions, we would prefer excluding this larger subspace represented by the smaller set of constraints. Fortunately, as stated by Lemma 11, the algorithm will discover and exclude this higher-dimensional intersection in a finite number of steps. We first prove the following helper lemmas:

Lemma 9 *Let* $C = \begin{bmatrix} A \\ B \end{bmatrix}$ *be an* $m \times n$ *matrix composed of* A *and* B*, and let* $\text{HNF}(C) = \begin{bmatrix} H_A & 0 \\ X & Y \end{bmatrix}$*. Then,* $\text{HNF}(A) = H_A$.

Proof 5 *Our proof uses the HNF construction outlined in [69]. Let* i *be a row that the algorithm is currently working on and let* i' *be another row such that* $i' < i$*. Then, by construction, any entry* $c_{i'j}$ *where* $j > i'$ *is* 0*. Since any column operation performed while processing row* i *adds a multiple of column* $k \geq i$ *to another column, entry* $c_{i'k}$ *must be* 0*. Thus, any column operation is idempotent on row* i'*.*

Using blockwise inversion to invert $HNF(C)$, it can be easily shown that:

$$HNF(C)^{-1} = \begin{bmatrix} H_A^{-1} & 0 \\ -Y^{-1}XH_A^{-1} & Y^{-1} \end{bmatrix}$$

Thus, it is easy to see that $HNF(C)^{-1}C = HNF(C)^{-1}\vec{b'}$ implies $HNF(A)^{-1}A = HNF(A)^{-1}\vec{b}$ if $\vec{b'}$ is obtained by adding entries to the bottom of $\vec{b}$. This is the case because both $HNF(C)^{-1}$ and $HNF(A)^{-1}$ are lower triangular matrices. Intuitively, this result states that if $A\vec{x} = \vec{b}$ has a proof of unsatisfiability, we cannot "lose" this proof by adding extra rows at the bottom of A.

Example 18 *Consider the constraints from Example 16. Suppose we add the additional constraint $x = 1$ at the bottom of matrix A. Then, we obtain:*

$$A = \begin{bmatrix} 0 & 0 & 1 \\ -3 & 3 & 1 \\ 1 & 0 & 0 \end{bmatrix} \qquad \vec{b} = \begin{bmatrix} 0 \\ -1 \\ 1 \end{bmatrix} \qquad H = \begin{bmatrix} 1 & 0 & 0 \\ -2 & 3 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$H^{-1}A = \begin{bmatrix} 0 & 0 & 1 \\ -1 & 1 & 1 \\ 1 & 0 & 0 \end{bmatrix} \quad H^{-1}\vec{b} = \begin{bmatrix} 0 \\ -\frac{1}{3} \\ 1 \end{bmatrix}$$

Clearly, $-3x + 3y + 3z = -1$ is still obtained as a proof of unsatisfiability from the second row of $H^{-1}A = H^{-1}b$.

Lemma 10 *Consider any proof of unsatisfiability $\Sigma a_i x_i = c$ of any subset of the initial system $A\vec{x} \leq \vec{b}$. Then, $\forall i.|a_i| \leq \alpha \cdot \gcd(a_1, \ldots, a_n)$.*

Proof 6 *The coefficients a_i are obtained from the matrix $H^{-1}A'$ where A' is a matrix whose rows are a subset of those of A. Recall from basic linear algebra $H^{-1} = \frac{1}{det(H)}\text{adj}(H)$ where $\text{adj}(H)$ is the classical adjoint of H. Let the notation $||A||$ denote $\max_{ij}|a_{ij}|$. It is shown in [78] that:*

$$||\text{adj}(H)|| \leq \det(H)$$

for any matrix H in Hermite normal form. Hence any coefficient c in H^{-1} satisfies $|c| \leq 1$, and the entries in $H^{-1}A'$ are therefore bound by $\alpha = n \cdot |a_{\max}|$. Since the proof of unsatisfiability is some row of $H^{-1}A'$ multiplied by some $d_i > 1$, $d_i \leq \gcd(a_1, \ldots, a_n)$ as d_i is a divisor of each a_i. Thus, any coefficient in the proof of unsatisfiability is bound by $\alpha \cdot \gcd(a_1, \ldots, a_n)$.

Using the above lemmas, we can now show the following result:

Lemma 11 *Let F be a k-dimensional face without integer points of the initial polyhedron P with $dim(P) = d$. Suppose* `lp_solve` *repeatedly returns vertices that lie on this face. The algorithm will exclude F from P in a finite number of steps.*

Proof 7 *Every time* `lp_solve` *yields a vertex that lies on F, the algorithm excludes from the search space the intersection of the current defining constraints; thus, the next time* `lp_solve` *yields a vertex, one of these constraints will no longer be defining. At some point, when* `lp_solve` *returns a vertex on F, its defining constraints will be exactly the $d - k$ of the original constraints defining F, along with new constraints that were added to the bottom of the matrix. By Lemma 9, the additional constraints preserve the proof of unsatisfiability of the original $d - k$ constraints. Furthermore, by Lemma 10, this proof of unsatisfiability will have coefficients with absolute value of at most $\alpha \cdot \gcd(a_1, \ldots, a_n)$. Thus, the algorithm will obtain a proof of unsatisfiability for F and exclude all of F from the solution space.*

As Lemma 11 elucidates, the Cuts-from-Proofs algorithm discovers any relevant face without integer points on a demand-driven basis without explicitly considering all possible subsets of the initial set of inequalities. This allows the algorithm to add exactly the relevant cuts while staying computationally tractable in practice.

8.3.3 Soundness and Completeness

It is easy to see that the algorithm given above is correct because it never excludes integer points in the solution space. For arguing termination, we can assume, as standard, that the polyhedron P is finite; if it is not, one can compute maximum and minimum bounds on each variable without affecting the satisfiability of the original problem (see, for example [75, 69]). The key observation is that the volume we cut off the polyhedron cannot become infinitesimally small over time as we add more cuts. To see this, observe that there is a finite set of normal vectors N for the planes added by the Cuts-from-Proofs algorithm. Clearly, this holds for planes added by case 1 of step 4 since all such planes are parallel to one of the coordinate planes. This fact also holds for planes added in case 2 of step 4 since the coefficients of the normal vectors

must be less than or equal to α. Since the set N of normal vectors is finite, the algorithm will either terminate or, at some point, it will have to add planes parallel to already existing ones. The following lemma states that these parallel planes are at least some minimal distance ϵ apart:

Lemma 12 (Progress) *Let E be a plane added by the Cuts-from-Proofs algorithm and let E' be another plane parallel to E, also added by the algorithm. Then, E and E' are at least some minimum distance $\epsilon > 0$ apart.*

Proof 8 *Let E be defined by $\vec{n} \cdot \vec{x} = c_1$ and E' be defined by $\vec{n} \cdot \vec{x} = c_2$. Since c_1 and c_2 are integers and $c_1 \neq c_2$, E and E' are a minimum $d = 1/\sqrt{n_1^2 + \ldots + n_k^2}$ apart. Since there are a finite number of non-parallel planes added by the algorithm, choose ϵ to be the minimum such d.*

Let $\vec{n} \in N$ be any normal vector along which the algorithm must eventually cut. Because P is finite, there is a finite distance δ we can move along $\vec{n}$ through P. Since the distance we move along $\vec{n}$ is at least ϵ, the algorithm can cut perpendicular to $\vec{n}$ at most δ/ϵ times. Hence, the algorithm must terminate.

8.4 Implementation

In Section 8.4.1, we first discuss improvements over the basic algorithm presented in Section 8.3; then, in Section 8.4.2, we discuss the details of our implementation.

8.4.1 Improvements and Empirical Observations

An improvement over the basic algorithm described in Section 8.3 can be achieved by selectively choosing the proofs of unsatisfiability that the algorithm branches on. In particular, recall from Lemma 11 that if `lp_solve` repeatedly returns vertices on the same face with no integer points, the algorithm will also repeatedly obtain the same proof of unsatisfiability. Thus, in practice, it is beneficial to delay branching on a proof until the same proof is obtained at least twice. This can be achieved by using case 1 in step 4 of the algorithm instead of case 2 each time a new proof is discovered.

Since few of these proofs appear repeatedly, this easy modification often allows the algorithm to exclude only the highest-dimensional intersection with no integer points without having to branch around additional intermediate proofs. In our experience, this optimization can improve running time up to a factor of 3 on some examples.

An important empirical observation about the algorithm is that the overwhelming majority ($> 99\%$) of the proofs of unsatisfiability do not result in true branching. In practice, one of the planes parallel to the proof of unsatisfiability often turns out to be a valid inequality, while the other parallel plane lies outside the feasible region, making its LP-relaxation immediately unsatisfiable. Thus, in practice, the algorithm only branches around fractional components of an assignment.

8.4.2 Implementation Details

Our implementation of the Cuts-from-Proofs algorithm is written in C++ and consists of approximately 5000 lines of code, including modules to perform various matrix operations as well as support for infinite precision arithmetic. The Cuts-from-Proofs algorithm is a key component of the Mistral constraint solver, which implements the decision procedure for the combined theory of integer linear arithmetic and uninterpreted functions. Mistral is used in the Compass program analysis system mentioned in earlier chapters for the purpose of solving real-world constraints that arise from modeling contents of container data structures.

Our Simplex implementation, used as the `lp_solve` procedure in the Cuts-from-Proofs algorithm, uses *Bland's rule* for pivot selection [75]. Mistral utilizes a custom-built infinite precision arithmetic library based on the GNU MP Bignum Library (GMP) [5]. Our library performs computation natively on 64-bit values until an overflow is detected, and then switches to GNU bignums. If no overflow is detected, our implementation results in less than 25% slow down over native word-level arithmetic. We also found the selective use of hand-coded SIMD instructions to improve performance of Simplex by approximately a factor of 2.

Our implementation for Hermite normal form conversion is based on the algorithm given in [28]. This algorithm uses the modulo reduction technique of [42] to control

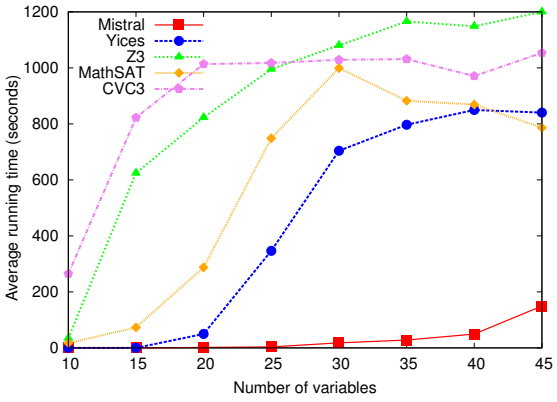
the number of required bits in any intermediate computation. In practice, the Hermite normal form conversion takes less than 5% of the overall running time and is not a bottleneck.

The implementation of the core Cuts-from-Proofs algorithm takes only about 250 lines of C++ code and does not require any features beyond what is discussed in this chapter. In our implementation, α was chosen to be $10n \cdot |a_{max}|$, and we have not observed the coefficients in the computed proofs of unsatisfiability to exceed this limit. In practice, the coefficients stay reasonably small.

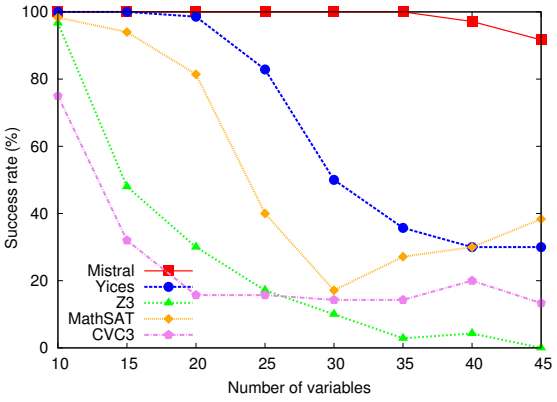
8.5 Experimental Results

To evaluate the effectiveness of the Cuts-from-Proofs algorithm, we compared Mistral with the four leading competitors (by score) in the QF-LIA category of SMT-COMP '08, namely Yices 1.0.16, Z3.2, MathSAT 4.2, and CVC3 1.5 obtained from [11]. We did not compare Mistral against (mixed) integer linear programming tools specialized for optimization problems. Existing tools such as GLPK [4], lp-solve [8], and CPLEX [2] all use floating point numbers instead of infinite precision arithmetic and yield unsound results for determining satisfiability even on small systems due to rounding errors. Furthermore, we did not use the QF-LIA benchmarks from SMT-COMP because they contain arbitrary boolean combinations of linear integer inequalities and equalities, making them unsuitable for comparing different algorithms to solve integer linear programs. The full set of test inputs and running times for each tool is available from `http://www.stanford.edu/~isil/benchmarks.tar.gz`. All experiments were performed on an 8 core 2.66 GHz Xeon workstation with 24 GB of memory. (All the tools, including Mistral, are single-threaded applications.) Each tool was given a maximum running time of 1200 seconds as well as 4 GB of memory. Any run exceeding the time or memory limit was aborted and marked as failure. If a run was aborted, its running time was assumed to be 1200 seconds for computing average running times.

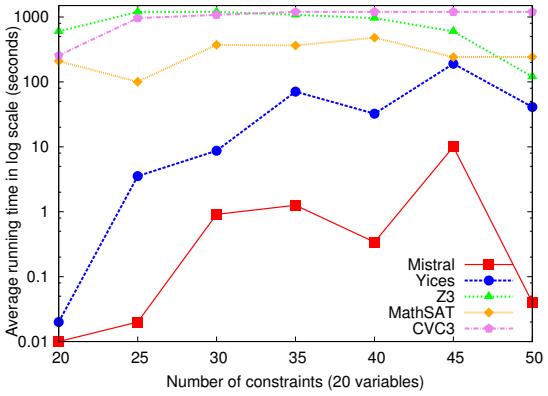
In the experiments, presented in Figure 8.3, we randomly generated more than 500 systems of linear inequalities, containing between 10 and 45 variables and between



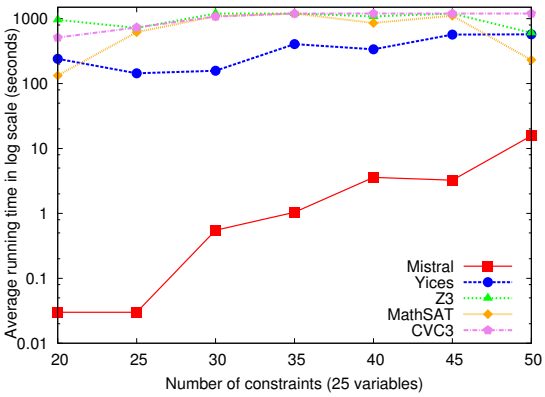
(a) Number of variables vs. average running time (15-50 inequalities)



(b) Number of variables vs. percent of successful runs (15-50 inequalities)



(c) Number of constraints vs. average running time in logarithmic scale for 20 variables



(d) Number of constraints vs. average running time in logarithmic scale for 25 variables

Figure 8.3: Experimental Results (fixed coefficient)

15 and 50 inequalities per system with a fixed maximum coefficient size of 5. Figure 8.3a plots the number of variables against the average running time over all sizes of constraints, ranging from 15 to 50. As is evident from this figure, the Cuts-from-Proofs algorithm results in a dramatic improvement over all existing tools. For instance, for 25 variables, Yices, Mistral's closest competitor, takes on average 347 seconds while Mistral takes only 3.45 seconds. This trend is even more pronounced in Figure 8.3b, which plots number of variables against the percentage of successful runs. For example, for 35 variables, Yices has a success rate of 36% while Mistral

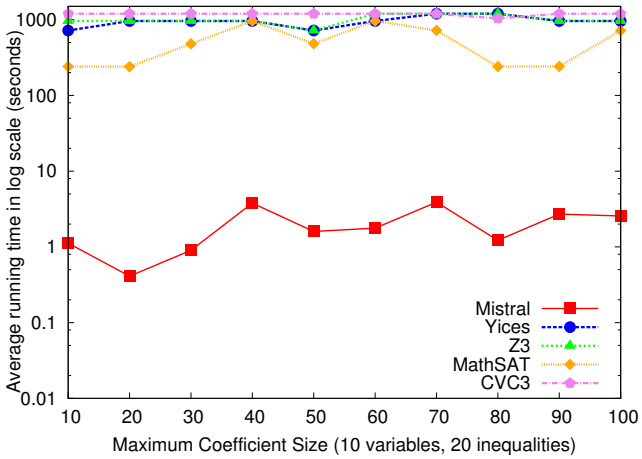
successfully completes 100% of its runs, taking an average of only 28.11 seconds.

Figures 8.3c and 8.3d plot the number of inequalities per system against average running time on a logarithmic scale for 20 and 25 variables, respectively. We chose not to present detailed breakouts for larger numbers of variables since such systems trigger time-out rates over 50% for all tools other than Mistral. These graphs demonstrate that the Cuts-from-Proofs algorithm reliably performs significantly, and usually at least an order of magnitude, better than any of the other tools, regardless of the number of inequalities per system.

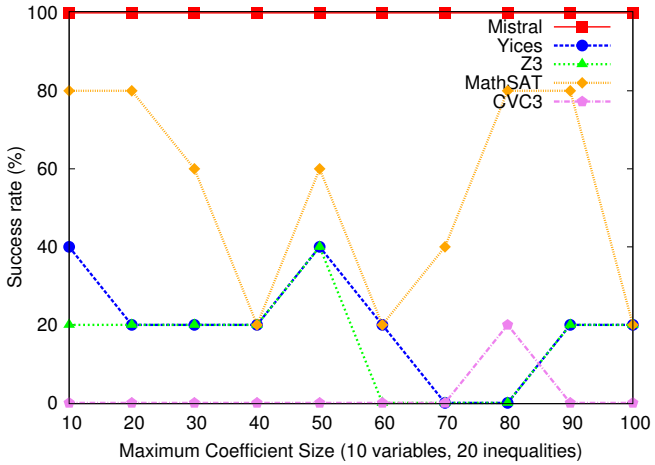
To evaluate the sensitivity of different algorithms to maximum coefficient size, we also compared the running time of different tools for coefficients ranging from 10 to 100 for systems with 10 variables and 20 inequalities. As shown in Figure 8.4, Mistral is less sensitive to coefficient size than the other tools. For example, for maximum coefficient 50, Mistral's closest competitor, MathSAT, takes an average of 482 seconds with a success rate of 60% while Mistral takes an average of 1.6 seconds with a 100% success rate.

Among the tools we compared, Yices and Z3 use a Simplex-based branch-and-cut approach, while CVC3 implements the Omega test. MathSAT mainly uses a Simplex-based algorithm augmented with the Omega test as a fallback mechanism. In our experience, one of the main differences between Simplex-based and Omega test based algorithms is that the former run out of time, while the latter run out of memory. On average, Simplex-based tools seem to perform better than tools using the Omega test.

We believe these experimental results demonstrate that the Cuts-from-Proofs algorithm outperforms leading implementations of existing techniques by orders of magnitude and significantly increases the size and complexity of integer linear programs that can be solved. Furthermore, our algorithm is easy to implement and does not require extensive tuning to make it perform well. We believe that the Cuts-from-Proofs algorithm can be profitably incorporated into existing SMT solvers that integrate the theory of linear integer arithmetic.



(a) Maximum coefficient vs. average running time for a 10 x 20 system



(b) Maximum coefficient vs. percent of successful runs for a 10 x 20 system

Figure 8.4: Experimental Results (fixed dimensions)

8.6 Related Work

As discussed in Section 8.1, there are three major approaches for solving linear inequalities over integers. LP-based approaches include branch-and-bound, Gomory's cutting planes method, and various combinations of the two [69, 75]. The cutting

planes method derives valid inequalities from the final Simplex tableau. More abstractly, a Gomory cut can be viewed as the proof of unsatisfiability of a single inequality obtained from a linear combination of the original set of inequalities. This is in contrast with our Cuts-from-Proofs algorithm which obtains a proof from the set of defining constraints, rather than from a single inequality in the final Simplex tableau. Unfortunately, the number of cuts added by Gomory's cutting planes technique is usually very large, and few of these cuts ultimately prove helpful in obtaining an integer solution [75]. Branch-and-cut techniques that combine branch-and-bound and variations on cutting planes techniques have proven more successful and are used by many state-of-the-art SMT solvers [43, 34, 25]. However, the algorithm proposed in this chapter significantly outperforms leading implementations of the branch-and-cut technique.

Another technique for solving linear integer inequalities is the Omega test, an extension of the Fourier-Motzkin variable elimination for integers [71]. A drawback of this approach is that it can consume gigabytes of memory even on moderately sized inputs, causing it to perform worse in practice than Simplex-based techniques.

A third approach for solving linear arithmetic over integers is based on finite automata theory [46]. Unfortunately, while complete, automata-based approaches perform significantly worse than all of the aforementioned techniques. The authors are not aware of any tools based on this approach that are currently under active development.

Another proposal [77] for solving linear arithmetic over integers is to translate the formula into an equisatisfiable boolean formula, whose satisfiability can then be checked using a standard boolean SAT solver. This technique is mainly targeted for special classes of ILP problems that arise frequently in verification where most of the constraints are difference constraints and each of the remaining non-difference constraints contains few variables.

Hermite normal forms are a well-studied topic in number theory, and efficient polynomial-time algorithms exist for computing Hermite normal forms [28, 78]. Their

application to solving systems of linear diophantine equations is discussed, for example, in [69, 75]. Jain et al. study the application of Hermite normal forms to computing interpolants of systems of linear diophantine equalities and disequalities [55]. We adopt the term "proof of unsatisfiability" from the literature on Craig interpolation [32, 67].

Chapter 9

Conclusion

In this thesis, we have presented static analysis techniques for precise and fully-automatic reasoning about the contents of an important class of data structures as well as a new constraint solving algorithm that makes such analyses practical.

The static analysis techniques we have described integrate reasoning about arrays and containers directly into a heap analysis, allowing much more precise points-to information for programs that manipulate heap objects through these data structures. Since most programs written in modern languages heavily make use of such abstract data structures, we believe the techniques presented in this thesis substantially extend the class of programs that can be automatically verified in a practical way.

The static analyses we have proposed leverage constraint solving algorithms by reducing some of the difficulty of analyzing container-manipulating programs to a combination of standard logic operations and integer constraints. Thus, our static analyses benefit directly from advances in constraint solving techniques. Towards this, we have also described a new algorithm called Cuts-from-Proofs for solving linear inequalities over integers that greatly improves the practicality of the proposed symbolic heap analysis techniques.

Bibliography

[1] C++ Standard Template Library. http://www.sgi.com/tech/stl/.

[2] CPLEX. http://www.ilog.com/products/cplex/.

[3] Digikam Photo Management Program. http://www.digikam.org/.

[4] GLPK GNU Linear Programming Kit. http://www.gnu.org/software/glpk/.

[5] GNU MP Bignum Library. http://gmplib.org/.

[6] Inkscape Vector Graphics Editor. http://www.inkscape.org/.

[7] LiteSQL C++ Object Persistence Framework. http://sourceforge.net/apps/trac/litesql/.

[8] lp_solve reference guide. http://lpsolve.sourceforge.net/5.5/.

[9] OpenSSH 5.3p1. http://www.openssh.com/.

[10] QT Framework. http://qt.nokia.com/products/.

[11] SMT-COMP'08. http://www.smtcomp.org.

[12] Toyota Prius brake problems. http://articles.cnn.com/2010-02-04/world/japan.prius.complaints_1_brake-system-anti-lock-prius-hybrid?_s=PM:WORLD .

[13] Toys R Us shoppers double charged. http://abclocal.go.com/wtvg/story?section=news/consumerid=7149701.

[14] Unix Coreutils. http://www.gnu.org/software/coreutils/.

[15] Wrong organ donations due to software bugs. http://www.dailymail.co.uk/news/article-1322081/Wrong-organs-removed-donors-glitch.html.

[16] Xavier Allamigeon. Non-disjunctive numerical domain for array predicate abstraction. In *Proceedings of the European Symposium on Programming (ESOP)*, pages 163–177, 2008.

[17] Tod Amon, Gaetano Borriello, Taokuan Hu, and Jiwen Liu. Symbolic timing verification of timing diagrams using presburger formulas. In *DAC '97: Proceedings of the 34th annual conference on Design automation*, pages 226–231, New York, NY, USA, 1997. ACM.

[18] Clark Barrett and Cesare Tinelli. CVC3. In Werner Damm and Holger Hermanns, editors, *Proceedings of the* 19^{th} *International Conference on Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer-Verlag, July 2007. Berlin, Germany.

[19] N. Blanc, A. Groce, and D. Kroening. Verifying C++ with STL Containers via Predicate Abstraction. In *IEEE/ACM Conference on Automated software engineering*, pages 521–524. ACM, 2007.

[20] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. *The Essence of Computation*, pages 85–108, 2002.

[21] Miquel Bofill, Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. The Barcelogic SMT Solver. In *Proceedings of the 20th international conference on Computer Aided Verification*, pages 294–298, Berlin, Heidelberg, 2008. Springer-Verlag.

[22] I. Bogudlov, T. Lev-Ami, T. Reps, and M. Sagiv. Revamping TVLA: Making parametric shape analysis competitive. *Lecture Notes in Computer Science*, 4590:221, 2007.

[23] A. Bradley, Z. Manna, and H. Sipma. Whats decidable about arrays? In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 427–442. Springer, 2006.

[24] R. Brinkmann and R. Drechsler. RTL-datapath verification using integer linear programming. In *Proceedings of the 2002 Asia and South Pacific Design Automation Conference*, pages 741–746. IEEE Computer Society, 2002.

[25] Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, and Roberto Sebastiani. The MathSAT 4 SMT Solver. In *Proceedings of the 20th International Conference on Computer Aided Verification (CAV)*, pages 299–303, Berlin, Heidelberg, 2008. Springer-Verlag.

[26] S. Chandra and T. Reps. Physical type checking for C. In *ACM SIGSOFT Software Engineering Notes*, volume 24, pages 66–75. ACM, 1999.

[27] D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *Programming Language Design and Implementation (PLDI)*, pages 296–310, NY, USA, 1990. ACM.

[28] Henri Cohen. *A Course in Computational Algebraic Number Theory*. Graduate Texts in Mathematics. Springer-Verlag, 1993.

[29] D.C. Cooper. Theorem proving in arithmetic without multiplication. *Machine Intelligence*, 7:91–100, 1972.

[30] P. Cousot. Verification by abstract interpretation. *Lecture notes in computer science*, pages 243–268, 2003.

[31] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Principles of Programming Languages (POPL)*, pages 84–96, NY, USA, 1978. ACM.

[32] W. Craig. Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. *The Journal of Symbolic Logic*, 22(3):269–285, 1957.

[33] George Dantzig. *Linear Programming and Extensions*. Princeton U. Press, 1963.

[34] Leonardo De Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 337–340, April 2008.

[35] A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *Pprogramming Language Design and Implementation (PLDI)*, pages 230–241. ACM NY, USA, 1994.

[36] I. Dillig, T. Dillig, and A. Aiken. Cuts from Proofs: A complete and practical technique for solving linear inequalities over integers. In *Computer Aided Verification (CAV)*, pages 233–247. Springer, 2009.

[37] I. Dillig, T. Dillig, and A. Aiken. SAIL: Static Analysis Intermediate Language. *Stanford University Computer Science Department Technical Report*, 2009.

[38] I. Dillig, T. Dillig, and A. Aiken. Fluid updates: Beyond strong vs. weak updates. *Programming Languages and Systems*, pages 246–266, 2010.

[39] I. Dillig, T. Dillig, and A. Aiken. Small formulas for large programs: On-line constraint simplification in scalable static analysis. *Static Analysis Symposium (SAS)*, pages 236–252, 2011.

[40] I. Dillig, T. Dillig, A. Aiken, and M. Sagiv. Precise and compact modular procedure summaries for heap manipulating programs. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, pages 567–577. ACM, 2011.

[41] D. Distefano, P. OHearn, and H. Yang. A local shape analysis based on separation logic. *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 287–302, 2006.

[42] P. Domich, R. Kannan, and Jr. L. Trotter. Hermite normal form computation using modulo determinant arithmetic. *Mathematics of Operations Research*, 12(1):50–59, February 1987.

[43] Bruno Dutertre and Leonardo De Moura. The Yices SMT Solver. Technical report, SRI International, 2006.

[44] S.J. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective typestate verification in the presence of aliasing. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 17(2):1–34, 2008.

[45] C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *Principles of Programming Languages (POPL)*, pages 191–202. ACM NY, USA, 2002.

[46] Vijay Ganesh, Sergey Berezin, and David Dill. Deciding presburger arithmetic by model checking and comparisons with other methods. In *FMCAD '02: Proceedings of the 4th International Conference on Formal Methods in Computer-Aided Design*, pages 171–186, London, UK, 2002. Springer-Verlag.

[47] Y. Ge and L. de Moura. Complete instantiation for quantified formulas in Satisfiabiliby Modulo Theories. In *Computer Aided Verification (CAV)*, pages 306–320. Springer, 2009.

[48] D. Gopan, T. Reps, and M. Sagiv. A framework for numeric analysis of array operations. In *Principles of Programming Languages (POPL)*, pages 338–350, NY, USA, 2005. ACM.

[49] D. Gregor and S. Schupp. STLlint: Lifting static checking from languages to libraries. *Software Practice and Experience*, 36(3):225–254, 2006.

[50] S. Gulwani, B. McCloskey, and A. Tiwari. Lifting abstract interpreters to quantified logical domains. In *Principles of Programming Languages (POPL)*, pages 235–246. ACM New York, NY, USA, 2008.

[51] S. Gulwani and M. Musuvathi. Cover algorithms and their combination. In *Proceedings of the Theory and practice of software, 17th European conference on Programming languages and systems*, pages 193–207. Springer-Verlag, 2008.

[52] Sumit Gulwani, Krishna K. Mehra, and Trishul Chilimbi. SPEED: Precise and efficient static estimation of program computational complexity. In *Principles of Programming Languages (POPL)*, pages 127–139, 2009.

[53] P. Habermehl, R. Iosif, and T. Vojnar. What else is decidable about integer arrays? *Foundations of Software Science and Computational Structures*, pages 474–489, 2008.

[54] N. Halbwachs and M. Péron. Discovering properties about arrays in simple programs. In *Programming Language Design and Implementation (PLDI)*, pages 339–348, NY, USA, 2008. ACM.

[55] Himanshu Jain, Edmund Clarke, and Orna Grumberg. Efficient craig interpolation for linear diophantine (dis)equations and linear modular equations. In *Proceedings of the 20th international conference on Computer Aided Verification*, pages 254–267, Berlin, Heidelberg, 2008. Springer-Verlag.

[56] R. Jhala and K. McMillan. Array abstractions from proofs. In *Computer Aided Verification (CAV)*, pages 193–206. Springer, 2007.

[57] N.D. Jones and S.S. Muchnick. Flow analysis and optimization of LISP-like structures. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 244–256. ACM, 1979.

[58] M. Karr. Affine relationships among variables of a program. *Acta Informatica*, 6(2):133–151, 1976.

[59] L. Kovács and A. Voronkov. Finding loop invariants for programs over arrays using a theorem prover. *Fundamental Approaches to Software Engineering*, pages 470–485, 2009.

[60] D. Kroening and O. Strichman. *Decision procedures: an algorithmic point of view*. Springer-Verlag New York Inc, 2008.

[61] V. Kuncak, P. Lam, K. Zee, and MC Rinard. Modular Pluggable Analyses for Data Structure Consistency. *IEEE Transactions on Software Engineering*, 32(12):988–1005, 2006.

[62] P. Lam, V. Kuncak, and M. Rinard. Generalized typestate checking for data structure consistency. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 430–447. Springer, 2005.

[63] P. Lam, V. Kuncak, and M. Rinard. Hob: A tool for verifying data structure consistency. In *Compiler Construction*, pages 137–138. Springer, 2005.

[64] W. Landi and B.G. Ryder. Safe approximate algorithm for interprocedural pointer aliasing. In *the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation (PLDI)*, pages 235–248, 1992.

[65] S.H. Lee and D.H. Cho. Packet-scheduling algorithm based on priority of separate buffers for unicast and multicast services. *Electronics Letters*, 39(2):259–260, 2003.

[66] J. McCarthy. Towards a mathematical science of computation. *Information Processing*, 62:21–28, 1962.

[67] K.L. McMillan. Applications of Craig interpolants in model checking. *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 1–12, 2005.

[68] R. Monavich. *Partially Disjunctive Shape Analysis*. PhD thesis, Tel Aviv University, 2009.

[69] George L. Nemhauser and Laurence Wolsey. *Integer and Combinatorial Optimization*. John Wiley & Sons, 1988.

[70] K. Nguyen, T. Nguyen, and S. Cheung. Peer-to-peer streaming with hierarchical network coding. In *Multimedia and Expo, 2007 IEEE International Conference on*, pages 396–399. IEEE, 2007.

[71] W. Pugh. The Omega Test: A fast and practical integer programming algorithm for dependence analysis. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 4–13, 1991.

[72] G. Ramalingam, A. Warshavsky, J. Field, D. Goyal, and M. Sagiv. Deriving Specialized Program Analyses for Certifying Component-client Conformance. In *Programming Language Design and Implementation (PLDI)*, page 94, 2002.

[73] T. W. Reps, S. Sagiv, and R. Wilhelm. Static program analysis via 3-valued logic. In *Computer Aided Verification (CAV)*, volume 3114 of *Lecture Notes in Comp. Sc.*, pages 15–30. Springer, 2004.

[74] J.C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, 2002.

[75] Alexander Schrijver. *Theory of Linear and Integer Programming*. J. Wiley & Sons, 1986.

[76] M.N. Seghir, A. Podelski, and T. Wies. Abstraction Refinement for Quantified Array Assertions. In *Static Analysis Symposium (SAS)*, pages 3–18. Springer-Verlag, 2009.

[77] S.A. Seshia and R.E. Bryant. Deciding quantifier-free presburger formulas using parameterized solution bounds. In *Logic in Computer Science, 2004. Proceedings of the 19th Annual IEEE Symposium on*, pages 100–109. IEEE, 2005.

[78] Arne Storjohann and George Labahn. Asymptotically fast computation of hermite normal forms of integer matrices. In *Proc. Int'l. Symp. on Symbolic and Algebraic Computation: ISSAC '96*, pages 259–266. ACM Press, 1996.

[79] A. Stump, C.W. Barrett, D.L. Dill, and J. Levitt. A decision procedure for an extensional theory of arrays. In *IEEE Symposium on Logic in Computer Science*, pages 29–37, 2001.

[80] P. Wolper and B. Boigelot. An automata-theoretic approach to Presburger arithmetic constraints. *Static Analysis*, pages 21–32, 1995.

[81] H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. O'Hearn. Scalable shape analysis for systems code. *Computer Aided Verification (CAV)*, pages 385–398, 2008.