Precise Reasoning for Programs Using Containers

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

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

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

Abstract

Containers are general-purpose data structures that provide functionality for inserting, reading, removing, and iterating over elements. Since many applications written in modern programming languages, such as C++ and Java, use containers as standard building blocks, precise analysis of many programs requires a fairly sophisticated understanding of container contents. In this paper, we present a sound, precise, and fully automatic technique for static reasoning about contents of containers. We show that the proposed technique adds useful precision for verifying real C++ applications and that it scales to applications with over 100,000 lines of code.

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

General Terms Languages, Verification, Experimentation

1. Introduction

Containers are a family of general-purpose abstract data structures that 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.

We propose 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

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

POPL'11, January 26–28, 2011, Austin, Texas, USA.

Copyright © 2011 ACM 978-1-4503-0490-0/11/01...$10.00

```
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 1. Example illustrating key features of the technique

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.

1.1 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 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 approach has been previously used for successful reasoning about array contents [1], and, as we shall see, our approach extends these benefits 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

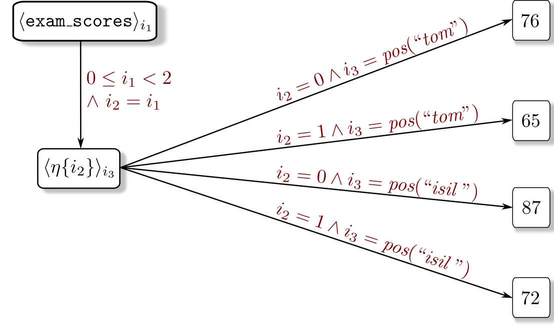

Figure 2. The representation of container `exam_scores` after the analysis of code from Figure 1

$\langle\alpha\rangle_i$ 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 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 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 2 shows a graphical representation of the facts established about the contents of `exam_scores` after analyzing the code from Figure 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 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.
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 $0 \leq i_1 < 2$.
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` asso-

ciates 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.

The rest of this paper is organized as follows: Section 2 gives a small language in which we formalize our technique. Section 3 describes the analysis and states the soundness theorem. Section 4 describes some extensions that are useful for modeling containers in real applications, and Section 5 discusses our implementation. Section 6 presents experimental results, Section 7 surveys related work, and, finally, Section 8 concludes. To summarize, this paper makes the following key contributions:

- We present a unified, sound, and precise technique for client-side reasoning about contents of an important family of data structures known as containers.
- We describe a fully automatic static analysis for containers that provides a detailed, per-element understanding of their contents and that supports arbitrary nestings of containers.
- We show experimentally that our technique is scalable enough to analyze C++ programs ranging between 16,000 to 128,000 lines of code that make heavy use of containers.
- We demonstrate experimentally that precise reasoning about contents of containers reduces false alarms by an order of magnitude compared to an analysis that treats containers as sets of values when verifying the absence of null dereferences, memory leaks, and deleted memory accesses in C++ programs.

2. Language and Concrete Semantics

We first introduce a simple statically-typed language used to formalize our technique:

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

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 $v_1.\texttt{read}(v_2)$ reads the value of element with key v_2 from container v_1, and $v_1.\texttt{write}(v_2,\ v_3)$ 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 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 3.

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

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

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

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

Figure 3. Type checking rules

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`.

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 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 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 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 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 contigu-

$$\frac{\begin{array}{l}E(v) = l\\ S(l,0) = l'\end{array}}{E,S,C \vdash v : l', S} \qquad \frac{}{E,S,C \vdash c : c, S} \qquad \frac{}{E,S,C \vdash \texttt{nil} : \texttt{NIL}, S} \qquad \frac{\begin{array}{c}l_n \notin \mathit{dom}(S)\\ S' = S[\forall i.(l_n, i) \leftarrow \texttt{NIL}]\end{array}}{E,S,C \vdash \texttt{new}\,\tau : l_n, S'} \qquad \frac{\begin{array}{l}E,S \vdash e : l, S'\\ E' = E[v \leftarrow l_n] \;\; (l_n \notin \mathit{dom}(S'))\\ S'' = S'[(l_n,0) \leftarrow l]\\ E', S'', C \vdash e' : l', S'''\end{array}}{E,S,C \vdash \texttt{let}\; v : \tau = e \;\texttt{in}\; e' : l',\; S''' \backslash l_n}$$

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

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

$$\frac{i = \mathit{Size}(\Delta)}{E,S,(i{::}C) \vdash \mathit{process}\;((v_1,v_2)\;\; \mathit{in}\;\; \Delta)\;\; \mathit{do}\; e : S} \qquad \frac{\begin{array}{l}E,S,C \vdash e_1 : l_1, S_1\\ E,S_1,C \vdash e_2 : l_2, S_2\end{array}}{E,S,C \vdash e_1; e_2 : l_2, S_2} \qquad \frac{\begin{array}{l}E(v) = l \;\; S(l,0) = l'\\ E,S,C \vdash e_1 : l_r, S' \;\; \mathit{if}\;\; l' \neq \texttt{NIL}\\ E,S,C \vdash e_2 : l_r, S' \;\; \mathit{if}\;\; l' = \texttt{NIL}\end{array}}{E,S,C \vdash \texttt{if}\; v \neq \texttt{nil}\;\texttt{then}\; e_1\;\texttt{else}\; e_2\;\texttt{fi} : l_r, S'}$$

Figure 4. Operational Semantics

ous region of memory whereas value-dependent 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 $[0, \mathit{size})$ 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.

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 3.1) and then discuss our abstract model of containers (Section 3.2). In Section 3.3, we present the analysis, and in Section 3.4, we state the soundness theorem.

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. As mentioned in Section 1, 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:

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

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), constraint (ϕ) pairs:

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

Here, 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 paper, 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 $\phi_1 \vee \phi_2$.

3.1.1 Bracketing Constraints

The constraints we require are more elaborate than we have indicated so far. Since most static analyses overapproximate program behavior, the reader may expect that the constraints ϕ used in the abstraction are overapproximations. In other words, if ϕ^* is a constraint describing some subset of elements in the container during a concrete execution, then $\phi^* \Rightarrow \phi$ if ϕ is an overapproximation. Now, our analysis relies crucially on negating constraints to model certain constructs well, specifically updates to containers and path

conditions. But, unfortunately, if ϕ is a *strict* overapproximation, then $\neg\phi$ is a strict underapproximation of $\neg\phi^*$, i.e., $\neg\phi^* \not\Rightarrow \neg\phi$. Clearly, for soundness, we need a negation operation that preserves overapproximations.

To solve this problem, all the constraints used in our abstraction are *bracketing constraints* $\langle\varphi_{may}, \varphi_{must}\rangle$, simultaneously representing over- and underapproximations of some set of concrete elements. A bracketing constraint is *well-formed* if $\varphi_{must} \Rightarrow \varphi_{may}$. The key benefit of bracketing constraints is that they preserve over- and underapproximations under negation:

$$\neg\langle\varphi_{may}, \varphi_{must}\rangle = \langle\neg\varphi_{must}, \neg\varphi_{may}\rangle$$

In other words, if φ_{may} and φ_{must} are over- and underapproximations for some fact F, then $\neg\varphi_{must}$ and $\neg\varphi_{may}$ are over- and underapproximations for $\neg F$ respectively. We briefly review some basic properties of bracketing constraints [1]:

$$\begin{array}{rcl}
\neg\langle\varphi_{may}, \varphi_{must}\rangle & = & \langle\neg\varphi_{must}, \neg\varphi_{may}\rangle \\
\langle\varphi_{may1}, \varphi_{must1}\rangle \wedge \langle\varphi_{may2}, \varphi_{must2}\rangle & = & \langle\varphi_{may1} \wedge \varphi_{may2}, \varphi_{must1} \wedge \varphi_{must2}\rangle \\
\langle\varphi_{may1}, \varphi_{must1}\rangle \vee \langle\varphi_{may2}, \varphi_{must2}\rangle & = & \langle\varphi_{may1} \vee \varphi_{may2}, \varphi_{must1} \vee \varphi_{must2}\rangle \\
SAT(\langle\varphi_{may}, \varphi_{must}\rangle) & \equiv & SAT(\varphi_{may}) \\
VALID(\langle\varphi_{may}, \varphi_{must}\rangle) & \equiv & VALID(\varphi_{must})
\end{array}$$

In this paper, any constraint ϕ is assumed to be a bracketing constraint unless explicitly stated otherwise. To make this clear, any time we do not use a bracketing constraint, we use the letter φ instead of ϕ. Furthermore, if the may and must conditions of a bracketing constraint are the same, we write a single constraint instead of a pair. For example, we abbreviate the bracketing constraint $\langle i = k, i = k\rangle$ as $i = k$. The constraints φ_{may} and φ_{must} we use inside bracketing constraints in this paper belong to the combined theory of linear integer arithmetic and uninterpreted functions.

3.2 Abstract Model of Containers

As discussed in Section 1, 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.

3.2.1 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 1. **(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 2. **(Index Selection $\clubsuit$ 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 $\clubsuit$ 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 $\clubsuit$, 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.

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

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

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 $\phi'_j = \text{QE}(\exists I.\ (\phi_j \wedge \phi))$. 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, we use existential quantifier elimination in the general case. Furthermore, it is not required that this

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

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

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

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

Figure 5. Transformers not Directly Related to Containers

quantifier elimination procedure be exact since our technique uses bracketing constraints. In particular, since quantifier elimination in the theory of uninterpreted functions is not always exact, we may use quantifier-free over- and underapproximations [2].

EXAMPLE 1. *Consider the abstract value set*

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

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:*

$$\theta \bowtie (0 \le i \le 2) \ = \ \{(0, \langle \text{true}, \text{false}\rangle), (1, \langle \text{true}, \text{false}\rangle)\}$$

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

3.3 The Analysis

We describe the analysis as deductive rules of the form:

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

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 5), then we describe the abstract semantics for reading from, writing to, and allocating containers (Figure 6), and, finally, we give the abstract semantics of the `foreach` construct (Figure 7).

Most of the transformers presented in Figure 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 $\mathbb{S}$:

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

Read from Position Dependent Container

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

Read from Value Dependent Container

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

Newval

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

Update

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

Write to Value Dependent Container

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

Write to Position Dependent Container

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

Container Allocation

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

Figure 6. Abstract Semantics for Container Operations

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

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

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*.

3.3.1 Abstract Semantics for Container Operations

We now consider the abstract semantics for reading from containers, presented in Figure 6. In the first two rules of Figure 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 3.2.1. 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 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 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 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 2. *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
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:*

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

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:*

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

[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".

Key, value pair at kth Iteration for pos_adt

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

Key, value pair at kth Iteration for val_adt

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

Foreach

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

Fix

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

Figure 7. Abstract Semantics for Iterating over Containers

3.3.2 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 $j = \mathit{pos}(k)$ 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 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 3. *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 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 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 5:

DEFINITION 4. *(**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 5. *(**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 $(\pi_i, \langle\varphi'_{\text{may}}, \varphi'_{\text{must}}\rangle) \in \mathbb{S}_2(\delta)$:*

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

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 paper 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 [3, 4] and quantifier elimination [1, 5] can be used for finding invariants. In particular, our previous work on array analysis [1] presents an algorithmic way of finding such invariants in this domain, and we use the algorithm from [1] in our implementation.

3.3.3 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 $\mathbb{S}(\beta^p) = \{(\langle\eta_p\rangle_{i_1}, true)\}$. After the allocation at line 5 during some arbitrary k'th iteration of the loop, the abstract environment and stores are:

$$\mathbb{E}(\texttt{papers}) = \beta^p \;\; \mathbb{E}(\texttt{paper_scores}) = \beta^a$$
$$\mathbb{E}(\texttt{scores}) = \beta^d \;\; \mathbb{E}(\texttt{cur_paper}) = \beta^c$$

[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.

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

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 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:

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

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:

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

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.

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 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 $\mathit{id}(l, i)$ 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 4 to track $\mathit{id}(l, i)$ and $A(l, i)$, we assume this additional bookkeeping information is available. We can now define the abstraction function as follows:

DEFINITION 6. **(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:*

1. $(\langle\eta_\rho\{\vec{i_\rho}\}\rangle_{i_0^\rho}, \vec{i_\rho} = C \wedge i_0^\rho = k)$ if $e^\rho = \texttt{new}^\rho\ \texttt{pos_adt}\ \tau$
2. $(\langle\eta_\rho\{\vec{i_\rho}\}\rangle_{i_0^\rho}, \vec{i_\rho} = C \wedge i_0^\rho = \mathrm{pos}(k))$ if $e^\rho = \texttt{new}^\rho\ \texttt{val_adt}\ \tau$
3. $(\beta^\rho, \mathrm{true})$ otherwise

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

$$
\alpha(v) = \left\{ \begin{array}{ll}
(\mathit{NIL}, \mathit{true}) & \textit{if } v = \texttt{NIL} \\
(c, \mathit{true}) & \textit{if } v \textit{ is integer constant } c \\
\alpha(l, k) & \textit{if } v \textit{ is a memory location}(l, k)
\end{array} \right.
$$

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

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 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 8. **(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 7)
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 1. **(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 the appendix.

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.

5. Implementation

We have implemented the ideas presented in this paper in our Compass program verification framework for analyzing C and C++ programs. Compass utilizes a gcc and g++ based front-end called SAIL [6] which translates C and C++ code to a low-level representation, similar to 3-address code. Compass uses the Mistral SMT solver [7, 8] 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 [9].

6. Experimental Evaluation

To demonstrate the usefulness of the ideas presented in this paper, 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 [10], either directly as position- or value-dependent containers or by nesting them inside already annotated STL containers.

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 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 Section 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 $[0, \mathit{size})$ into a stack and a queue and asserts that the top of the stack is the last

	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 8. Experimental Results of the Case Study

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 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.

6.2 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 [11], 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) [12]. 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 [13].

Both LiteSQL and the Inkscape widget library use the C++ standard template library (STL), while Digikam uses container libraries of the QT framework [14]. 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 paper, while in the second configuration, we track which set of elements a

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 9. Proving memory safety properties

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 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 9, we provide the running times of the analyses both on 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 9, observe that the technique presented in this paper improves the precision of the analysis over the second configura-

tion 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 paper 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 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 9 illustrate that our technique adds useful precision without incurring significant extra computational resources.

7. Related Work

In this work, we share the goal of separating the verification of data structure implementations from the verification of their client-side use with the Hob verification framework [15–17]. 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 [17]. 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 [18], 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 [20], 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. We believe that the problem of understanding container contents and the problem of verifying the correct usage of an ADT interface are orthogonal and complementary.

The idea of using numeric constraints to specify elements of unbounded data structures goes back to [21], which uses this idea for tracking may-alias pairs in lists. In our previous work on analysis of array contents, we use the combination of indexed locations and bracketing constraints to model arrays [1]. While this paper also uses some of the same underlying technical machinery as in [1], the contributions of this work include a formalization of the differences and similarities between different kinds of containers, and a precise and uniform analysis framework that both leverages what is the same and expresses what is different about containers.

In this paper, we observe that a key requirement for precise reasoning about nested containers is to differentiate between containers allocated in different loop iterations. The necessity to distinguish allocations arising from the same expression also arises in other contexts, such as static race detection in [22]. The technique described in [22] also uses a vector of loop counters to distinguish between distinct allocations.

The work described in [23] 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 [24]. 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.

In this paper, we observe that differences among various position-dependent containers and different value-dependent containers are insignificant for developing a methodology for reasoning about their contents, although these differences may have significant performance implications. This paper does not address the problem of selecting efficient containers (which is considered in [25]) or estimating their computational complexity (addressed in [26]).

8. Conclusion

In this paper, we have presented a precise, scalable, and fully automatic technique for reasoning about contents of containers. We have demonstrated experimentally that precise client-side reasoning about containers is important for successful verification of memory safety properties in real C++ programs.

9. Acknowledgments

We would like to thank Roy Frostig for his help with extending our front-end for the C language to C++.

References

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

[2] Gulwani, S., Musuvathi, M.: Cover Algorithms. In: ESOP. (2008) 193–207

[3] Cousot, P., Halbwachs, N.: Automatic Discovery of Linear Restraints among Variables of a Program. In: POPL, ACM (1978) 84–96

[4] Karr, M.: Affine relationships among variables of a program. A.I. (1976) 133–151

[5] Kovacs, L., Voronkov, A.: Finding loop invariants for programs over arrays using a theorem prover. In: FASE 2009. (2009) 470–485

[6] Dillig, I., Dillig, T., Aiken, A.: SAIL: Static Analysis Intermediate Language. Stanford University Technical Report

[7] Dillig, I., Dillig, T., Aiken, A.: Cuts from Proofs: A Complete and Practical Technique for Solving Linear Inequalities over Integers. In: CAV. (2009)

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

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

[10] http://www.sgi.com/tech/stl/: C++ standard template library

[11] http://sourceforge.net/apps/trac/litesql/: LiteSQL

[12] http://www.inkscape.org/: Inkscape

[13] http://www.digikam.org/: Digikam

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

[15] Lam, P., Kuncak, V., Rinard, M.: Hob: A tool for verifying data structure consistency. In: Compiler Construction. 237–241

[16] Lam, P., Kuncak, V., Rinard, M.: Generalized Typestate Checking for Data Structure Consistency. In: VMCAI. (2005) 430–447

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

[18] Ramalingam, G., Warshavsky, A., Field, J., Goyal, D., Sagiv, M.: Deriving Specialized Program Analyses for Certifying Component-client Conformance. In: PLDI. (2002) 94

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

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

[21] Deutsch, A.: Interprocedural may-alias analysis for pointers: Beyond k-limiting. In: PLDI, ACM NY, USA (1994) 230–241

[22] Naik, M., Aiken, A.: Conditional Must not Aliasing for Static Race Detection. In: POPL. (2007) 338

[23] Fink, S., Yahav, E., Dor, N., Ramalingam, G., Geay, E.: Effective Typestate Verification in the Presence of Aliasing. TOSEM **17**(2) (2008) 1–34

[24] Reps, T.W., Sagiv, S., Wilhelm, R.: Static Program Analysis via 3-Valued Logic. In: CAV. (2004) 15–30

[25] Shacham, O., Vechev, M., Yahav, E.: Chameleon: Adaptive Selection of Collections. In: PLDI, ACM (2009) 408–418

[26] Gulwani, S., Mehra, K.K., Chilimbi, T.: Speed: Precise and efficient static estimation of program computational complexity. In: POPL. (2009) 127–139

[27] Gulwani, S., McCloskey, B., Tiwari, A.: Lifting abstract interpreters to quantified logical domains. In: POPL, ACM (2008) 235–246

Appendix: Proof of Soundness

In this section, we sketch the proof of soundness of the key rules from Section 3.3. The proof is a standard induction on the inference rules from Figures 5, 6, and 7. We only focus on the rules that involve containers.

A.1 Preliminaries

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

DEFINITION 9. *($\sigma(\theta)$) 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 10. *($\lceil\phi\rceil$, $\lfloor\phi\rfloor$) Let ϕ be the bracketing constraint $\langle\varphi_{\text{may}}, \varphi_{\text{must}}\rangle$. Then, $\lceil\phi\rceil = \varphi_{\text{may}}$ and $\lfloor\phi\rfloor = \varphi_{\text{must}}$.*

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 $(\pi, \mathit{false}) \in \theta$.

A.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:

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

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 $pos(k)$ and θ is computed as:

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

Now, we consider a write $v_1.\mathtt{write}(v_2, v_3)$ 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:

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

Let:

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

In the write rule from Figure 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:

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

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

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

where $\theta \wedge \phi$ is shorthand for conjoining ϕ with every constraint in θ, as described in Section 3.3.

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

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

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

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

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 $(\pi_v^*, \langle \textit{true}, * \rangle)$. Observe that $\sigma_i[k/i]$ must assign i_j to k. Hence, by using (*), we know that the constraint

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

evaluates to $\langle \textit{true}, * \rangle$ 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 $(\pi_v^*, \langle \textit{true}, * \rangle) \in \sigma_v^*(\mathbb{S}'(\delta) \bowtie \sigma_i[k/i])$, 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 $\forall \pi_v' \neq \pi_v^*.\; (\pi_v', \langle *, \textit{false} \rangle) \in \sigma_v^*(\mathbb{S}'(\delta) \bowtie \sigma_i[k/i])$, 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:

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

Now, assume 2a. From (*), we know that under assignment σ_{e_i}, $(\pi_{e_i}, \langle \textit{true}, * \rangle)$. 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 $\pi_k \neq i \Rightarrow \sigma_i(\lfloor \phi_k \rfloor) = \textit{false}$; hence the sufficient condition for $\theta_2 \diamondsuit i_j$ is *false*. Thus, $\lceil \neg(\theta_2 \diamondsuit i_j) \rceil = \textit{true}$, which implies $(\pi_{e_i}, \langle \textit{true}, * \rangle) \in \sigma_{e_i}(\mathbb{S}'(\delta) \bowtie \sigma_i)$, 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 $\textit{pos}(i)$ according to the definition of the abstraction function, and $\mathbb{S}'$ is computed as:

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

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 $\textit{key}_k$ be the concrete key visited during the k'th loop iteration such that $\alpha(\textit{key}_k) = (\textit{key}_k, \sigma_k)$ where σ_k gives interpretation to function symbols $\textit{pos}^\rho$. In the abstract semantics, the value set $\theta_{\textit{key}}$ during the k'th iteration is given by

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

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

But, under interpretation σ_k, we have:

$$\begin{array}{l} \pi_k = \textit{key}_k \Rightarrow \sigma_k(\textit{pos}^\rho) = k \\ \pi_k \neq \textit{key}_k \Rightarrow \sigma_k(\textit{pos}^\rho) \neq k \end{array}$$

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