Shape Analysis of Sets

Jan Reineke

Master's Thesis

Saarland University
Chair for Programming Languages
and Compiler Construction
Prof. Dr. R. Wilhelm

INTERREG IIIC/e-Bird
Workshop "Trustworthy Software" 2006
http://drops.dagstuhl.de/opus/volltexte/2006/698

Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbständig, ohne fremde Hilfe und ohne Benutzung anderer als der von mir angegebenen Quellen angefertigt zu haben. Alle aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche gekennzeichnet. Die Arbeit wurde noch keiner Prüfungsbehörde in gleicher oder ähnlicher Form vorgelegt.

Jan Reineke

Acknowledgements

Dank gilt an erster Stelle Jörg Bauer für die Betreuung der Arbeit. Seine Ratschläge und seine hartnäckige Kritik trugen sehr zum Fortschritt der Arbeit bei. Björn Wachter möchte ich für viele Diskussionen rund um den Themenkomplex und Verbesserungsvorschläge danken. Weiterhin gilt mein Dank Prof. Reinhard Wilhelm für die Vergabe des Themas und die Übernahme der Begutachtung. Nicht zuletzt danke ich meiner Familie, insbesondere meinen Eltern, meinem Bruder und meiner Oma, für ihre Unterstützung während des gesamten Studiums.

Abstract

Shape Analysis is concerned with determining "shape invariants", i.e. structural properties of the heap, for programs that manipulate pointers and heap-allocated storage. Recently, very precise shape analysis algorithms have been developed that are able to prove the partial correctness of heap-manipulating programs. We explore the use of shape analysis to analyze abstract data types (ADTs). The ADT Set shall serve as an example, as it is widely used and can be found in most of the major data type libraries, like STL, the Java API, or LEDA. We formalize our notion of the ADT Set by algebraic specification. Two prototypical C set implementations are presented, one based on lists, the other on trees. We instantiate a parametric shape analysis framework to generate analyses that are able to prove the compliance of the two implementations to their specification.

The scalability of shape analysis algorithms could be improved by modular analysis. Some types of aliasing are, however, preventing modular analysis. We investigate the negative effects of aliasing on set implementations. For this purpose we introduce RESET, a language with sets as primitives. We give two semantics for RESET that differ in the way sets are represented. One representation is idealized, the other makes a step towards the set implementations. After formally relating the two semantics, we develop a shape analysis for the second semantics of RESET. In a small case study we analyze a program that computes the intersection of two sets.

Finally, we deal with modular analysis in a more general sense. We briefly introduce the concept of modularity and discuss benefits of modular analysis. Earlier, we observed that aliasing can be harmful. We introduce some existing encapsulation schemes that restrict aliasing to allow for modular analysis. On this basis we discuss modular shape analysis and how our previous analyses relate to this.

Contents

1 Introduction

This thesis deals with Shape Analysis and the Abstract Data Type (ADT) Set. It has two main goals:

- To use Shape Analysis to prove that Set implementations written in C comply to an algebraic specification of the ADT Set.
- To investigate Modular Shape Analysis. Is it possible to modularly analyze programs using set implementations? Does this depend on the specific implementation?

Let us go into a little more detail: Shape Analysis [CWZ90, GH96, SRW99, SRW02] is concerned with determining "shape invariants", i.e. structural properties of the heap, for programs that manipulate pointers and heap-allocated storage. Formerly, it was primarily used to aid compilers. Knowledge about the structure of the heap allows to carry out several optimizations, for instance, compile-time garbage collection, better instruction scheduling and automatic parallelization.

Recently, more precise shape analysis algorithms have been developed that are able to prove the partial correctness of heap-manipulating programs. In [LARSW00] bubble-sort and insertion-sort procedures are analyzed. The analyses were able to infer that the procedures indeed returned sorted lists. They also successfully analyzed destructive list reversal and the merging of two sorted lists.

The analyses of [LARSW00] and our analyses are based on the Shape Analysis Framework presented in [SRW02]. Logical structures are used to represent the program state in this framework. The concrete semantics is specified in first-order logic. By interpreting the concrete semantics in a 3-valued domain sound and precise abstractions can be extracted automatically. We will formally describe the framework in Chapter 2.

Set implementations are widely used and can be found in most of the major data type libraries, like STL [MS96], the Java API [Mic04], or LEDA [MN99]. The ADT Set shall serve as an example of abstract data types. One of the main goals of this thesis is to show the partial correctness of set implementations using Shape Analysis. For this purpose we will formally define our notion of the ADT Set. As a motivation, we will first examine mathematical sets, because they share some key properties with the ADT that we want to define. Early efforts to formalize the notion of mathematical sets, now called *Naïve Set Theory* led to contradictions. The most famous of these is known as Russell's paradox. Several independent efforts were undertaken to overcome these problems. Russell and

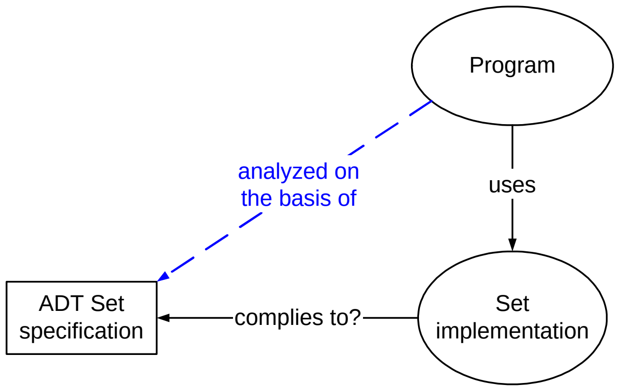


Figure 1.1: Modular Analysis

Whitehead proposed a solution in their *Principia Mathematica* introducing *Type Theory*. A hierarchy of types ensured that contradictions were prevented. Interestingly, such type restrictions are also useful when using sets as data abstractions in programming languages.

On this basis, we will go on to formally define the ADT Set using the algebraic specification [EM85, EM90, LEW97]. It shall serve as a reference for the implementations described later. Algebraic Specification allows us to express the intended behaviour independently of possible concrete implementations. Such specifications consist of a signature and a set of axioms. The axioms specify the meaning of the predicate and function symbols of the signature. The following two axioms are taken from our definition in Chapter 3:

$$a \in s.\texttt{insert}(b) \leftrightarrow a =_{el} b \vee a \in s, \quad (3)$$
$$a \in s.\texttt{remove}(b) \leftrightarrow a \neq_{el} b \wedge a \in s \quad (4)$$

They capture the effect of the $\cdot.\texttt{insert}(\cdot)$- and $\cdot.\texttt{remove}(\cdot)$-functions on the $\in$-predicate. Notice that they do not make any statement about the concrete data structures or algorithms employed.

After formally defining our notion of the ADT Set we will present two prototypical C implementations. One implementation is based on singly-linked lists, the other on binary trees. Using Shape Analysis, we will demonstrate that these implementations comply to our specification of the data type. This involves creating precise analyses using the framework of [SRW02] and linking the results to the specification of the ADT.

The second major question we deal with in this context is how to analyze programs using the ADT Set. Can we perform a modular analysis? What is a modular analysis? Modularity is an important concept in software engineering. Some of the advantages that a modular approach yields in the design process also translate to advantages of modular analyses. Figure 1.1 illustrates the idea of modular analysis in our particular setting. A

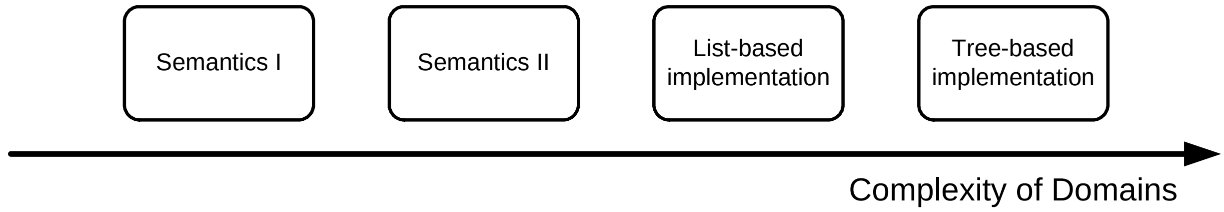


Figure 1.2: Complexity of Domains

conventional analysis would analyze the program as a whole including the set implementation. In a modular analysis we would divide this into two steps. In the first step we would show the compliance of the implementation to its specification. Then, we could analyze the program on the basis of the specification. This has several benefits. Usually, a specification is much simpler than its implementation. This yields smaller domains and could thus help to improve the scalability of shape analysis algorithms. In addition, we could then more easily distinguish between bugs in the program and bugs in the set implementation. Other aspects of modularity yield additional advantages that we will discuss in Chapter 7.

Unfortunately, it is not always possible to perform modular analyses. Problems arise, where modules are not completely separated from each other. A modular view requires that changes to the state of a module can only be made by calls to the interface. Often, this can not be guaranteed. When a memory location is reachable through different access paths, this is called aliasing. Aliasing allows to manipulate the heap at one place, causing problems at another. We claim that the extent of problems caused by aliasing rises with the complexity of the data structures employed. For instance, tree data structures suffer more than list structures. To further investigate this proposition, we create RESET, a language with sets as primitives. We specify two semantics for this language. Semantics I provides an idealized view of an implementation of the ADT Set defined in Chapter 3. Semantics II comes a little closer to the list- and tree-based set implementations. Figure 1.2 illustrates this.

Finally, we discuss some existing approaches to control aliasing in such a way that enables modular analysis. We also briefly investigate how modular shape analysis could look like.

1.1 Overview

In Chapter 1 we introduce the topic and give an overview of the thesis. We then go on to describe the foundations of the shape analysis framework underlying our analyses in Chapter 2. Here, we also give a brief description of TVLA, a tool that implements the framework. Chapter 3 consists of a formalization of the Abstract Data Type (ADT) Set. It is motivated by a short introduction to Mathematical Sets and serves as a basis for the

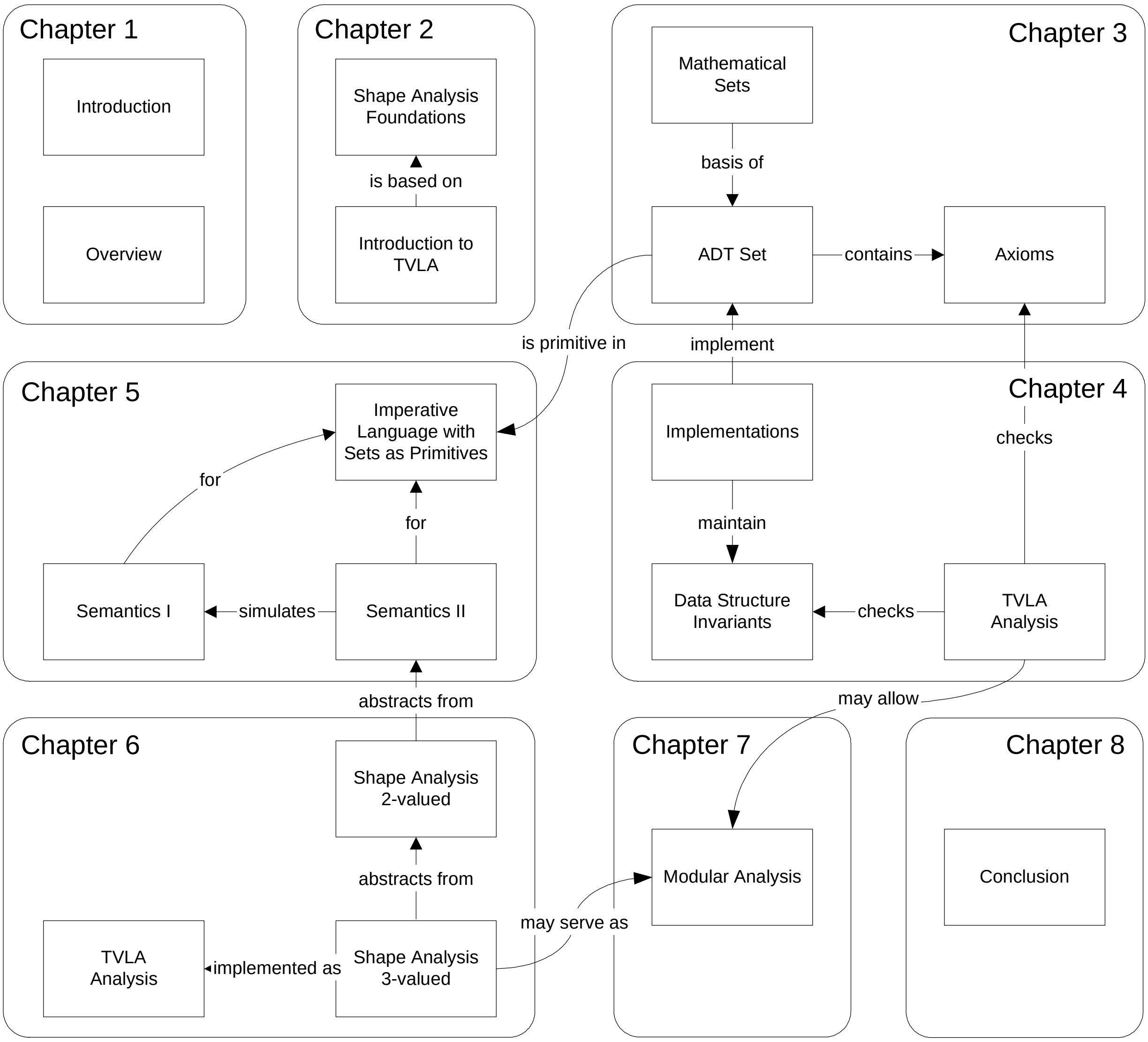


Figure 1.3: Structure of Thesis

following work.

In Chapter 4 we present two C implementations of the ADT Set defined in Chapter 3. We identify a number of data structure invariants specific to the implementations. Then we go on to present a shape analysis implemented in TVLA that checks two of the axioms of the ADT Set. The analyses rely on the data structure invariants to hold at entrance to the analyzed methods, but also show their maintenance throughout the execution of the methods. In Chapter 5 we introduce RESET, an imperative language with sets as primitives. Two semantics are given for this language and formally related. Chapter 6 builds on the second semantics of the previous chapter. We construct a shape analysis for it and use it to analyze a small program. In Chapter 7 we first explore modularity and modular analysis in a general sense. Then we investigate how a modular shape analysis could look like and how our previous shape analyses relate to this. Chapter 8 briefly summarizes the findings and discusses future work. Figure 1.3 illustrates the structure of the thesis.

Appendix A contains proofs of theorems and lemmas of Chapter 5. Source files of our implementations and shape analyses can be found in Appendix B.

2 Shape Analysis Foundations

2.1 Foundations of Shape Analysis

Shape Analysis is concerned with determining "shape invariants", i.e. structural properties of the heap, for programs that manipulate pointers and heap-allocated storage.

Our analyses fit into the Shape Analysis Framework introduced in [SRW02]. Their framework allows to specify the concrete semantics in first-order logic. By interpreting the concrete semantics in a 3-valued domain sound and precise abstractions can be extracted. We will therefore recapitulate the foundations before describing our analyses. For a more thorough treatment of these foundations consult [SRW02].

2.1.1 Concrete Semantics using 2-Valued Logic

Let $\mathcal{P} = \{p_1^{a(i)}, \ldots, p_n^{a(n)}\}$ be a set of predicate symbols. The arity of predicate $p_i^{a(i)}$ is $a(i)$.

Definition 1 (Syntax of First-Order Logic with Transitive Closure) *The set of first-order formulae with transitive closure over vocabulary $\mathcal{P}$, denoted $F(\mathcal{P})$, is defined inductively as follows:*

- **0** *and* **1** *are atomic formulae with no free variables.*
- $p_i^{a(i)}(v_1, \ldots, v_{a(i)})$ *is an atomic formula with free variables* $\{v_1, \ldots, v_{a(i)}\}$.
- $(v_1 = v_2)$ *is an atomic formula with free variables* $\{v_1, v_2\}$
- $\neg\phi_1, \phi_1 \wedge \phi_2, \phi_1 \vee \phi_2$ *are formulae with free variables* $V_1, V_1 \cup V_2, V_1 \cup V_2$, *respectively, if* ϕ_1 *and* ϕ_2 *are formulae with free variables* V_1 *and* V_2, *respectively.*
- $\forall v.\phi_1, \exists v.\phi_1$ *are formulae with free variables* $V_1 \setminus \{v\}$, *if* ϕ_1 *is a formula that has free variables* V_1.
- $(TCv_1, v_2 : \phi_1)(v_3, v_4)$ *is a formula with free variables* $(V_1 \setminus \{v_1, v_2\}) \cup \{v_3, v_4\}$, *where* V_1 *are the free variables of the formula* ϕ_1 *and* $v_3, v_4 \notin V_1$.

Definition 2 (2-valued Logical Structures) *A 2-valued logical structure (also called algebra) over vocabulary $\mathcal{P}$ is a tuple $S = \langle U^S, \iota^S \rangle$. The universe U^S is a set of individuals and ι^S maps each predicate symbol p^k to a truth-valued function: $\iota^S(p^k) : (U^S)^k \to \{0, 1\}$.*

We denote the set of 2-valued structures over vocabulary $\mathcal{P}$ by 2-STRUCT($\mathcal{P}$).

Logical Structure

indiv.	x
u_1	1

n	u_1
u_1	0

indiv.	x
u_1	1
u_2	0

n	u_1	u_2
u_1	0	1
u_2	0	0

indiv.	x
u_1	1
u_2	0
u_3	0

n	u_1	u_2	u_3
u_1	0	1	0
u_2	0	0	1
u_3	0	0	0

Graphical Representation

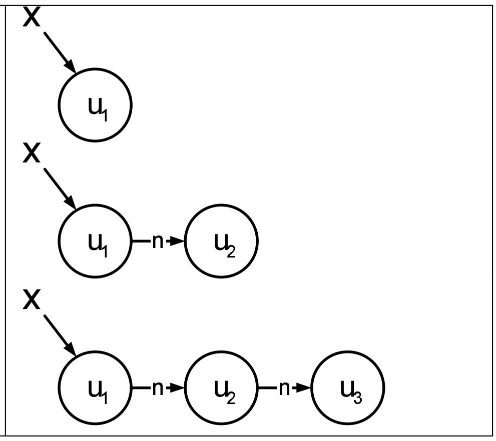


Figure 2.1: 2-valued logical structures representing lists of length l, with $1 \leq l \leq 3$

Such logical structures are used to represent the stores arising during the execution of programs. They can be graphically represented, following the intuition that unary predicates represent pointer variables and binary predicates represent pointer fields in the heap. A unary predicate is true for the particular heap cell the pointer variable points to. Binary predicates are true for those pairs of heap cells that are linked by the pointer field they represent. Figure 2.1 is an example of such a representation.

Definition 3 (Assignment) *An assignment (or valuation) β over a given structure $S = \langle U^S, \iota^S \rangle$ is a function that maps free variables to individuals: $\beta : \{v_1, \ldots v_n\} \rightarrow U^S$.*

Definition 4 (Meaning of Formulae) *The 2-valued meaning of a formula ϕ in a structure S under assignment β, denoted by $[\![\phi]\!]_2^S(\beta)$ is defined inductively. It yields a truth value in $\{0, 1\}$.*

$$[\![\mathbf{0}]\!]_2^S(\beta) = 0 \text{ and } [\![\mathbf{1}]\!]_2^S(\beta) = 1$$

$$[\![p_i^{a(i)}(v_1, \ldots, v_{a(i)})]\!]_2^S(\beta) = \iota^S(p_i^{a(i)})(\beta(v_1), \ldots, \beta(v_{a(i)}))$$

$$[\![v_1 = v_2]\!]_2^S(\beta) = \begin{cases} 1 & \beta(v_1) = \beta(v_2) \\ 0 & \beta(v_1) \neq \beta(v_2) \end{cases}$$

$$[\![\neg\phi_1]\!]_2^S(\beta) = 1 - [\![\phi_1]\!]_2^S(\beta)$$

$$[\![\phi_1 \wedge \phi_2]\!]_2^S(\beta) = min([\![\phi_1]\!]_2^S(\beta), [\![\phi_2]\!]_2^S(\beta))$$

$$[\![\phi_1 \vee \phi_2]\!]_2^S(\beta) = max([\![\phi_1]\!]_2^S(\beta), [\![\phi_2]\!]_2^S(\beta))$$

$$[\![\forall v.\phi_1]\!]_2^S(\beta) = \min_{u \in U^S} [\![\phi_1]\!]_2^S(\beta[v \mapsto u])$$

$$[\![\exists v.\phi_1]\!]_2^S(\beta) = \max_{u \in U^S} [\![\phi_1]\!]_2^S(\beta[v \mapsto u])$$

$$[\![(TC v_1, v_2 : \phi_1)(v_3, v_4)]\!]_2^S(\beta)$$
$$= \max_{n \geq 1, u_1, \ldots, u_{n+1} \in U^S, \beta(v_3) = u_1, \beta(v_4) = u_{n+1}} \min_{i \in \{1, \ldots, n\}} [\![\phi_1]\!]_2^S(\beta[v_1 \mapsto u_i, v_2 \mapsto u_{i+1}])$$

To express the effect of program statements predicate-update formulae are used. They relate the interpretation of the predicates in $\mathcal{P}$ after the execution of the statement to their interpretation before.

Definition 5 ($\mathcal{P}$ Transformer) *Let st be a program statement, and for every k-ary predicate p in vocabulary $\mathcal{P}$, let p'_{st} be the predicate-update formula for p at statement st over free variables $x_1, \ldots, x_k$. Then the $\mathcal{P}$* transformer associated with *st, denoted by $[\![st]\!] : 2\text{-}STRUCT[\mathcal{P}] \rightarrow 2\text{-}STRUCT[\mathcal{P}]$, is defined as follows.*

$$[\![st]\!](S) = \langle U^S, \lambda p.\lambda u_1, \ldots, u_k.[\![p'_{st}]\!]_2^S([x_1 \mapsto u_1, \ldots, x_k \mapsto u_k]) \rangle.$$

To analyze imperative programs in this setting they have to be translated into *Control Flow Graphs.*

Definition 6 (Control Flow Graph) *A* Control Flow Graph *is a tuple $G = \langle V(G), bg(G), As(G), Id(G), E(G), Tb(G), Fb(G) \rangle$, where*

- *$V(G)$ denotes the set of vertices of G,*
- *$bg(G) \in V(G)$ denotes the entrance vertex of G,*
- *$As(G) \subseteq V(G)$ denotes the set of assignment statements that manipulate the state,*
- *$Id(G) \subseteq V(G)$ denotes the set of statements that have no effect on the state as well as unconditional branch points,*
- *$E(G) \subseteq V(G) \times V(G)$ denotes the set of edges of the graph,*
- *$Tb(G) \subseteq E(G)$ denotes the set of edges that represents true branches,*
- *$Fb(G) \subseteq E(G)$ denotes the set of edges that represents false branches.*
- *$cond(w)$ denotes the formula for the program condition at w.*

Figure 2.2 shows a simple C program and a graphical representation of its corresponding *Control Flow Graph.*

Collecting Semantics. The goal of an analysis is to compute all possible structures arising at a given program point. This is formalized by *Collecting Semantics.* Let *ConcStructSet[v]* denote the (possibly infinite) set of structures that may arise on entry to v for

```
void insertElement(List* list, void* element)
{
  List* prev = 0;

  while (list != 0)
  {
    if (compare(list->data, element) == 0)
      return;

    prev = list;
    list = list->next;
  }
}
```

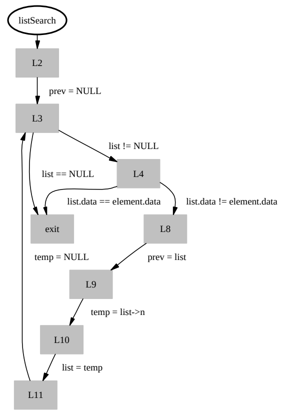


Figure 2.2: C program and corresponding Control Flow Graph

the set of input structures In. Then it can be defined as the least fixed pointed in terms of set inclusion of the following system of equations.

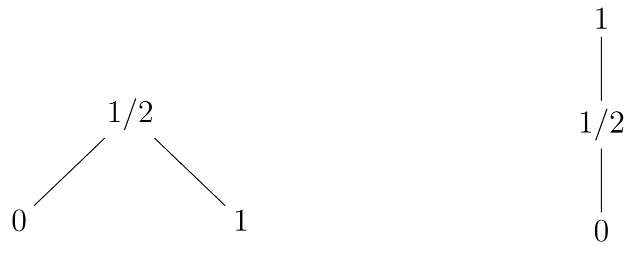
$$
ConcStructSet[v] = \begin{cases} In & \text{if } v = bg(G) \\ \left.\begin{array}{lll} & \bigcup\limits_{w\to v \in E(G),\ w \in As(G)} \{[\![st(w)]\!](S) \mid S \in ConcStructSet[w]\} & (1) \\ \cup & \bigcup\limits_{w\to v \in E(G),\ w \in Id(G)} \{S \mid S \in ConcStructSet[w]\} & (2) \\ \cup & \bigcup\limits_{w\to v \in Tb(G)} \{S \mid S \in ConcStructSet[w] \text{ and } S \models cond(w)\} & (3) \\ \cup & \bigcup\limits_{w\to v \in Fb(G)} \{S \mid S \in ConcStructSet[w] \text{ and } S \models \neg cond(w)\} & (4) \end{array}\right\} & \text{otherwise.} \end{cases}
$$

The effect of assignment statements is captured by (1). In (3) and (4) conditional branches are handled by transferring the specific structures the structures that fulfill the condition associated with the edge.

2.1.2 Abstract Semantics using 3-Valued Logic

As noted before, the *Collecting Semantics* defined above can yield infinite sets of structures. The least fixed point is not computable in general. In this section we show how 3-valued logical structures can be used to overcome this problem.

Figure 2.3 shows the semi-bilattice of 3-valued logic. In addition to the *definite* truth values 0 and 1 a third *indefinite* truth value 1/2 is introduced. The information order captures certainty of the information, i.e. 1/2 is less certain than 0 or 1. In the logical order $\wedge$ and $\vee$ are meet and join of the lattice. Figure 2.4 shows how $\wedge$ and $\vee$ are interpreted in the 3-valued domain.



(a) Information Order (b) Logical Order

Figure 2.3: The semi-bilattice of 3-valued logic

$\wedge$	0	1/2	1
0	0	0	0
1/2	0	1/2	1/2
1	0	1/2	1

$\vee$	0	1/2	1
0	0	1/2	1
1/2	1/2	1/2	1
1	1	1	1

Figure 2.4: Meaning of $\wedge$ and $\vee$ in the 3-valued domain

3-valued logical structures are defined similarly to their 2-valued counterparts:

Definition 7 (3-valued Logical Structures) *A 3-valued logical structure (also called algebra) over vocabulary $\mathcal{P}$ is a tuple $S = \langle U^S, \iota^S \rangle$. The universe U^S is a set of individuals and ι^S maps each predicate symbol p^k to a truth-valued function:*

$$\iota^S(p^k) : (U^S)^k \rightarrow \{0, 1, 1/2\}$$

We denote the set of 3-valued structures over vocabulary $\mathcal{P}$ by 3-STRUCT($\mathcal{P}$).

We assume every 3-valued logical structures to include a unary predicate sm. sm stands for "summary node". These are individuals of a 3-valued structure that possibly represent more than one individual in corresponding 2-valued structures. Using the sm predicate we can define the 3-valued meaning of formulae, denoted by $[\![\phi]\!]_3^S(\beta)$. It is defined inductively as in the definition for 2-valued structures, with the following difference:

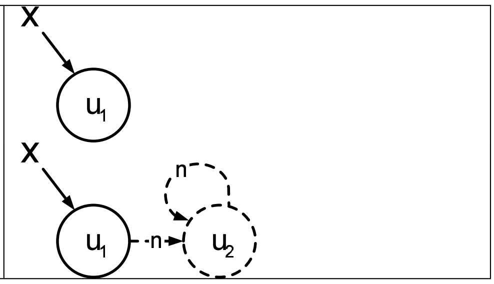
$$[\![v_1 = v_2]\!]_3^S(\beta) = \begin{cases} 0 & \beta(v_1) \neq \beta(v_2) \\ 1 & \beta(v_1) = \beta(v_1) \text{ and } \iota^S(sm)(\beta(v_1)) = 0 \\ 1/2 & \text{otherwise} \end{cases}$$

We say that S and β *potentially satisfy* ϕ, denoted by $S, \beta \models_3 \phi$, if $[\![\phi]\!]_3^S(\beta) = 1/2$ or $[\![\phi]\!]_3^S(\beta) = 1$. We write $S \models_3 \phi$ if for every β we have $S, \beta \models_3 \phi$.

Logical Structure	Graphical Representation

indiv.	x	sm
u_1	1	0

n	u_1
u_1	0



indiv.	x	sm
u_1	1	0
u_2	0	1/2

n	u_1	u_2
u_1	0	1/2
u_2	0	1/2

Figure 2.5: 3-valued logical structures representing the 2-valued structures of Figure 2.1

As in the case of 2-valued logical structures, 3-valued logical structures can be represented graphically. Figure 2.5 illustrates this. Summary nodes are identified by dashed lines.

In order to relate 2-valued and 3-valued structures we introduce the concept of *embedding*.

Definition 8 (Embedding Order) *Let $S = \langle U^S, \iota^S \rangle$ and $S' = \langle U^{S'}, \iota^{S'} \rangle$ be two structures, and let $f : U^S \to U^{S'}$ be a surjective function. We say that f embeds S in S', denoted by $S \sqsubseteq^f S'$ if (1) for every predicate symbol $p \in \mathcal{P} \cup \{sm\}$ of arity k and all $u_1, \ldots, u_k \in U^S$,*

$$\iota^S(p)(u_1, \ldots, u_k) \sqsubseteq \iota^{S'}(p)(f(u_1), \ldots, f(u_k))$$

and (2) for all $u' \in U^{S'}$

$$(|\{u | f(u) = u'\}| > 1) \sqsubseteq \iota^{S'}(sm)(u').$$

Condition (2) ensures that if several individuals from U^S are mapped to one individual in $U^{S'}$ than sm will be 1/2[1] in S' for that individual.

The definition of *Embedding* allows the predicates in S' to be less precise than they could be regarding their universe. A *tight embedding* minimizes the loss of information.

Definition 9 (Tight Embedding) *A structure $S' = \langle U^{S'}, \iota^{S'} \rangle$ is a* tight embedding *of $S = \langle U^S, \iota^S \rangle$ if there exists a surjective function $t\_embed : U^S \to U^{S'}$ such that, for every $p \in \mathcal{P}$ of arity k,*

$$\iota^{S'}(p)(u'_1, \ldots, u'_k) = \bigsqcup_{\substack{(u_1, \ldots, u_k) \in (U^S)^k, s.t. \\ t\_embed(u_i) = u'_i \in U^{S'}, 1 \le i \le k}} \iota^S(p)(u_1, \ldots, u_k)$$

and for every $u' \in U^{S'}$,

[1] It cannot be 1 because of condition (1), since $0 \not\sqsubseteq 1$.

$$\iota^{S'}(sm)(u') = (|\{u|\ t\_embed(u) = u'\}| > 1) \sqcup \bigsqcup_{\substack{u \in (U^S)^k, s.t. \\ t\_embed(u) = u' \in U^{S'}}} \iota^S(sm)(u).$$

Not only do the two embedding definitions define what it means for 2-valued structures to be embedded in 3-valued structures, they also define embedding of 3-valued structures in other 3-valued structures. Embedding allows to define the set of 2-valued concrete structures that a 3-valued structure represents:

$$\gamma(S) = \{S^\natural \in \text{2-STRUCT}[\mathcal{P}] \mid \text{there exists a function } f, \text{ s.t. } S^\natural \sqsubseteq^f S\}.$$

Theorem 1 (Embedding Theorem) *Let $S = \langle U^S, \iota^S \rangle$ and $S' = \langle U^{S'}, \iota^{S'} \rangle$ be two structures, and let $f : U^S \to U^{S'}$ be a function such that $S \sqsubseteq^f S'$. Then, for every formula ϕ and complete assignment β for ϕ, $[\![\phi]\!]_3^S(\beta) \sqsubseteq [\![\phi]\!]_3^S(f \circ \beta)$.*

Proof:

See [SRW02]. ■

Program analysis can benefit from this theorem. It ensures that any information extracted from an abstract 3-valued S' via a formula ϕ is a conservative approximation of the information extracted from any concrete 2-valued structure S embedded in S'. In particular, a definite value for ϕ in S' means that ϕ yields the same definite value in all $S \in \gamma(S)$.

The number of 3-valued structures above is still unbounded. One way of guaranteeing termination[2] of a program analysis is to operate on a finite domain. Monotonicity of the updates then ensures termination.

Definition 10 (Bounded Structure) *A bounded structure over vocabulary $\mathcal{P} \cup \{sm\}$ is a structure $S = \langle U^S, \iota^S \rangle$ such that for every $u_1, u_2 \in U^S$, where $u_1 \neq u_2$, there exists an abstraction predicate symbol $p \in \mathcal{A}$ such that $\iota^S(p)(u_1) \neq \iota^S(p)(u_2)$. Let* B-STRUCT$[\mathcal{P} \cup \{sm\}]$ *denote the set of such structures.*

This definition limits the size of the universes to $|U^S| \leq 3^{|\mathcal{A}|}$. Every abstraction predicate can take any of the three truth values for every individual. Canonical Abstraction is a way to obtain such bounded structures.

Definition 11 (Canonical Abstraction) *The canonical abstraction of a structure S, denoted by $t\_embed_c(S)$, is the tight embedding induced by the following mapping.*

$$t\_embed_c(u) = u_{\{p \in \mathcal{A} | \iota^S(p)(u)=1\}, \{p \in \mathcal{A} | \iota^S(p)(u)=0\}}$$

[2] One could also demand a finite height lattice, which need not be of finite size. Alternatively, widenings and narrowings can be used to ensure termination if the lattice is not of finite height.

Predicate	Defining Formula	Intended Meaning
$is[n](v)$	$\exists v_1, v_2.(v_1 \neq v_2 \wedge n(v_1, v) \wedge n(v_2, v))$	v is shared.
$c[n](v)$	$\exists v_1.(n(v_1, v) \wedge n^*(v_1, v_2))$	v resides on a cycle.
$r[n, x](v)$ for each $x \in Var$	$\exists v_1.(x(v_1) \wedge n^*(v_1, v))$	v is reachable from x via *next*-fields.

Table 2.1: Examples of Instrumentation Predicates

"$u_{\{p \in \mathcal{A} | \iota^S(p)(u)=1\},\{p \in \mathcal{A} | \iota^S(p)(u)=0\}}$" is known as the *canonical name* of individual u.

Instrumentation Predicates. Instrumentation predicates can be used to improve the precision of an analysis. They are predicates defined in terms of core predicates. Core predicates are those that are used to define the semantics of statements. We call the set of core predicates $\mathcal{C}$. Then the set of predicates $\mathcal{P}$ is disjointly partitioned into $\mathcal{C}$ and the set of instrumentation predicates $\mathcal{I}$.

Table 2.1 shows some examples of instrumentation predicates and their defining formulae. There are several ways in which instrumentation predicates can increase the precision of an analysis:

1. Evaluating the defining formulae of instrumentation predicates may yield definite values, while the evaluation on the core predicates evaluates to 1/2. In the example structure in Figure 2.6 we might ask whether everything reachable from y is also reachable from x. Without the reachability predicates this question could not be answered.

2. There are less concrete structures represented by a 3-valued structure if instrumentation predicates have definite values. If $c[n](v)$ is false for all elements of a list, then only concrete structures with acyclic lists are represented by the structure.

3. Instrumentation predicates can be used as abstraction predicates, keeping more precise information about parts of the heap. This is also depicted in Figure 2.6. If the reachability predicates $r[n, x]$ and $r[n, y]$ were not used as abstraction predicates the two summary nodes would be collapsed.

In order to gain more precise analyses through instrumentation predicates, it is usually necessary to devise special update formulae for these predicates. A simple way of creating update-formulae is simply evaluating the defining formula on the updated core predicates. This approach may yield very imprecise answers though. Most of the times an update formula can rely on the previous value of the instrumentation predicate to achieve a more precise result. In [RSL03] an approach is presented to automatically generate precise update formulae.

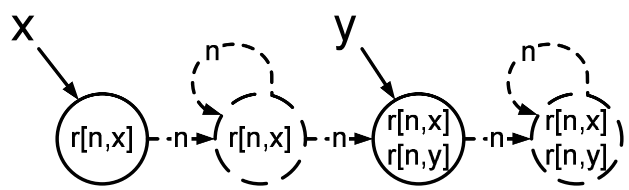


Figure 2.6: Example Structure with Instrumentation Predicates

One can define a collecting semantics similar to the one for the 2-valued semantics. This yields very imprecise analyses however. The reason for this is that the abstraction is usually not suited for the update formulae. The heap nodes manipulated are often part of summary nodes. If we wanted to update the structure shown in Figure 2.6 for the statement `y = y->n`, y would be indefinite and point into the summary node on the right.

Focus. The *focus* operation tackles this problem. The idea is to make those parts of the heap that are being manipulated concrete. Formally, the *focus* operation takes a set of formulae and a set of structures and returns a set of structures. The resulting set of structures should represent the same concrete structures as the input structures. In addition the set of formulae provided should evaluate to definite values on the input formulae. In general, an infinite number of structures may be necessary to fulfill this task. [SRW02] presents an algorithm that computes *focus* for an interesting class of formulae.

Besides implementing *focus* the question is which formulae to focus on. When applying an update formula, it is necessary that those parts of the heap that are manipulated have definite values. This can be characterized by the L-values of the left-hand side and the R-values of the right-hand side of a statement. For `y = y->n` this is $\exists v_1 : y(v_1) \wedge n(v_1, v)$. Applying *focus* on the structure in Figure 2.6 yields the three structures shown in Figure 2.7.

These structures allow us to apply the update-formulae for the statement to gain more precise results than before. The results are displayed in Figure 2.8. The figure illustrates a problem arising when using working in the 3-valued domain. It is possible to generate structures that represent no concrete structures. This is the case for the first structure. Some predicates are less precise than they could be regarding the information stored in the instrumentation predicates. The second and third structure in the figure fall into this category. We know that y is functional, that is it can only point to one heap cell at a time. We can also exclude the n-pointer from right to left in the third structure. It would imply a shared heap cell, which is not the case since $is[n]$ is false.

Coerce. The *coerce* operation sharpens such structures and eliminates structures that do not represent any concrete structures. It uses *compatibility constraints* to do so.

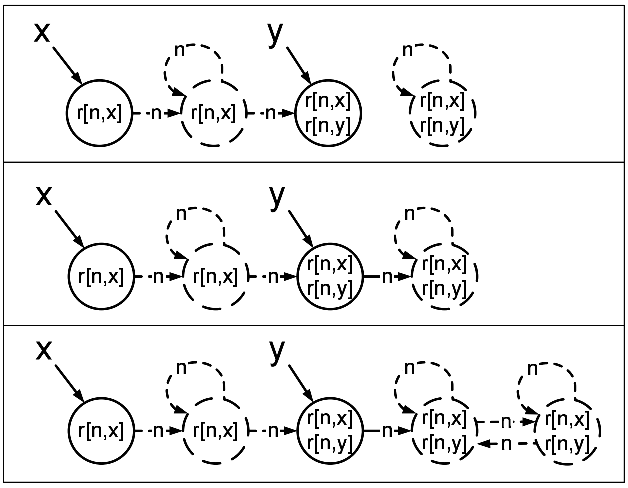


Figure 2.7: Structures after Focus

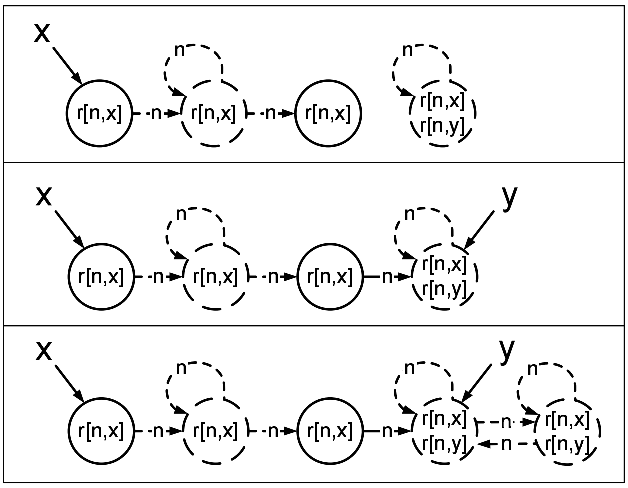


Figure 2.8: Structures after Update

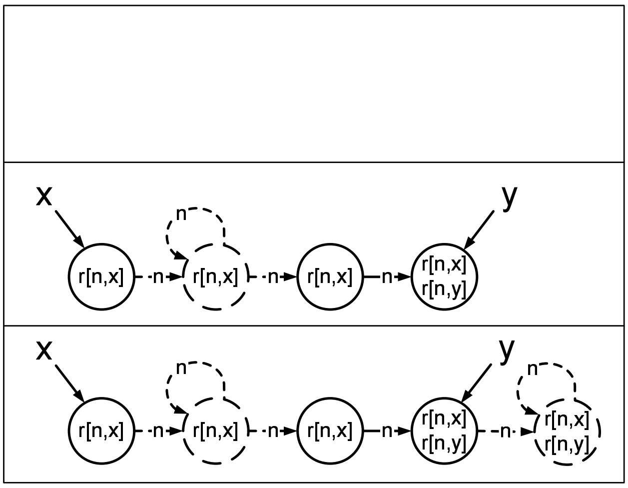


Figure 2.9: Structures after Coerce

Definition 12 (Compatibility Constraint) *A* compatibility constraint *is a term of the form $\phi_1 \rhd \phi_2$, where ϕ_1 is an arbitrary formula, and ϕ_2 is either an atomic formula or the negation of an atomic formula. A 3-valued structure S and an assignment β* satisfy $\phi_1 \rhd \phi_2$, *denoted by $S, \beta \models \phi_1 \rhd \phi_2$, if whenever β is an assignment such that $[\![\phi_1]\!]_3^S(\beta) = 1$, we also have $[\![\phi_2]\!]_3^S(\beta) = 1$. We say that S satisfies $\phi_1 \rhd \phi_2$, denoted by $S \models \phi_1 \rhd \phi_2$, if for every β we have $S, \beta \models \phi_1 \rhd \phi_2$.*

The algorithm for *coerce* discards the structure if ϕ_1 evaluates to 1 while ϕ_2 evaluates to 2. If ϕ_2 evaluates to 1/2 it changes the interpretation of the predicate in such a way that makes ϕ_2 evaluate to 1. When ϕ_2 is an equality it adjusts the *sm*-predicate.

There are two sources of *compatibility constraints*:

1. The defining formulae of instrumentation predicates, and
2. additional formulae that formalize the properties of stores that are compatible with the semantics of C. For instance, the fact that pointer variables can point to only one heap cell.

If we apply *coerce* to the structures arising after applying the update-formulae (Figure 2.8) we arrive at the two structures depicted in Figure 2.9. The top-most structure was eliminated because the summary node on the right is definitely not reachable from x or y. The two other structures were sharpened.

Collecting Semantics in the 3-valued Domain. We are now ready to define an abstract semantics, which includes *focus* and *coerce*. *Focus* and *coerce* are used in the way described above.

$$
\mathit{StructSet}[v] = \\
\begin{cases}
\mathit{In} & \text{if } v = bg(G) \\
\left.\begin{array}{l}
\displaystyle\bigcup_{w \to v \in E(G),\ w \in As(G)} \widehat{t\_embed}_c(\widehat{coerce}([\![\widehat{st(w)}]\!]_3^S(\widehat{focus_{F(w)}}(\mathit{StructSet}[w])))) \\
\cup \displaystyle\bigcup_{w \to v \in E(G),\ w \in Id(G)} \{S \mid S \in \mathit{StructSet}[w]\} \\
\cup \displaystyle\bigcup_{w \to v \in Tb(G)} \left\{ t\_embed_c(S) \;\middle|\; \begin{array}{l} S \in \widehat{coerce}(\widehat{focus_{F(w)}}(\mathit{StructSet}[w])) \\ \text{and } S \models_3 cond(w) \end{array} \right\} \\
\cup \displaystyle\bigcup_{w \to v \in Fb(G)} \left\{ t\_embed_c(S) \;\middle|\; \begin{array}{l} S \in \widehat{coerce}(\widehat{focus_{F(w)}}(\mathit{StructSet}[w])) \\ \text{and } S \models_3 \neg cond(w) \end{array} \right\}
\end{array}\right\} & \text{otherwise.}
\end{cases}
$$

2.2 TVLA - Three-Valued-Logic Analyzer

TVLA implements the shape analysis framework from [SRW02] described above. It was developed by Tal Lev-Ami at Tel-Aviv University for his Master's thesis [LA00, LAS00]. Since then it has been consistently extended. This includes new abstraction mechanisms, an improved *focus* operation that can be applied to arbitrary formulae, an enhanced version of *Coerce*, automatic generation of update formulae for instrumentation predicates [RSL03] by finite differencing, and the possibility of analyzing concurrent systems.

The TVLA distributions contain sample analyses dealing with singly- and doubly-linked lists, sorting programs, etc. Since the semantics of statements can be separated from the concrete programs that are analyzed, it is possible to reuse them in new analyses. This also includes instrumentation predicates, because they are only concerned with the data structures analyzed and not the specific algorithms.

3 Sets as Data Abstractions

In this chapter we would like to formally define the Abstract Data Type (ADT) Set. As a motivation, it is interesting to examine mathematical sets, because they share some key properties with the ADT we want to define.

3.1 Mathematical Sets

Ordinarily, one thinks of sets as collections of objects. The objects of a set are called members or elements. Elements of sets can be anything, letters of the alphabet, numbers, people, or sets themselves. There are different ways of describing sets:

- By listing its elements:
  $A = \{9, 16, 25\}$, or $B = \{\{9\}, \{3, 7, 8\}\}$
- By specifying a property of its elements:
  $C = \{x \in \mathbb{N} \mid 3 \leq x \leq 5\}$, or $D = \{y^2 \mid y \in C\}$, or $E = \{x \in \mathbb{Z} \mid x \text{ is odd}\}$

The same set can be expressed in many ways. If the sets A and D are equal, we write $A = D$. The order of elements in the description or the repetition of elements have no effect: $\{4, 3, 5\} = \{5, 4, 3\} = \{3, 3, 3, 4, 4, 5\} = C$. Two sets are considered equal if they have the same members. This is known as extensionality.

Set membership is symbolized by $\in$.

- $9 \in A$, but not $9 \in B$, written $9 \notin B$,
- $\{9\} \in B$ and $blue \notin D$.

Sets can also contain no elements at all. Such a set is called the *empty set*, denoted by $\varnothing$. The cardinality of $\varnothing$ is 0. In general, the cardinality of a set A is determined by the number of distinct elements of A. It is denoted by $|A|$. For example:

- $|A| = |C| = |D| = 3$,
- $|B| = 2$, not 4 as one could possibly think,
- $|E| = \aleph_0$, an example of an infinite set.

If every member of a set A is also a member of a set B, then A is said to be a *subset* of B, written $A \subseteq B$. This may not be confused with set membership:

- While $\{9\} \notin A$, we have $\{9\} \subseteq A$.
- On the other hand $\{9\} \in B$, but $\{9\} \nsubseteq A$,
- and $\{9\} \in \{9, \{9\}\}$ and $\{9\} \subseteq \{9, \{9\}\}$.

The empty set is a subset of every set S and every set S is a subset of itself:

- $\varnothing \subseteq S$
- $S \subseteq S$

One can construct new sets by combining existing sets by *union* and *intersection*. The *union* of two sets A, B, denoted by $A \cup B$, contains exactly the elements of A and B. The *intersection* of two sets A, B, $A \cap B$, consists of the elements that A and B have in common. For example:

- $A \cap B = \varnothing, A \cup B = \{9, 16, 25, \{9\}, \{3, 7, 8\}\}$,
- $E \cap A = \{9, 25\}, E \cup A = \{x \in \mathbb{Z} \mid x = 16 \text{ or } x \text{ is odd}\}$.

The view of sets presented above stems from the work of Cantor in the 19th century. It is now called *Naïve Set Theory*. The intuitive ideas behind it are still present, though. The possibility to specify sets by a property of their elements led to contradictions. In 1901, Bertrand Russell discovered what is now known as Russell's paradox: Consider the set R to be "the set of all sets that do not contain themselves". Formally:

$$R = \{E \mid E \notin E\}$$

Then, we can ask whether R is an element of itself. If $R \in R$, then by definition of R we have $R \notin R$. If we assume $R \notin R$, then $R \in R$ by definition.

There were efforts to overcome this problem and other contradictions. Today, the *Zermelo-Fraenkel axioms of set theory (ZF)* are the standard axioms of axiomatic set theory, which forms the basis of all ordinary mathematics. An alternative axiom system is the *Von Neumann-Bernays-Gödel set theory (NBG)* which is a conservative extension of ZF [Ebb94, Obe94].

Russell and Whitehead also proposed a solution in their *Principia Mathematica* introducing *Type Theory*. A hierarchy of types ensured that contradictions were prevented. Whenever set inclusions of the form $A \in B$ occurred in the definitions of sets, the type of A has to be "smaller" than the type of B. This prohibits circular inclusion relations, especially the primitive case $E \in E$. The approach was not considered flexible enough for set theory in that it constrained the definition of sets too strongly. Type theory found practical applications in programming languages, however. For the ADT Set Russell and Whitehead's system seems appropriate though. When using sets as data abstractions it is sensible to only store elements of one particular type in a set.

3.2 Abstract Data Type Set

We now go on to define what we consider to be the Abstract Data Type (ADT) Set. It will serve as a reference for the implementations introduced later. The definition should be independent of possible implementations. Notice that a concrete implementation would also constitute a formal specification. It would however contain many design decisions that are not specific to the data type itself.

A method widely used for the specification of data types is known as *Algebraic Specification of Data Types* [EM85, EM90, LEW97]. Here, a specification consists of a signature and axioms. The signature introduces operations on the data type, while the axioms capture the meaning of the given operations. Data Types defined in this way are often called Abstract Data Types. This is for three reasons:

- The specification is concerned with the data type itself as an abstract mathematical object and not with its implementation by a concrete program in a particular programming language.
- Specifications may be incomplete by only partially specifying the meaning of operations.
- They maybe defined in terms of other data types that serve as parameters. This is also called generic specification.

A typical first example of this kind of specification is the ADT Stack. Its signature contains four functions and one predicate. We use the notation of [BRS+00].

$$
\begin{array}{llcllcll}
\textbf{constants} & \mathit{EmptyStack} & : & & & & \rightarrow & \text{stack} \\
\textbf{functions} & \mathit{Push} & : & \text{stack} & \times & \text{element} & \rightarrow & \text{set} \\
 & \mathit{Pop} & : & \text{stack} & & & \rightarrow & \text{stack} \\
 & \mathit{Top} & : & \text{stack} & & & \rightarrow & \text{element} \\
\textbf{predicates} & \mathit{IsEmpty} & : & \text{stack} & & & \rightarrow & \text{Bool}
\end{array}
$$

To give meaning to these symbols axioms are provided:

$$
\begin{array}{ll}
\textbf{variables} & s \ : \ \text{stack} \\
 & a \ : \ \text{element} \\
\textbf{axioms} & IsEmpty(EmptyStack) = True, \\
 & IsEmpty(Push(s,a)) = False, \\
 & Pop(Push(s,a)) = s, \\
 & Top(Push(s,a)) = a.
\end{array}
$$

Note that this example covers all aspects of abstraction described above. It abstracts from implementation issues. Its specification is in fact incomplete: Neither *Pop(EmptyStack)* nor *Top(EmptyStack)* are constrained by the axioms. The definition depends on a parameter type *element*. The axioms make use of equality on that type in the last axiom.

While we easily grasp an intuitive meaning of these specifications, it is of course profitable to give a formalization of the concept. We will not go into detail about this since we do not rely on the precise definitions in the following chapters. The semantics of such a specification is a set of many-sorted algebras. An algebra belongs to this set if it is a model of the axioms of the specification. The axioms are implicitly universally quantified. Usually, there are many non-isomorphic models of a given specification reflecting the incompleteness of the definition. The interested reader may consult [EM85] and [LEW97] for an in-depth treatment of the topic.

We are now ready to specify the ADT Set in these terms. The full specification is displayed in Table 3.1. Our specification is parameterized by an *element* type. This could also be instantiated with a *set* itself, building sets of sets of some primitive type, and so on. We are assuming an existing specification of the natural numbers *nat*.

The empty set is provided as a constant. Other sets can be constructed by inserting and removing elements using `.insert`$(\cdot)$ and `.remove`$(\cdot)$. The `.selectAndRemove` function returns an element and removes it from the set. It can be used to iterate over a set. The `.sizeOf` function returns the cardinality of the set as a natural number. The $\in$ predicate allows to test set membership. $\subseteq$ and $=$ correspond to subset and equality of sets.

Most of the axioms are straightforward. We distinguish equality on sets $=$, equality on elements $=_{el}$, and equality on natural numbers $=_{nat}$. Axiom (1) assures that every possible set can be constructed by applications of $\varnothing$ and `.insert`. In axiom (5) we only have an implication because the `.selectAndRemove` function chooses an element nondeterministically. Axioms (6) and (7) correspond to the extensionality axiom of set theory. Axioms (8)-(13) deal with the cardinality of sets. The axioms are complete in the sense that the meaning of arbitrary formulae over the given alphabet (the functions and predicates of the ADT specification) can be derived.

set =
begin generic specification

parameter	element							
using	nat							
sorts	set							
constants	$\varnothing$	:	set					
functions	$\cdot$`.insert(`$\cdot$`)`	:	set	$\times$	element	$\rightarrow$	set	
	$\cdot$`.remove(`$\cdot$`)`	:	set	$\times$	element	$\rightarrow$	set	
	$\cdot$`.selectAndRemove`	:	set			$\rightharpoonup$	element	$\times$ set
	$\cdot$`.sizeOf`	:	set			$\rightarrow$	nat	
predicates	$\cdot \in \cdot$	:	element	$\times$	set			
	$\cdot \subseteq \cdot$	:	set	$\times$	set			
	$\cdot = \cdot$	:	set	$\times$	set			
variables	s, s'	:	set					
	a, b	:	element					

axioms

set **generated by** $\varnothing$, `.insert`; (1)

$\neg(a \in \varnothing)$, (2)

$a \in s$`.insert(`b`)` $\leftrightarrow a =_{el} b \vee a \in s$, (3)

$a \in s$`.remove(`b`)` $\leftrightarrow a \neq_{el} b \wedge a \in s$, (4)

$(a, s') = s$`.selectAndRemove` $\rightarrow a \in s \wedge a \notin s' \wedge s'$`.insert(`$a$`)` $= s$, (5)

$s \subseteq s' \leftrightarrow a \in s \rightarrow a \in s'$, (6)

$s = s' \leftrightarrow s \subseteq s' \wedge s' \subseteq s$, (7)

$\varnothing$`.sizeOf` $=_{nat} 0$, (8)

s`.insert(`b`).sizeOf` $=_{nat}$ s`.sizeOf` $\leftrightarrow b \in s$, (9)

s`.insert(`b`).sizeOf` $=_{nat}$ s`.sizeOf` $+ 1 \leftrightarrow \neg(b \in s)$, (10)

s`.remove(`b`).sizeOf` $=_{nat}$ s`.sizeOf` $\leftrightarrow \neg(b \in s)$, (11)

s`.remove(`b`).sizeOf` $=_{nat}$ s`.sizeOf` $- 1 \leftrightarrow b \in s$, (12)

$(a, s') = s$`.selectAndRemove` $\rightarrow s'$`.sizeOf` $=_{nat}$ s`.sizeOf` $- 1$. (13)

end generic specification

Table 3.1: ADT Set

4 Shape Analysis of Implementations

In this chapter we analyze two prototypical C implementations of the ADT Set. One implementation is based on singly-linked lists, the other on binary trees. After briefly introducing parts of the two implementations, we proceed to describe our analyses. The main goal of the analyses is to prove that the implementations comply with the ADT specification given in Chapter 3. The implementations each contain the two methods, `insertElement`, `removeElement` and the function `isElement`. They implement the $\cdot.\mathtt{insert}(\cdot)$, $\cdot.\mathtt{remove}(\cdot)$ functions and the $\cdot \in \cdot$ predicate, respectively. We chose to show the following two axioms, since they capture the most important aspects of the ADT Set:

$$a \in s.\mathtt{insert}(b) \leftrightarrow a =_{el} b \vee a \in s, \quad (3)$$
$$a \in s.\mathtt{remove}(b) \leftrightarrow a \neq_{el} b \wedge a \in s \quad (4)$$

Our analyses are conducted using TVLA [LAS00] and are based on previous analyses on lists and trees contained in the TVLA 2 distribution.

4.1 List-based Implementation

```
typedef struct List
{
  void* data;
  struct List* next;
} List;

typedef struct Set
{
  List* list;
  int (*compare)(void*, void*);
  int size;
} Set;
```

(a)

```
int isElement(Set* set, void* element)
{
  List* list = set->list;

  while (list != 0)
  {
    if (compare(list->data, element) == 0)
      return 1;

    list = list->next;
  }

  return 0;
}
```

(b)

Figure 4.1: C structure declarations for Lists and Sets and C source of membership test

Our first set implementation uses singly-linked lists to store the elements. It also maintains the size of the current set. The structure declarations are visible in Figure 4.1. When allocating such a set, a compare-function has to be given, that establishes an equivalence relation on the data elements.

Figure 4.1 also shows the code for testing set membership. The method simply iterates over the list, comparing each item with the element that is tested for set membership.

```
void insertElement(Set* set, void* element)
{
  List* list = set->list;
  List* prev = 0;

  while (list != 0)
  {
    if (compare(list->data, element) == 0)
        return;

    prev = list;
    list = list->next;
  }

  List* newList = (List*)malloc(sizeof(List));
  newList->data = element;
  newList->next = 0;
  set->size++;

  if (prev == 0) //list is empty
  {
    set->list = newList;
  }
  else //append item to list
  {
    prev->next = newList;
  }
}
```

(a)

```
void* removeElement(Set* set, void* element)
{
  List* temp;
  List* list = set->list;

  if (list == 0)
    return;

  if (compare(list->data, element) == 0)
  {
    set->size--;
    set->list = list->next;
    free(list);
  }
  else
    while (list->next != 0)
    {
      if (compare(list->next->data, element) == 0)
      {
        void* deletedElement = list->next->data;
        set->size--;
        temp = list->next->next;
        free(list->next);
        list->next = temp;
        return deletedElement;
      }
      list = list->next;
    }
}
```

(b)

Figure 4.2: C source of Insertion and Removal methods

Figure 4.2 shows the implementations of the insertion and removal methods. The insertion method iterates over the list until it either finds the element or reaches the final element of the list, indicated by a null-pointer in the next-field. If the element was not found it is appended at the end. Removal works similarly. When the element is found, it is decoupled from the list and the memory is freed.

4.1.1 Data Structure Invariants

Our analyses rely on a number of data structure invariants at entrance to the methods. Showing their maintenance is part of the proof. By data structure invariants we mean invariants that are related directly to the concrete data structure employed to implement the ADT Set. In this case properties of singly-linked lists:

- The list is acyclic

- The list does not contain any duplicate elements

We use instrumentation predicates to capture these properties formally using first-order logic.

4.2 Tree-based Implementation

As in the list-based case, a compare-function is needed. This time it has to implement a reflexive total order. This is necessary, to build an ordered tree. Figure 4.3 shows the structure declarations. Every node in the tree stores one of the set elements and maintains pointers to two children nodes *left* and *right*.

```
typedef struct Tree
{
  void* data;
  struct Tree* left;
  struct Tree* right;
} Tree;

typedef struct Set
{
  Tree* tree;
  int (*compare)(void*, void*);
  int size;
} Set;
```

(a)

```
int isElement(Set* set, void* element)
{
  Tree* tree = set->tree;

  while (tree != 0)
  {
    if (compare(tree->data, element) == 0)
      return 1;
    else if (compare(tree->data, element) < 0)
      tree = tree->left;
    else
      tree = tree->right;
  }

  return 0;
}
```

(b)

Figure 4.3: C structure declarations for Trees and Sets and C source of isElement test

Figure 4.3 also contains the source of the set membership test. The method simply traverses the tree until it either finds the element or reaches a leaf node. The source of the insertion and removal methods on trees can be found in the appendix, since it is too large to be dealt with here. We restrict ourselves to mentioning the main ideas of the two algorithms. New elements are always inserted as new leaf nodes, by traversing the tree to the correct position. While insertion of elements if fairly easy and quite similar to its list pendant, removal of elements is a non-trivial task. Figure 4.4 illustrates this. Removing elements that are stored in leaf nodes is simple (left). They can simply be decoupled from their respective parent nodes. If the node has one child, we can connect this child at the place of the node to its former parent node (middle). The most complicated case arises when the particular node has two child nodes (right). In this case, we have to find another node in the tree to replace the element node. This node has to be smaller than all nodes on the right and greater than all nodes on the left. There are two ways to find such an element. Either one can take the right-most element of the left subtree or the left-most

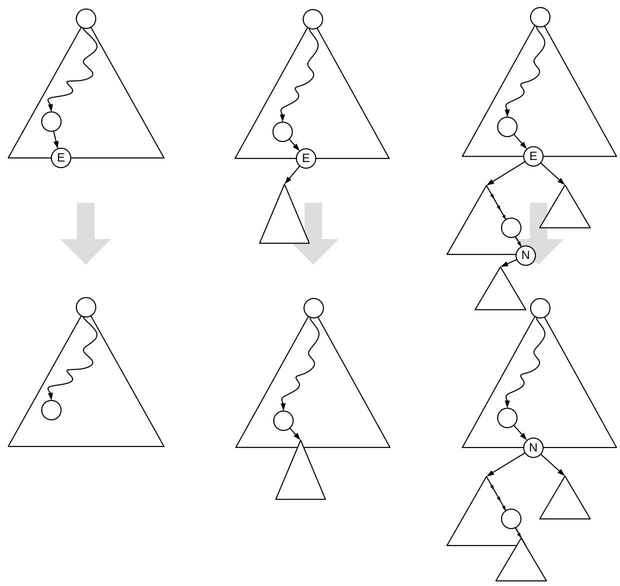


Figure 4.4: Removal from Ordered Tree

element of the right subtree. We chose to always take the right-most element of the left subtree. In addition, there are some special cases of the latter case. For instance, if the root of the left subtree is already the right-most element of the left subtree.

4.2.1 Data Structure Invariants

In order to prove our ADT Set axioms we need to maintain two data structure invariants:

- The structure representing the set is a tree

  Out of many equivalent definitions for "binary treeness", we chose the following: Whenever an element is reachable from the left child of a node in the structure, then it is not reachable from the right child, and vice versa.

- The tree is ordered

  Every element reachable from the left child is smaller and every element reachable from the right child is greater. This implies that the tree does not contain duplicate elements. It also implies the first data structure invariant. It is still useful to consider the first invariant, because it may help in proving this one.

Again, we used instrumentation predicates to formalize the two invariants using first-order logic. Proving the latter proved to be quite difficult. It is a global property, i.e. it does relate elements in the tree that are not directly connected. We will go into more detail about this in the analysis section.

4.3 Shape Analysis

To prove the ADT Set axioms we perform three analyses for each implementation. The analyses of the insertion methods prove the following:

$$isElement(a, s\texttt{.insertElement}(b)) \leftrightarrow a =_{el} b \vee isElement(a, s)$$

Notice the difference compared with the corresponding axiom (3). The instrumentation predicate *isElement* replaces the $\cdot \in \cdot$ predicate. That is we prove the property of the insertion method in terms of an instrumentation predicate. The same holds for the removal methods and axiom (4). There, we prove:

$$isElement(a, s\texttt{.removeElement}(b)) \leftrightarrow a \neq_{el} b \wedge isElement(a, s)$$

To conclude the proofs we show that the `isElement` functions in both implementations are equivalent to the instrumentation predicate *isElement*:

$$isElement(a, s) \leftrightarrow s.\texttt{isElement}(a)$$

Combining this equivalence with the two preceding proofs yields:

$$s\texttt{.insertElement}(b)\texttt{.isElement}(a) \leftrightarrow a =_{el} b \vee s\texttt{.isElement}(a)$$
$$s\texttt{.removeElement}(b)\texttt{.isElement}(a) \leftrightarrow a \neq_{el} b \wedge s\texttt{.isElement}(a)$$

These two equivalences correspond directly to axioms (3) and (4).

4.3.1 Shape Analysis of List-based Implementation

Our analysis is based on existing analyses on lists and trees. We borrowed the concrete semantics of most of the statements from these. The following table shows how we represent the state by logical predicates.

Predicate	**Intended Meaning**
$x(v)$ for each $x \in Var$	Pointer variable x points to heap cell v.
$n(v_1, v_2)$	The *next* selector of v_1 points to v_2.
$deq(v_1, v_2)$	The *data*-fields of v_1 and v_2 are equal.
$isSet(v)$	v represents a set.
$or[n, x](v)$ for each $x \in Var$	v was reachable from x via *next*-fields.

As depicted, pointer variables are represented by unary predicates. The *next*-field is modeled by a binary predicate. Since we can only model the structure of the heap by these predicates, primitive values have to be dealt with differently. Abstracting from the concrete values of the *data*-fields, we capture the equivalence relation between *data*-fields by the binary predicate *deq*. This corresponds to the compare-function needed in the implementation. To differentiate between set locations and other locations in the heap, the *isSet* predicate is used. To be able to relate elements contained in the list before the execution of one of our procedures with their output structures, we mark elements reachable from x via *next*-fields using the $or[n,x]$ predicate.

While the above core predicates suffice to define the concrete semantics of all the statements, we need additional instrumentation predicates to gain precision.

Predicate	**Defining Formula**	**Intended Meaning**
$is[n](v)$ $c[n](v)$ $t[n](v_1, v_2)$ $r[n,x](v)$ for each $x \in Var$	$\exists v_1, v_2.(v_1 \neq v_2 \land n(v_1, v) \land n(v_2, v))$ $\exists v_1.(n(v_1, v) \land n^*(v_1, v_2))$ $n^*(v_1, v_2)$ $\exists v_1.(x(v_1) \land t[n][v_1, v))$	v is shared. v resides on a cycle. Transitive reflexive closure of *next*. v is reachable from x via *next*-fields.
$noeq[deq, n](v)$	$\forall v_1.(((t[n](v_1, v) \lor t[n](v, v_1)) \land v_1 \neq v) \rightarrow (\neg deq(v_1, v) \land \neg deq(v, v_1)))$	The *data*-field of v is different from the *data*-fields of locations that can reach v and that are reachable from v.
$validSet(v)$ $isElement(v_1, v_2)$	$isSet(v) \land noeq[deq, n](v)$ $isSet(v_2) \land \exists v.(t[n](v_2, v) \land deq(v_1, v) \land v \neq v_2)$	v represents a valid set (no duplicate entries). v_1 is an element of set v_2.

The first four of these instrumentation predicates capture general properties of the shape of the heap. They have been used in previous analyses of list-manipulating programs. $c[n]$ covers the acyclicity data structure invariant mentioned in the implementation section.

The $noeq[deq, n]$ predicate is tailored specifically to the current task. It expresses that no two elements in the list have equal *data*-fields. The definition comprises both directions, i.e. both elements reachable from v and elements from which v is reachable. This actually makes it easier to reestablish the property when manipulating the list. It is a formalization of the second data structure invariant for lists. *validSet* does not help to increase precision. It only increases the readability of the output structures.

To capture our notion of set membership we define the *isElement*-predicate. v_1 is an element of set v_2 if its *data*-field is equal to one of the nodes reachable from v_2. Our analysis shows that the effect of the insertion and removal methods on set membership, expressed by *isElement* conforms to the ADT Set axioms.

Our input structures cover all possible lists representing sets pointed to by *set*. *element* points to the element that shall be inserted into the set. Figure 4.5 displays these structures. In (a) *set* is empty. In (b) *set* is non-empty and set membership of *element* is unknown, *isElement*'s value is indefinite for the nodes pointed to by *element* and *set*.

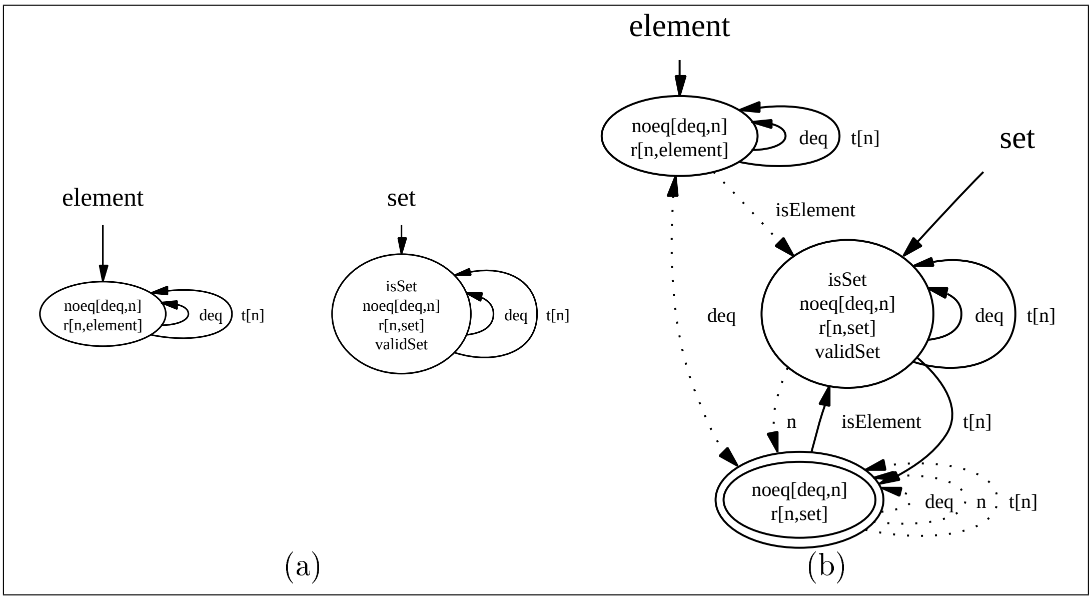


Figure 4.5: Input Structures for List-based Insertion and Removal

Insertion

Running the analysis for insertion yields three output structures that are shown in Figure 4.6. All of the resulting structures fulfill the data structure invariants, i.e. $noeq[deq, n]$ is true for the set and $c[n]$ is false everywhere. Also, *isElement* is true for the nodes pointed to by *element* and *set*. In addition, the $or[n, set]$-predicate indicates that elements which were formerly reachable from *set* are still reachable after the execution of *setInsert*.

Looking at the structures one can identify the different cases that the insertion method has to deal with. Structure (a) corresponds to the empty set as input structure. In structure (b) a new element had to be appended to the list, because the *data*-field of *element* is not equal to any of the original elements of the list (the *deq* predicate is false). In structure (c) *element* was already contained in the list, indicated by the *isElement*-predicate.

Removal

When translating the C code into a Control Flow Graph in TVLA, we omitted the deallocation of the element in the list. This is only for illustration purposes.

Running *setRemove* results in four output structures displayed in Figure 4.7. Again, the maintenance of the data structure invariants is proven: $noeq[deq, n]$ is true and $c[n]$ is false

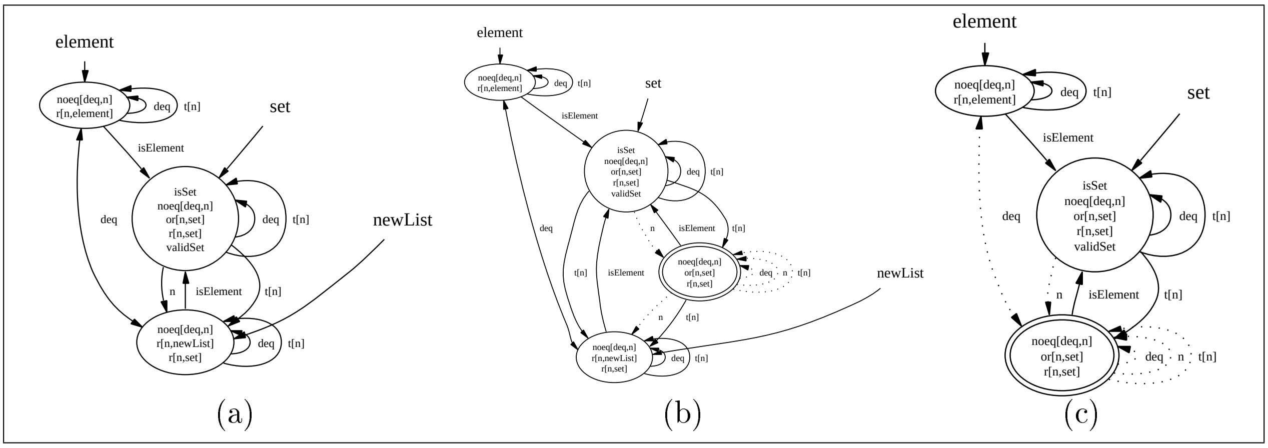


Figure 4.6: Output Structures for List-based Insertion

everywhere. The element has indeed been removed from the list. This can be observed by the $isElement$-predicate. Other elements of the set are still contained, as indicated by the $or[n, set]$-predicate.

Structures (a) and (c) correspond to the case where *element* was not contained in the set before. The two other structures (a) and (d) reflect the case where *element* was indeed part of the set. The abstraction also distinguishes between empty (c and d) and non-empty sets (a and b).

Membership Test

We omit to display the output structures of this analysis, since the routine is not manipulating the heap at all. The analysis checked that our *isElement* function returns true if and only if the $isElement$-predicate holds. This is done by separating the structures into those that reach a point where true is returned and those structures that reach a point where false is returned. By this, we establish a connection between the different analyses. The two other analyses on list insertion and removal only proved correctness in terms of the $isElement$-predicate. The current analysis shows that this was just.

4.3.2 Shape Analysis of Tree-based Implementation

The domain is represented in a similar way as in the list-based case. Instead of having a *next*-predicate, $left$- and *right*-predicates are used to model the $left$- and *right*-fields in the tree. The $left$-predicate is also used to model the *tree*-field in the set structure to minimize the number of predicates. The *tree*-field only occurs at most once in all of the structures.

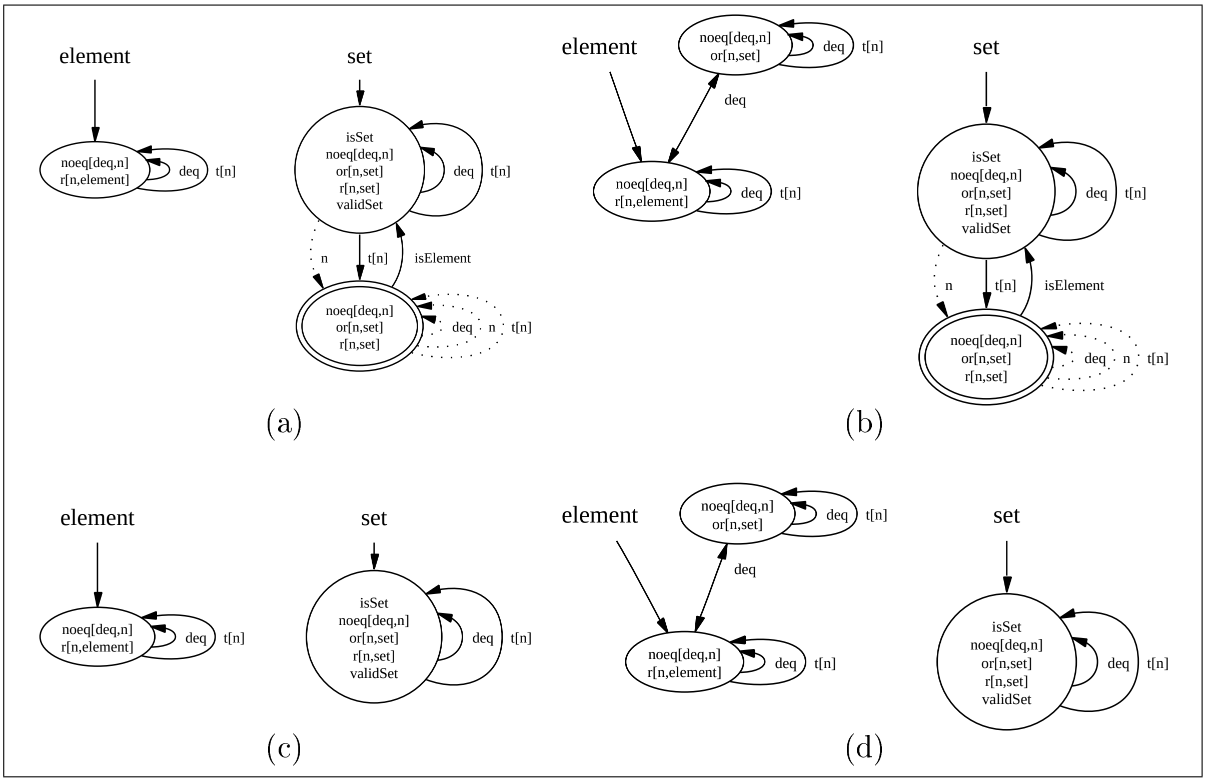


Figure 4.7: Output Structures for List-based Removal

Predicate	Intended Meaning
$x(v)$ for each $x \in Var$ $sel(v_1, v_2)$ for each $sel \in \{left, right\}$ $dle(v_1, v_2)$	Pointer variable x points to heap cell v. The $left$ ($right$) selector of v_1 points to v_2. v_1->$data \leq v_2$->$data$.
$or[x](v)$ for each $x \in Var$ $isSet(v)$	v was reachable from x via $left$- and $right$-fields. v represents a set.

As noted in the implementation section, an ordering relation is needed here. It is modeled by the dle-predicate, which is assumed to be reflexive and transitive during the analysis. $or[x]$ and $isSet$ have the same meaning as before.

While the core predicates used to model the domain were very similar to the list-based case, the choice of instrumentation predicates was quite different. We separate them into two parts. One is solely concerned with the structure of the trees. The other also deals with ordering.

Predicate	Defining Formula	Intended Meaning
$down(v_1, v_2)$	$left(v_1, v_2) \vee right(v_1, v_2)$	The union of the two selector predicates $left$ and $right$.
$downStar(v_1, v_2)$	$down^*(v_1, v_2)$	Records reachability between tree nodes.
$downStar[sel](v_1, v_2)$ for each $sel \in \{left, right\}$	$\exists v.(sel(v_1, v) \wedge down^*(v, v_2))$	Remembers the first selector needed to reach v_2 from v_1.
$r[x](v)$ for each $x \in Var$	$\exists v_1.(x(v_1) \wedge downStar(v_1, v))$	v is transitively reachable from x.
$treeNess$	$\forall v_1, v_2, v.((downStar[left](v, v_1) \wedge downStar[right](v, v_2)) \Rightarrow (\neg downStar(v_1, v_2) \wedge \neg downStar(v_2, v_1)))$	The heap consists of trees.

The two $downStar[sel]$-predicates record reachability between tree-nodes, where the first selector on the path is sel. In ordered trees this determines the relation between the elements in the tree. To be able to check whether the ordering is maintained, it is important to keep this relation precise for elements that are manipulated. $treeNess$ records the first data structure invariant mentioned in the implementation section. We decided to make $treeNess$ a global nullary predicate to reduce the size of the domain. There is a drawback to this approach however. It is nearly impossible to reestablish the property once it is violated, because we lose information about parts of the heap that still satisfy the property. A unary $treeNess$ predicate would be able to capture local violations and make it easier to reestablish the property after it was temporarily destroyed. The methods that we checked maintain $treeNess$ in the entire heap permanently allowing to use the nullary predicate.

Predicate	**Defining Formula**	**Intended Meaning**
$dle[x, left](v)$ for each $x \in Var$	$\exists v_1.(x(v_1) \wedge dle(v, v_1) \wedge \neg dle(v_1, v))$	The *data*-field of v is less than the *data*-field of v_1, where v_1 is pointed to by x.
$dle[x, right](v)$ for each $x \in Var$	$\exists v_1.(x(v_1) \wedge \neg dle(v, v_1) \wedge dle(v_1, v))$	The *data*-field of v is greater than the *data*-field of v_1, where v_1 is pointed to by x.
$inOrder[dle]$	$\forall v_2, v_4.(downStar[left](v_2, v_4) \Rightarrow (dle(v_4, v_2) \wedge \neg dle(v_2, v_4))) \wedge \forall v_2, v_4.(downStar[right](v_2, v_4) \Rightarrow (\neg dle(v_4, v_2) \wedge dle(v_2, v_4)))$	All the trees in the heap are in order.
$isElement(v_1, v_2)$	$isSet(v_2) \wedge \exists v_{equal}.(downStar(v_2, v_{equal}) \wedge dle(v_{equal}, v_1) \wedge dle(v_1, v_{equal}) \wedge v_{equal} \neq v_2$	v_1 is an element of set v_2.

The $dle[x, sel]$ captures the relation between the node pointed to by x and other heap nodes. These predicates are used to partition the heap into elements less than the node pointed to by x and those that are greater. Being unary predicates they can be used as abstraction predicates. This could be called a “pseudo-binary abstraction”, since parts of the binary predicate *dle* are taken to form several unary predicates.

$inOrder[dle]$ formalizes the second data structure invariant for ordered trees. It requires elements in the left subtree of a node to be smaller and elements in the right subtree to be greater than the node itself. Smaller and greater are expressed in terms of *dle*.

The set membership property *isElement* is formalized similarly to the list-based case. v_1 is an element of set v_2 if its *data*-field is equal to one of the nodes reachable from v_2, where equal can be formulated in terms of *dle*.

Figure 4.8 displays the input structures for our analysis of the insertion and removal methods. In the following we omitted several predicates to make the visualizations more readable. The predicates that we left our were $left, right, down, downStar$. Again, we want to cover all possible sets by these abstract structures. In structure (a) *set* is empty and thus *element* is not an element of *set*. Structure (b) represents non-empty sets. *element* might be part of the set, indicated by the dotted *isElement*-predicate and the dotted *dle*-predicate between *element* and the contents of *set*. We also had to assign a value to the *dle*-predicate for *set* which does not have a *data*-field. Its *data*-field is assumed to be greater than all other *data*-fields. Elements that were originally reachable from *set* are marked with *or*[*set*] as in the list-based case.

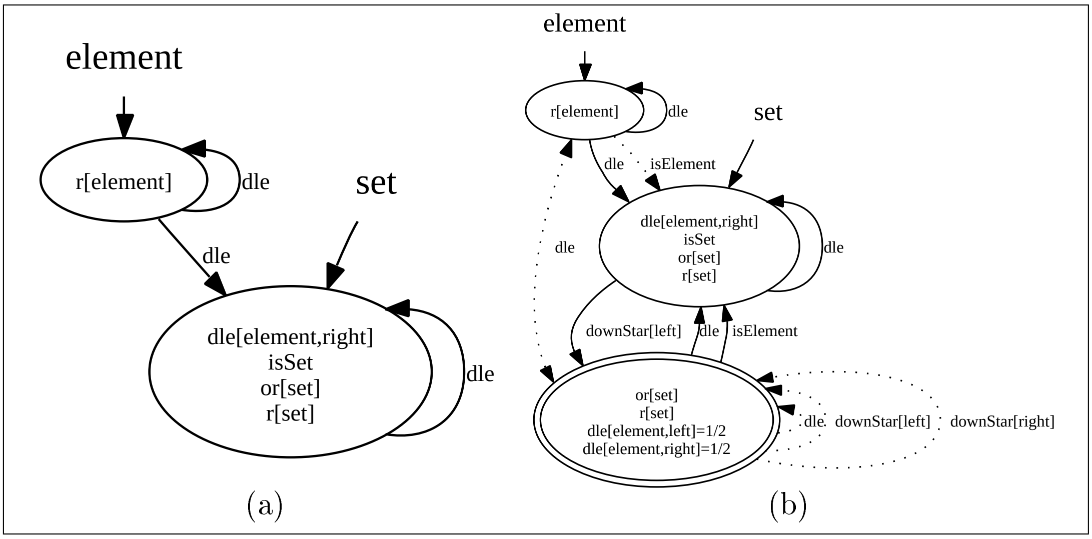


Figure 4.8: Input Structures for Tree-based Insertion and Removal

Insertion

Running the analysis for set insertion yields 21 structures at exit. Most of them concern special cases where the element had to be inserted in the left- or right-most position of the tree or where the left or the right subtree of the root was empty. All resulting structures fulfilled the data structure invariants and *element* had been inserted into *set*. We picked two structures that represent the most general cases. They can be seen in Figure 4.9.

Due to the number of binary predicates involved in the analysis the output structures are hard to read. Also, the visualization engine does not know our intuition behind the different predicates, which could help to generate more readable output. In structure (a) the algorithm found a node in the tree that is equal to *element*. The three summary nodes make up the rest of the tree. The summary node to the right represents the subtree of the node that was found. The other two summary nodes partition the parents and neighbors into those that have a smaller *data*-field and those that have a greater *data*-field. For this particular case the partitioning of the set is not important. For structure (b) however it is the key to proving that the ordering is preserved. Here, no node in the tree was found that was equal to *element*. Therefore a new heap node was allocated and inserted into the tree, preserving the ordering. This is were the partition into smaller and larger elements becomes important. Nodes that are greater than the new node can only reach it via a path that starts by going left: $downStar[left]$ is indefinite and $downStar[right]$ is false. Nodes with a smaller *data*-field can in turn only reach it via a path that starts with a *right*-edge ($downStar[right] = 1/2$ and $downStar[left] = 0$).

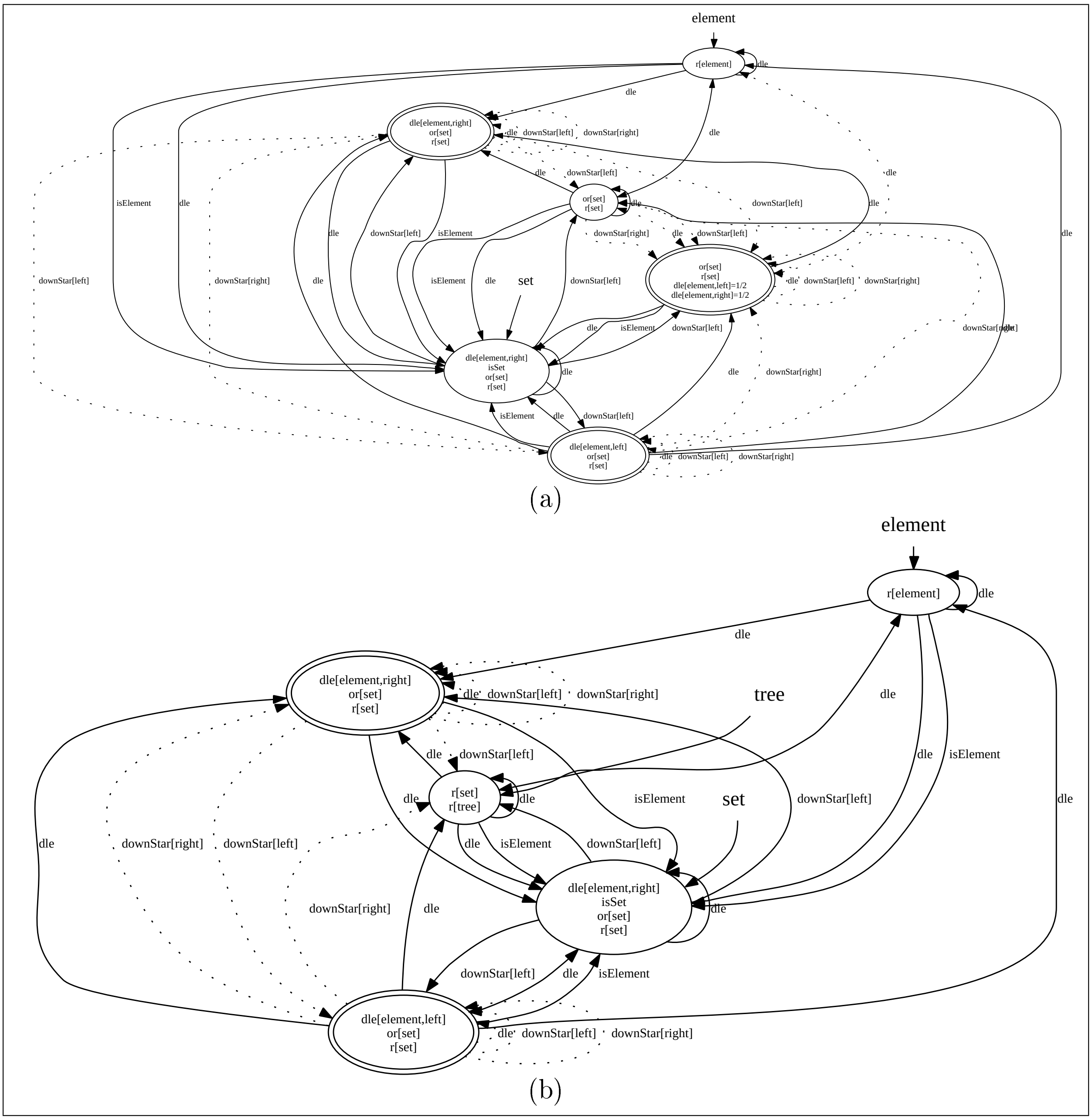


Figure 4.9: Sample Output Structures for Tree-based Insertion

Removal

As noticed in the implementation section, tree-based removal was the most complicated routine that we analyzed. Its size and complexity led to very time-consuming analyses that did not allow a trial and error approach when choosing the abstraction predicates. We used the same predicates as in the analysis of the insertion algorithm. They were developed for this method though and proved to work for the simpler insertion routine, too.

Proving that *element* is not a member of *set* after the analysis was simple, once the data structure invariants could be established. The ordering property ensures that every element only occurs once in the tree. Showing that the ordering data structure invariant was maintained was more difficult. The key predicates involved in proving this were $dle[x, sel]$ and $downStar[sel]$. The use of these predicates in the insertion routine already hints at why they are useful for removal. Figure 4.4 illustrates the different possibilities when removing an element from the tree. As the algorithm keeps track of the relevant nodes (those represented by circles in the figure) in the graph through pointer variables, $dle[x, sel]$ delivers the necessary partition to keep relevant ordering information. In addition $downStar[sel]$ captures the important first selectors on paths between these parts of the tree.

To cope with the long analysis times we decomposed the problem into smaller ones first:

- Finding the element to delete.
- The element has one or no children.
- The element has two children, the most difficult case.

In the end we put everything together.

Again, we decided to present only two representative output structures out of overall eight. They are shown in Figure 4.10. Both structures satisfy the two data structure invariants modeled by $inOrder[dle]$ and $treeNess$. In structure (a) *element* was contained in *set* and therefore removed from it. For demonstration purposes we did not free the element taken from the tree. One can see that the tree has been partitioned into nodes with a greater *data*-field and nodes with a smaller *data*-field than *element*. The same holds for structure (b). In this case *element* was not contained in *set* at the invocation of the routine. No node was removed from the tree.

Membership Test

Again, we omit to display the output structures. It is quite obvious that the analysis succeeds, because the tree traversal analyzed is part of the insertion and removal methods as well, which were analyzed before.

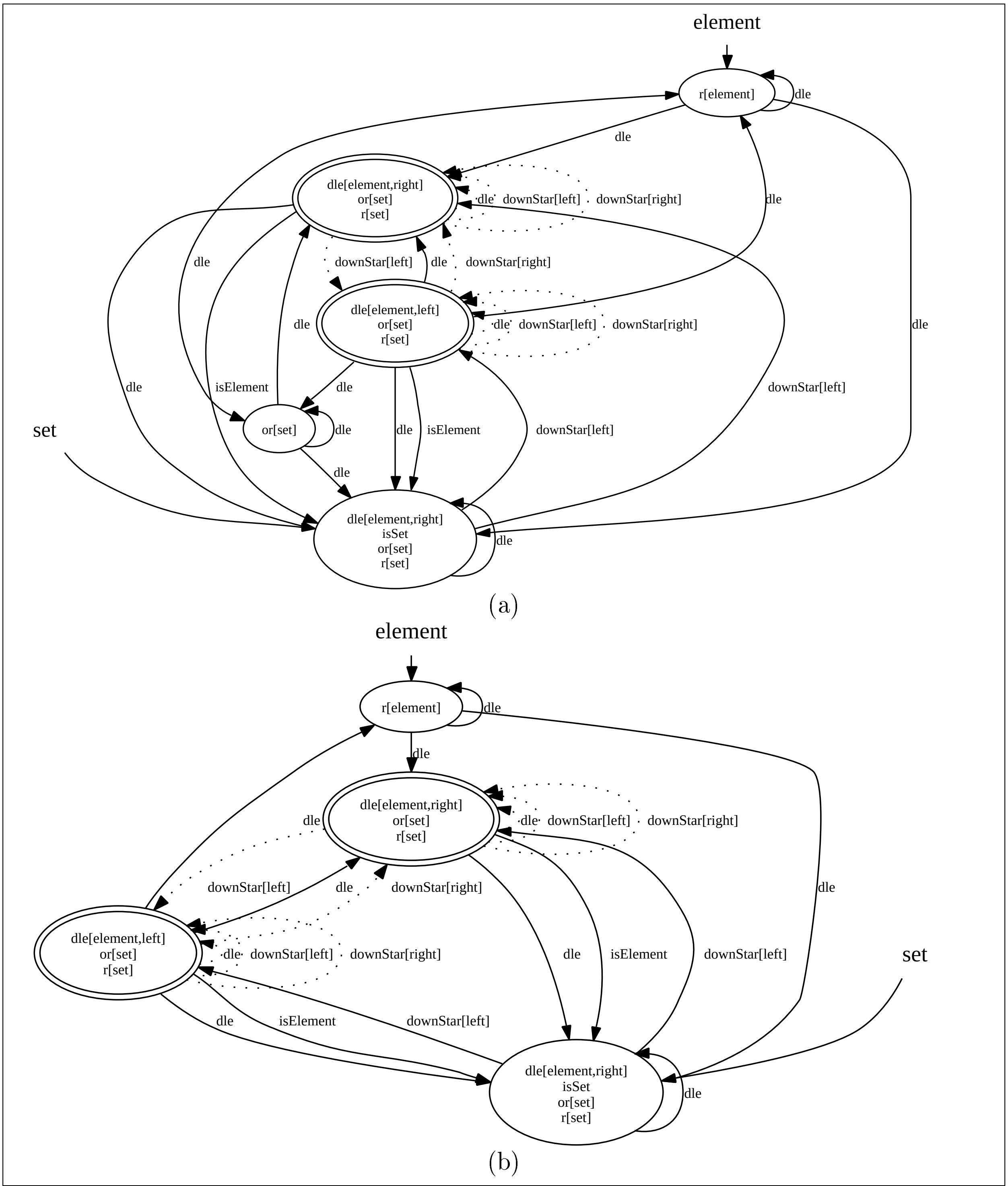


Figure 4.10: Sample Output Structures for Tree-based Removal

Analysis	#locations in CFG	#unary predicates	#binary predicates	#structures	average #structs per location	maximal #structs per location	time
Membership, List-based	9	20	5	28	3	6	2.570s
Insertion, List-based	19	29	5	81	4	11	2.720s
Removal, List-based	22	29	5	124	5	11	4.050s
Membership, Tree-based	10	18	11	84	8	19	32.84s
Insertion, Tree-based	25	24	11	536	21	91	69.23s
Removal, Tree-based	76	42	11	27697	364	3132	21767s

Table 4.1: Empirical Results

4.3.3 Empirical Results

Table 4.1 presents some data about the four analyses. The analysis of the insertion, removal and membership test methods of our list-based implementation resulted in a similar number of structures and relatively short analysis times. In the tree-based case, however, the difference was considerable. This can probably be explained with the higher number of unary predicates in the removal analysis, which led to more structures per location. The worst-case complexity of the analysis is doubly-exponential in the number of abstraction predicates. Additionally, the control flow graph (see Figure 4.11) for removal contains more than three times as many locations as the CFG for insertion.

4.3.4 Discussion

We managed to show interesting properties of list- and tree-based set implementations. Our analyses assumes data structure invariants specific to the respective implementation to hold at the entrance. The maintenance of these invariants throughout the execution of the routines is established. Using these invariants our analysis was able to prove that the effect of the insertion and removal methods complies with axioms of the ADT Set. The nature of the shape analysis framework limited our proofs to partial correctness.

We used the *isElement*-predicate to relate different analyses. While the insertion and removal methods were proved correct in terms of *isElement*, the analysis of the set mem-

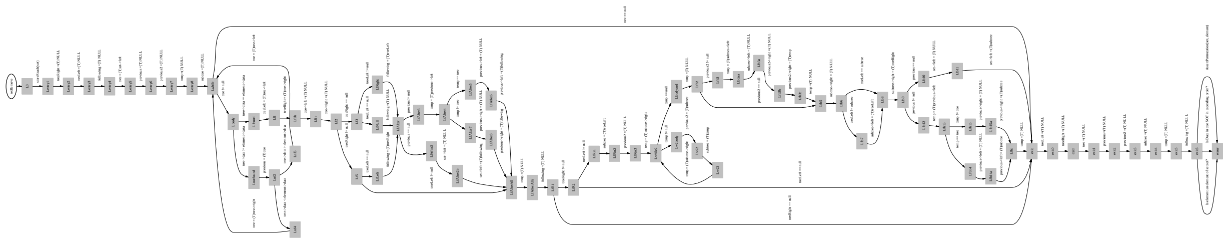


Figure 4.11: CFG for Tree Removal

bership routine showed the equivalence of this routine with *isElement*. This approach loosely corresponds to the abstraction mechanism used in [LKR04]. They use sets to abstract from more complex data structures, which limits them to statically allocated data structures. Our use of *isElement* on the other hand allows to handle dynamically allocated sets.

Choosing the right instrumentation predicates required a thorough understanding of the data structures involved. For trees this meant identifying that reachability alone is not very interesting, but that the first edge on a path from one node to another is important. However, the predicates are not tailored to specific algorithms, but to the underlying data structures. They might prove useful for other algorithms on trees and lists as well.

4.3.5 Abstraction Expressions

The need to partition the trees into smaller and larger elements led to the introduction of the $dle[x, sel]$-predicate family. The effect of these unary predicates on the abstraction could also be achieved by using the binary dle-predicate in the abstraction process. Here, individuals should only be joined if they have the same canonical name and if they agree on binary abstraction predicates to other canonical names. This is illustrated in Figure 4.12. The tree on the left is supposed to be in order. The ordering predicate is not visualized to make it more readable. Canonical Abstraction would collapse all the nodes not pointed to by x (a). The relation between the resulting summary node and the node pointed to by x would be indefinite. Additionally abstracting from dle would instead create two summary nodes and keep ordering information definite. Of course, the proposed abstraction can also be achieved using a number of unary abstraction predicates. The number of predicates needed for this is linear in the number of abstraction predicates though, to cover all canonical names.

We propose to specify the abstraction through *Abstraction Expressions*:

Definition 13 (Syntax of Abstraction Expressions) *The set of* Abstraction Expressions *over a set of unary predicates U and a set of binary predicates B is defined inductively*

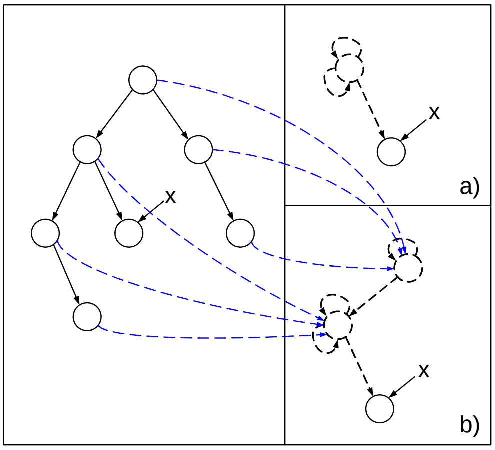


Figure 4.12: Abstraction Expressions

as follows:

- $\{u_1, \ldots, u_n\}$ *is an abstraction expression if* $\{u_1, \ldots, u_n\} \subseteq U$*,*
- $AE_1 \wedge AE_2$ *is an abstraction expression if* AE_1 *and* AE_2 *are abstraction expressions,*
- $AE \triangleright \{b_1, \ldots, b_n\}$ *is an abstraction expression if* AE *is an abstraction expression and* $\{b_1, \ldots, b_n\} \subseteq B$*.*

We define the semantics of *Abstraction Expressions* by giving an associated equivalence relation. The equivalence relation determines which nodes are to be merged.

Definition 14 (Semantics of Abstraction Expressions) *The associated equivalence relation* $\sim_{AE}$ *to an* Abstraction Expression AE *is defined inductively as follows:*

- $x \sim_{\{u_1,\ldots,u_n\}} y :\Leftrightarrow \bigwedge_{u \in \{u_1,\ldots,u_n\}} u(x) = u(y),$
- $x \sim_{AE_1 \wedge AE_2} y :\Leftrightarrow x \sim_{AE_1} y \wedge x \sim_{AE_2} y,$
- $x \sim_{AE \triangleright \{b_1,\ldots,b_n\}} y :\Leftrightarrow x \sim_{AE} y \wedge \bigwedge_{b \in \{b_1,\ldots,b_n\}} \forall z.(\bigsqcup_{\{w \mid w \sim_{AE} z\}} b(x,w) = \bigsqcup_{\{w \mid w \sim_{AE} z\}} b(y,w)).$

The *Abstraction Expression* $\{u_1, \ldots, u_n\}$ is equivalent to *Canonical Abstraction* over $\{u_1, \ldots, u_n\}$. The abstraction depicted in case (b) of Figure 4.12 can be specified using the *Abstraction Expression* $\{x\} \triangleright \{dle\}$. It will be interesting to see whether there are more applications, where abstraction can be specified more easily using such expressions than by plain *Canonical Abstraction.*

4.3.6 Future Work

Future work might try to deal with recursive implementations, following the approach presented in [RS01]. Another challenge are balanced search trees such as red-black trees or AVL trees, which have more complicated data structure invariants. The ADT Set specified also contained axioms dealing with the size of the sets. Analyzing these properties seems quite difficult using the current shape analysis framework. Integer values can be represented by list structures (by a zero node and successor nodes in the sense of the Peano axioms). Computation on them not very efficient though.

Splitting the current analysis into two phases could increase efficiency. The first phase could be solely devoted to proving the maintenance of the data structure invariants. The second could then rely on them and concentrate on the property originally desired to show.

Dead Predicates

To speed up the analyses we included additional actions in the control flow graphs of the tree-based programs. These actions nullified certain variables and allowed the engine to collapse structures that were otherwise isomorphic. This was only done for unary predicates representing dead variables, i.e. predicates that further steps of the analysis did not rely on. These predicates could be called dead predicates. A similar effect could have been achieved by marking these predicates as non-abstraction predicates locally. This approach was previously described in Roman Manevich's Master Thesis [Man03]. These dead predicates could be determined by a preceding static analysis. At the time the analyses were conducted it had not been integrated into TVLA yet. We believe that it may dramatically increase the performance of analyses in larger programs that contain many loosely coupled sections. Unfortunately, we cannot give experimental results about the magnitude of the effect. Our analysis for the tree-based removal method did not terminate within days without this optimization. Of course, the optimization could also decrease precision, because more structures are collapsed, possibly losing relevant information. However, in such a case it seems that the wrong abstraction is used, but the analysis succeeds by coincidence.

5 RESET - An imperative language with sets as primitives

In this section, we introduce RESET, an imperative language with sets as primitives. We give two semantics for RESET. Semantics I provides an idealized view of an implementation of the ADT Set defined in Chapter 3. Semantics II comes a little closer to the set implementations seen in Chapter 4. The difference between the semantics lies in the way sets are represented. In this respect, Semantics II is somewhere in between Semantics I and the implementations. We will compare the two semantics and show how they can be related formally.

5.1 Syntax

The syntax of RESET is given in BNF[1] in Figure 5.2. It contains the common control structures such as a conditional statement and a while loop. In addition to the typical constructs of an imperative language, we include expressions to test for set membership and set inclusion as well as two statements to allow for insertion and deletion of elements. The `.selectAndRemove` statement nondeterministically selects one of the elements of the set and assigns it to the given variable. This allows to perform some action on every element of a set. Pointer expressions are limited to x and $x.sel$ in order to simplify the specification of the semantics. Of course, it is still possible to access elements deeper in the heap by using temporary pointer variables.

Figure 5.3 shows a simple example RESET program. It computes the intersection of the two input sets X and Y. The references to the contents of X are first copied into

[1]Backus-Naur form

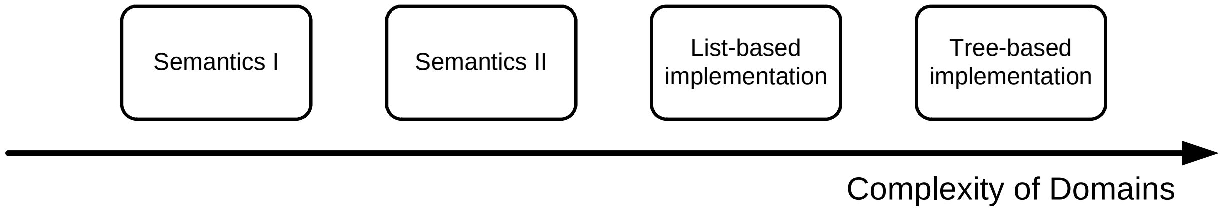


Figure 5.1: Complexity of Domains

$$
\begin{array}{rclcll}
 & & op_a & ::= & + \mid - \mid * \mid / & \text{(arithmetic op.)} \\
 & & op_b & ::= & \wedge \mid \vee & \text{(boolean op.)} \\
 & & op_r & ::= & < \mid = & \text{(relational op.)} \\
 & & & & & \\
x, y & \in & Var & & & \\
sel & \in & Sel & & & \\
p, q & \in & PExp & ::= & x \mid x.sel & \text{(pointer expr.)} \\
a & \in & AExp & ::= & p \mid Num \mid a_1\ op_a\ a_2 & \text{(arithmetic expr.)} \\
b & \in & BExp & ::= & \texttt{true} \mid \texttt{false} \mid & \text{(boolean expr.)} \\
 & & & & \neg b \mid b_1\ op_b\ b_2 \mid a_1\ op_r\ a_2 \mid & \\
 & & & & q \in p \mid q \subseteq p & \\
s, t & \in & Exp & ::= & p \mid a \mid b & \text{(expression)} \\
S, Prog & \in & Stmt & ::= & \texttt{skip} \mid p := s \mid S_1;\ S_2 \mid & \text{(statement)} \\
 & & & & \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \mid & \\
 & & & & \texttt{while}\ b\ \texttt{do}\ S \mid & \\
 & & & & p := \texttt{malloc} \mid p := \texttt{malloc set} \mid & \\
 & & & & x.\texttt{insert}(s) \mid x.\texttt{remove}(s) \mid & \\
 & & & & x := y.\texttt{selectAndRemove} &
\end{array}
$$

Figure 5.2: Syntactical Domains of RESET

the temporary variable $Temp$. $Temp$ is used to iterate over the contents of X without destroying X itself. For every element of X we check whether it is also an element of Y. In this case it is inserted into Z. In the end, Z contains the intersection of X and Y.

5.2 Static Semantics

A type system is specified in Figure 5.5. It restricts all the elements of one set to be of the same type. The syntax did not put any limitations on the sets and would for instance allow a single set to contain pointers as well as sets of pointers. This might actually be desirable since it resembles mathematical sets more closely. However it does raise problems in the second semantics that will be presented later. We assume variables and selectors to have a preassigned type. They are given by the function $\theta : (Var \rightarrow dataT) \cup (Sel \rightarrow dataT)$. Types are solely based on selectors because we do not distinguish between different pointer types. Possible types are specified in Figure 5.4. A program is correctly typed if we can derive the type `comm` for it using the given inference rules.

```
void intersection(Set X, Set Y) {
    Temp := malloc set;
    Temp := X;
    Z := malloc set;
    while (Temp != Empty)
    {
        p := Temp.selectAndRemove;

        if (p ∈ Y)
            Z.insert(p);
    }

    p := NULL;
    Temp := NULL;
}
```

Figure 5.3: RESET Program Computing the Intersection

$$
\begin{array}{rclcl}
 & & \mathit{type} & ::= & \mathit{dataT} \mid \texttt{comm} \\
t & \in & \mathit{dataT} & ::= & \texttt{bool} \mid \texttt{int} \\
 & & & \mid & \texttt{loc} \mid \mathit{dataT}\ \texttt{set}
\end{array}
$$

Figure 5.4: Types

5.3 Dynamic Semantics I

We give a nondeterministic structural operational semantics. Inference rules specify the semantics of the statements. They relate configurations which are pairs of statements and states. A state consists of three components, the stack, the heap and the set heap. Stack and heap cells may contain four different types of elements: booleans, integers, locations and set locations. The set heap stores sets. They are referenced by set locations. Sets are stored in single cells of the set heap (see Figure 5.7 for the details of the semantic domains). The definition of *Set* allows arbitrary nesting of sets and also different types of elements in one set. However, this is restricted by the type system of Figure 5.5.

Valuation functions like $\mathcal{X}$ and $\mathcal{B}$ are used to evaluate the meaning of expressions in the context of the state. They are shown in Figure 5.8. It is possible to give the semantics of expressions in this way, because expressions do not have any side-effects, i.e. they do not change the state. Their definitions are omitted for the most part, because they are mostly straight-forward. The evaluation of set membership ($q \in p$) and set inclusion ($q \subseteq p$) is

$$true : \texttt{bool} \qquad false : \texttt{bool}$$

$$\frac{b : \texttt{bool}}{\neg b : \texttt{bool}} \qquad \frac{b_1 : \texttt{bool} \quad b_2 : \texttt{bool}}{b_1 \; op_b \; b_2 : \texttt{bool}}$$

$$\frac{a_1 : \texttt{int} \quad a_2 : \texttt{int}}{a_1 \; op_r \; a_2 : \texttt{bool}} \qquad \frac{a_1 : \texttt{int} \quad a_2 : \texttt{int}}{a_1 \; op_a \; a_2 : \texttt{int}}$$

$$\frac{t = \theta(x)}{x : t} \qquad \frac{x : \texttt{loc} \quad t = \theta(sel)}{x.sel : t}$$

$$\frac{a : t \quad p : t \; \texttt{set}}{a \in p : \texttt{bool}} \qquad \frac{q : t \; \texttt{set} \quad p : t \; \texttt{set}}{q \subseteq p : \texttt{bool}}$$

$$skip : \texttt{comm} \qquad \frac{p : t \quad s : t}{p := s : \texttt{comm}}$$

$$\frac{S_1 : \texttt{comm} \quad S_2 : \texttt{comm}}{S_1; S_2 : \texttt{comm}} \qquad \frac{b : \texttt{bool} \quad S_1 : \texttt{comm} \quad S_2 : \texttt{comm}}{\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2 : \texttt{comm}}$$

$$\frac{b : \texttt{bool} \quad S : \texttt{comm}}{\texttt{while } b \texttt{ do } S : \texttt{comm}} \qquad \frac{p : \texttt{loc}}{p := \texttt{malloc} : \texttt{comm}}$$

$$\frac{p : t \; \texttt{set} \quad a : t}{p.\texttt{insert}(a) : \texttt{comm}} \qquad \frac{p : t \; \texttt{set}}{p := \texttt{malloc set} : \texttt{comm}}$$

$$\frac{p : t \; \texttt{set} \quad a : t}{a := p.\texttt{selectAndRemove} : \texttt{comm}} \qquad \frac{p : t \; \texttt{set} \quad a : t}{p.\texttt{remove}(a) : \texttt{comm}}$$

Figure 5.5: Type System

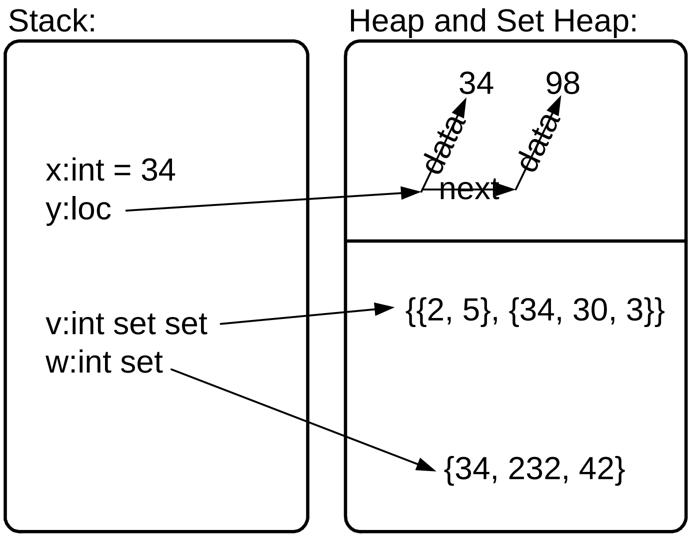


Figure 5.6: Example State in Semantics I

probably most interesting here. Since sets are completely stored in single cells, set inclusion translates to the common mathematical set inclusion. The same holds for set membership, where we have to look at two cases. Either p is a set of sets or it is just a set of primitive values or locations. In the former case we have to check for set membership of the referenced set by p and not of its location. This will be more complicated in the Semantics II.

The inference rules which define the semantics of statements are displayed in Figure 5.9. The non-standard part of the inference rules are again the rules concerning sets. The same case distinction as in the test of set membership also applies for the inference rules concerning assignments. When assigning sets we do not assign its location but its contents. Being able to assign set locations would introduce aliasing problems, which we want to avoid. $p :=$ `malloc set` stores a new set location at p and initializes the set heap at the new location with an empty set. This could not be done using a `malloc set`-expression plus assignment, since it is not possible to assign set locations. The statements for element insertion and removal also have to deal with these two cases. Otherwise, they translate to set insertion and removal respectively. The same holds for the `.selectAndRemove`-statement. You may have wondered why it is a nondeterministic semantics. The nondeterminism is introduced by the `.selectAndRemove`-statement, since it nondeterministically selects one of the elements of the specified set. All other statements are deterministic.

The semantics of a program can be seen as the finite and infinite sequences of states that follow its execution. More formally: $[\![Prog]\!] \ni \langle Prog_1, (\sigma_1, \eta_1)\rangle\langle Prog_2, (\sigma_2, \eta_2)\rangle \ldots$ where $Prog_1 = Prog$ and $\langle Prog_i, (\sigma_i, \eta_i)\rangle \triangleright \langle Prog_{i+1}, (\sigma_{i+1}, \eta_{i+1})\rangle$ for all i.

$$
\begin{array}{rcll}
\xi & \in & \mathit{Loc} & \\
\psi & \in & \mathit{SetLoc} & \\
b & \in & \mathbb{B} & = \{\mathbf{0}, \mathbf{1}\} \\
z & \in & \mathbb{Z} & = \{\dots, -1, 0, 1, \dots\} \\
s & \in & \mathit{Set} & = \bigcup\limits_{i \in \mathbb{N}} f^i(\varnothing) \text{ where } f = \lambda X.\mathcal{P}(X \cup \mathbb{B} \cup \mathbb{Z} \cup \mathit{Loc}) \\
e, i, j & \in & \mathit{Item} & = \mathbb{B} \cup \mathbb{Z} \cup \mathit{Loc} \cup \mathit{SetLoc} \\
\sigma & \in & \mathit{Stack} & = \mathit{Var} \rightharpoonup \mathit{Item} \\
\eta & \in & \mathit{Heap} & = (\mathit{Loc} \times \mathit{Sel}) \rightharpoonup \mathit{Item} \\
\varsigma & \in & \mathit{SetHeap} & = \mathit{SetLoc} \rightharpoonup \mathit{Set} \\
(\sigma, \eta, \varsigma) & \in & \mathit{State} & = \mathit{Stack} \times \mathit{Heap} \times \mathit{SetHeap} \\
\langle S, (\sigma, \eta, \varsigma) \rangle & \in & \mathit{Configuration} & = \mathit{Stmt} \times \mathit{State} \\
 & & & \\
 & & \mathcal{A} & : \mathit{AExp} \to \mathit{State} \rightharpoonup \mathit{Item} \\
 & & \mathcal{P} & : \mathit{PExp} \to \mathit{State} \rightharpoonup \mathit{Item} \\
 & & \mathcal{B} & : \mathit{BExp} \to \mathit{State} \rightharpoonup \mathbb{B} \\
 & & \mathcal{N} & : \mathit{Num} \to \mathbb{Z} \\
 & & \mathcal{X} & : \mathit{Exp} \to \mathit{State} \rightharpoonup \mathit{Item} \\
 & & & \\
 & & . \rhd . & \subseteq \mathit{Configuration} \times \mathit{Configuration} \\
 & & & \\
 & & [\![.]\!] & : \mathit{Prog} \to \mathcal{P}(\mathit{Configuration}^* \cup \mathit{Configuration}^\omega)
\end{array}
$$

Figure 5.7: Semantic Domains

$$
\begin{array}{rcl}
\mathcal{P}[\![x]\!](\sigma, \eta, \varsigma) & = & \sigma(x) \\
\mathcal{P}[\![x.sel]\!](\sigma, \eta, \varsigma) & = & \eta(\sigma(x), sel) \\
 & & \\
\mathcal{B}[\![q \in p]\!](\sigma, \eta, \varsigma) & = & \begin{cases} \varsigma(\mathcal{P}[\![q]\!](\sigma, \eta, \varsigma)) \in \varsigma(\mathcal{P}[\![p]\!](\sigma, \eta, \varsigma)), \text{ if } \mathcal{P}[\![q]\!](\sigma, \eta, \varsigma) \in \mathit{SetLoc} \\ \mathcal{P}[\![q]\!](\sigma, \eta, \varsigma) \in \varsigma(\mathcal{P}[\![p]\!](\sigma, \eta, \varsigma)), \text{ otherwise} \end{cases} \\
\mathcal{B}[\![q \subseteq p]\!](\sigma, \eta, \varsigma) & = & \varsigma(\mathcal{P}[\![q]\!](\sigma, \eta, \varsigma)) \subseteq \varsigma(\mathcal{P}[\![p]\!](\sigma, \eta, \varsigma)) \\
 & & \\
\mathcal{X}[\![s]\!](\sigma, \eta, \varsigma) & = & \begin{cases} \mathcal{P}[\![s]\!](\sigma, \eta, \varsigma), \text{ if } s \in \mathit{PExp}, \\ \mathcal{A}[\![s]\!](\sigma, \eta, \varsigma), \text{ if } s \in \mathit{AExp}, \\ \mathcal{B}[\![s]\!](\sigma, \eta, \varsigma), \text{ if } s \in \mathit{BExp} \end{cases}
\end{array}
$$

Figure 5.8: Exemplary Semantics of Expressions

$$\langle x := s, (\sigma,\eta,\varsigma)\rangle \triangleright \langle \texttt{skip}, (\sigma[x \mapsto \mathcal{X}[\![s]\!](\sigma,\eta,\varsigma)], \eta, \varsigma)\rangle \quad \text{if } \mathcal{X}[\![s]\!](\sigma,\eta,\varsigma) \in (\mathit{Item} \setminus \mathit{SetLoc}) \qquad \text{[Assignment]}$$

$$\langle x := s, (\sigma,\eta,\varsigma)\rangle \triangleright \langle \texttt{skip}, (\sigma,\eta,\varsigma[\sigma(x) \mapsto \varsigma(\mathcal{X}[\![s]\!](\sigma,\eta,\varsigma))])\rangle \quad \text{if } \mathcal{X}[\![s]\!](\sigma,\eta,\varsigma) \in \mathit{SetLoc} \qquad \text{[Assignment-Set]}$$

$$\langle x.\mathit{sel} := s, (\sigma,\eta,\varsigma)\rangle \triangleright \langle \texttt{skip}, (\sigma,\eta[(\sigma(x),\mathit{sel}) \mapsto \mathcal{X}[\![s]\!](\sigma,\eta,\varsigma)],\varsigma)\rangle \quad \text{if } \mathcal{X}[\![s]\!](\sigma,\eta,\varsigma) \in (\mathit{Item} \setminus \mathit{SetLoc}) \qquad \text{[Assignment-Heap]}$$

$$\langle x.\mathit{sel} := s, (\sigma,\eta,\varsigma)\rangle \triangleright \langle \texttt{skip}, (\sigma,\eta,\varsigma[\eta(\sigma(x),\mathit{sel}) \mapsto \varsigma(\mathcal{X}[\![s]\!](\sigma,\eta,\varsigma))])\rangle \quad \text{if } \mathcal{X}[\![s]\!](\sigma,\eta,\varsigma) \in \mathit{SetLoc} \qquad \text{[Assignment-Heap-Set]}$$

$$\langle \texttt{skip};\ S, (\sigma,\eta,\varsigma)\rangle \triangleright \langle S, (\sigma,\eta,\varsigma)\rangle \qquad \text{[Skip-Elimination]}$$

$$\frac{\langle S_1, (\sigma_1,\eta_1,\varsigma_1)\rangle \triangleright \langle S_2, (\sigma_2,\eta_2,\varsigma_2)\rangle}{\langle S_1;\ S, (\sigma_1,\eta_1,\varsigma_1)\rangle \triangleright \langle S_2;\ S, (\sigma_2,\eta_2,\varsigma_2)\rangle} \qquad \text{[Seq. Composition]}$$

$$\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2, (\sigma,\eta,\varsigma)\rangle \triangleright \langle S_1, (\sigma,\eta,\varsigma)\rangle \quad \text{where } \mathcal{B}[\![b]\!](\sigma,\eta,\varsigma) = \mathbf{1} \qquad \text{[If-True]}$$

$$\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2, (\sigma,\eta,\varsigma)\rangle \triangleright \langle S_2, (\sigma,\eta,\varsigma)\rangle \quad \text{where } \mathcal{B}[\![b]\!](\sigma,\eta,\varsigma) = \mathbf{0} \qquad \text{[If-False]}$$

$$\langle \texttt{while}\ b\ \texttt{do}\ S, (\sigma,\eta,\varsigma)\rangle \triangleright \langle S;\ \texttt{while}\ b\ \texttt{do}\ S, (\sigma,\eta,\varsigma)\rangle \quad \text{where } \mathcal{B}[\![b]\!](\sigma,\eta,\varsigma) = \mathbf{1} \qquad \text{[While-True]}$$

$$\langle \texttt{while}\ b\ \texttt{do}\ S, (\sigma,\eta,\varsigma)\rangle \triangleright \langle \texttt{skip}, (\sigma,\eta,\varsigma)\rangle \quad \text{where } \mathcal{B}[\![b]\!](\sigma,\eta,\varsigma) = \mathbf{0} \qquad \text{[While-False]}$$

$$\langle x := \texttt{malloc}, (\sigma,\eta,\varsigma)\rangle \triangleright \langle \texttt{skip}, (\sigma[x \mapsto \xi],\eta,\varsigma)\rangle \quad \text{where } \xi \in \mathit{Loc} \text{ and } \xi \notin (\mathit{im}(\sigma) \cup \mathit{dom}(\eta) \cup \mathit{im}(\eta) \cup \bigcup \mathit{im}(\varsigma)) \qquad \text{[Malloc]}$$

$$\langle x.\mathit{sel} := \texttt{malloc}, (\sigma,\eta,\varsigma)\rangle \triangleright \langle \texttt{skip}, (\sigma,\eta[(\sigma(x),\mathit{sel}) \mapsto \xi],\varsigma)\rangle \quad \text{where } \xi \in \mathit{Loc} \text{ and } \xi \notin (\mathit{im}(\sigma) \cup \mathit{dom}(\eta) \cup \mathit{im}(\eta) \cup \bigcup \mathit{im}(\varsigma)) \qquad \text{[Malloc-Heap]}$$

$$\langle x := \texttt{malloc set}, (\sigma,\eta,\varsigma)\rangle \triangleright \langle \texttt{skip}, (\sigma[x \mapsto \psi],\eta,\varsigma[\psi \mapsto \emptyset])\rangle \quad \text{where } \psi \in \mathit{SetLoc} \text{ and } \psi \notin (\mathit{im}(\sigma) \cup \mathit{im}(\eta) \cup \mathit{dom}(\varsigma)) \qquad \text{[Malloc-Set]}$$

$$\langle x.\mathit{sel} := \texttt{malloc set}, (\sigma,\eta,\varsigma)\rangle \triangleright \langle \texttt{skip}, (\sigma,\eta[(\sigma(x),\mathit{sel}) \mapsto \psi],\varsigma[\psi \mapsto \emptyset])\rangle \quad \text{where } \psi \in \mathit{SetLoc} \text{ and } \psi \notin (\mathit{im}(\sigma) \cup \mathit{im}(\eta) \cup \mathit{dom}(\varsigma)) \qquad \text{[Malloc-Set-Heap]}$$

$$\langle x.\texttt{insert}(s), (\sigma,\eta,\varsigma)\rangle \triangleright \langle \texttt{skip}, (\sigma,\eta,\varsigma[\sigma(x) \mapsto (\varsigma(\sigma(x)) \cup \{i\})])\rangle \quad \text{where } i = \begin{cases} \varsigma(\mathcal{X}[\![s]\!](\sigma,\eta,\varsigma)), & \text{if } \mathcal{X}[\![s]\!](\sigma,\eta,\varsigma) \in \mathit{SetLoc} \\ \mathcal{X}[\![s]\!](\sigma,\eta,\varsigma), & \text{otherwise} \end{cases} \qquad \text{[Set-Insert]}$$

$$\langle x.\texttt{remove}(s), (\sigma,\eta,\varsigma)\rangle \triangleright \langle \texttt{skip}, (\sigma,\eta,\varsigma[\sigma(x) \mapsto (\varsigma(\sigma(x)) \setminus \{i\})])\rangle \quad \text{where } i = \begin{cases} \varsigma(\mathcal{X}[\![s]\!](\sigma,\eta,\varsigma)), & \text{if } \mathcal{X}[\![s]\!](\sigma,\eta,\varsigma) \in \mathit{SetLoc} \\ \mathcal{X}[\![s]\!](\sigma,\eta,\varsigma), & \text{otherwise} \end{cases} \qquad \text{[Set-Remove]}$$

$$\langle x := y.\texttt{selectAndRemove}, (\sigma,\eta,\varsigma)\rangle \triangleright \langle \texttt{skip}, (\sigma[x \mapsto \mathit{el}],\eta,\varsigma[\sigma(y) \mapsto (\varsigma(\sigma(y)) \setminus \{\mathit{el}\})])\rangle \quad \text{where } \mathit{el} \in \varsigma(\sigma(y)) \text{ and } \mathit{el} \in (\mathit{Item} \setminus \mathit{SetLoc}) \qquad \text{[Set-SelectRemove]}$$

$$\langle x := y.\texttt{selectAndRemove}, (\sigma,\eta,\varsigma)\rangle \triangleright \langle \texttt{skip}, (\sigma,\eta,\varsigma[\sigma(y) \mapsto (\varsigma(\sigma(y)) \setminus \{\mathit{el}\})][\sigma(x) \mapsto \mathit{el}])\rangle \quad \text{where } \mathit{el} \in \varsigma(\sigma(y)) \text{ and } \mathit{el} \in \mathit{Set} \qquad \text{[Set-SelectRemove-Set]}$$

Figure 5.9: Structural Operational Semantics

5.4 Semantics II

We give an alternative semantics that is somewhat closer to implementations of sets. In contrast to the first semantics sets of sets are not stored in a single set heap cell. They are spread over the set heap via set locations. The domain *Set* is in fact the only difference in the domains of the two semantics (see Figure 5.11). Figure 5.10 displays the same state as Figure 5.6.

Spreading sets over the heap introduces some problems that have to be dealt with in the semantics of set expressions and the inference rules. Comparing two sets becomes more complicated in this case. If we compare the two sets $\{\psi'\}$ and $\{\psi''\}$ for instance, where ψ' and ψ'' are set locations, we have to compare $\varsigma'(\psi')$ and $\varsigma'(\psi'')$ with each other. These might contain other set locations again... The predicate $\approx$ is introduced for this purpose. It descends into the heap until reaching primitive values or locations. $\approx$ is well-defined. This is ensured by the type system. It prevents cyclic set relations and ensures that $\approx$ will eventually reach some base case. Without the type system this acyclicity property would have to be ensured by the semantics itself making it much more complicated.

The semantics of set expressions is defined in terms of $\approx$. q is an element of set p if the set referenced by p contains some element that is equal to q in the sense of $\approx$. Notice the similarity to the definition of the *isElement* instrumentation predicates in the two set implementations of Chapter 4. Set inclusion is handled similarly as can be seen in Figure 5.12.

Figure 5.13 shows the inference rules that differ. The two inference rules for `malloc set` differ only marginally. Set locations introduced by these statements may not occur in the image of the set heap. In the first semantics, the set heap could not contain set locations making this condition superfluous there. When inserting elements into sets the inference rule makes sure that we do not insert duplicate elements. In the first semantics this was ensured by the nature of mathematical sets. As we have seen, different set locations may represent equal sets in this case. Again, $\approx$ is utilized to deal with this problem. The same is done for element removal.

Only one of the two inference rules for `.selectAndRemove` differs from the original ones. If the statement is applied to sets of sets, it should return a set. However, in contrast to the first semantics, the set contains set locations. So we return the set at that set location.

5.5 Comparison

The main difference between the two semantics is the handling of set elements that are themselves sets. While Semantics I stores entire sets within a single heap cell, Semantics II spreads the set contents over the heap by just inserting set locations. The latter corresponds more closely to set implementations in imperative languages without sets as

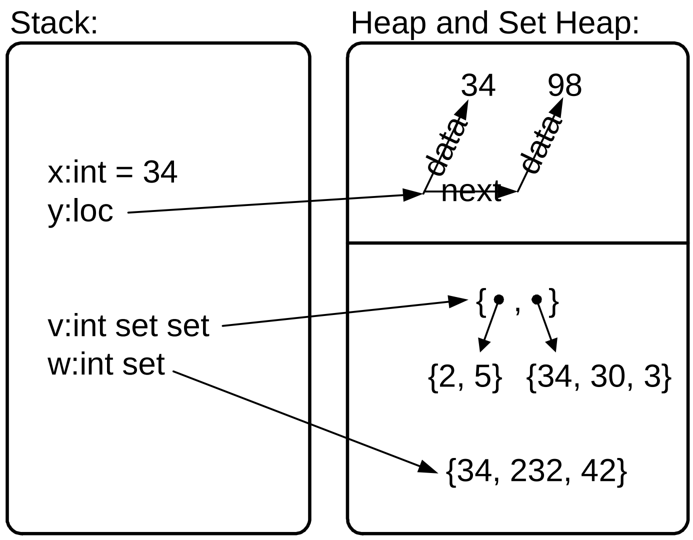


Figure 5.10: Example State in Semantics II

primitives. In set implementations the data structures that are used to represent the set are spread over the heap in a similar way. Also, elements may be more complex than just primitive values.

Comparing sets and testing set membership become more complicated in the alternative case. Two sets may be equal, but their elements may be at different locations in the heap. Another implication of spreading sets over the heap is aliasing, which allows elements to be manipulated after insertion and thus yielding unexpected results.

Without the type system both semantics allow to insert a set into itself or into one of its constituents. In the first semantics this would not cause any problems, since the contents of the set are copied completely. Semantics II would allow to create circular set definitions without the restriction of the type system though. The relation between the two options roughly corresponds to deep vs. shallow copying (just for elements that are sets, not for other pointers).

We would now like to formally relate the two given semantics, to which we will refer to as *Semantics I* and *Semantics II*. For this purpose we define a relation between the domains of the two semantics.

Definition 15 (Heap Correspondence Relation) *A relation* $\equiv$ *is called a* Heap Correspondence Relation *on* $(\eta, \varsigma) \in \text{Heap} \times \text{SetHeap}, (\eta', \varsigma') \in \text{Heap}' \times \text{SetHeap}'$*, iff*

$$\equiv \;\stackrel{def}{=}\; \equiv_{ss} \cup \equiv_{slsl} \cup \equiv_{ssl} \cup \equiv_{ll} \cup \equiv_{bb} \cup \equiv_{zz}$$

with

$$
\begin{array}{rclcl}
\xi' & \in & \mathit{Loc}' & & \\
\psi' & \in & \mathit{SetLoc}' & & \\
b & \in & \mathbb{B} & = & \{\mathbf{0},\mathbf{1}\} \\
z & \in & \mathbb{Z} & = & \{\ldots,-1,0,1,\ldots\} \\
e',i',j' & \in & \mathit{Item}' & = & \mathbb{B}\cup\mathbb{Z}\cup\mathit{Loc}'\cup\mathit{SetLoc}' \\
s' & \in & \mathit{Set}' & = & \mathcal{P}(\mathit{Item}') \\
\sigma' & \in & \mathit{Stack}' & = & \mathit{Var}\rightharpoonup\mathit{Item}' \\
\eta' & \in & \mathit{Heap}' & = & (\mathit{Loc}'\times\mathit{Sel})\rightharpoonup\mathit{Item}' \\
\varsigma' & \in & \mathit{SetHeap}' & = & \mathit{SetLoc}'\rightharpoonup\mathit{Set}' \\
(\sigma',\eta',\varsigma') & \in & \mathit{State}' & = & \mathit{Stack}'\times\mathit{Heap}'\times\mathit{SetHeap}' \\
\langle S',(\sigma',\eta',\varsigma')\rangle & \in & \mathit{Configuration}' & = & \mathit{Stmt}\times\mathit{State}' \\
\\
 & & .\approx .. & : & ((\mathit{Item}'\cup\mathit{Set}')\times(\mathit{Item}'\cup\mathit{Set}'))\rightarrow\mathit{State}'\rightharpoonup\mathbb{B} \\
\\
 & & \mathcal{A}' & : & \mathit{AExp}\rightarrow\mathit{State}'\rightharpoonup\mathit{Item}' \\
 & & \mathcal{P}' & : & \mathit{PExp}\rightarrow\mathit{State}'\rightharpoonup\mathit{Item}' \\
 & & \mathcal{B}' & : & \mathit{BExp}\rightarrow\mathit{State}'\rightharpoonup\mathbb{B} \\
 & & \mathcal{N}' & : & \mathit{Num}\rightarrow\mathbb{Z} \\
 & & \mathcal{X}' & : & \mathit{Exp}\rightarrow\mathit{State}'\rightharpoonup\mathit{Item}' \\
\\
 & & .\rhd' . & \subseteq & \mathit{Configuration}'\times\mathit{Configuration}' \\
\\
 & & [\![.]\!]' & : & \mathit{Prog}\rightarrow\mathcal{P}(\mathit{Configuration}'^{*}\cup\mathit{Configuration}'^{\omega})
\end{array}
$$

Figure 5.11: Semantic Domains II

$$
\begin{array}{rcl}
(i'\approx j')(\sigma',\eta',\varsigma') & = & \begin{cases} i'=j',\ \text{if } i',j'\in\mathbb{B}\cup\mathbb{Z}\cup\mathit{Loc}' \\ (\varsigma'(i')\approx\varsigma'(j'))(\sigma',\eta',\varsigma'),\ \text{if } i',j'\in\mathit{SetLoc}' \\ (i'\approx\varsigma'(j'))(\sigma',\eta',\varsigma'),\ \text{if } i'\in\mathit{Set}', j'\in\mathit{SetLoc}' \\ (\forall x\in i'.\exists z\in j'.(z\approx x)(\sigma',\eta',\varsigma')) \\ \wedge(\forall x\in j'.\exists z\in i'.(z\approx x)(\sigma',\eta',\varsigma')),\ \text{if } i',j'\in\mathit{Set}' \end{cases} \\
\mathcal{B}'[\![q\in p]\!](\sigma',\eta',\varsigma') & = & \exists z.(z\in\varsigma'(\mathcal{P}'[\![p]\!](\sigma',\eta',\varsigma'))\wedge(z\approx\mathcal{P}'[\![q]\!](\sigma',\eta',\varsigma'))(\sigma',\eta',\varsigma')) \\
\mathcal{B}'[\![q\subseteq p]\!](\sigma',\eta',\varsigma') & = & \forall x\in\varsigma'(\mathcal{P}'[\![q]\!](\sigma',\eta',\varsigma')).\exists z.(z\approx x)(\sigma',\eta',\varsigma')\wedge(z\in\varsigma'(\mathcal{P}'[\![p]\!](\sigma',\eta',\varsigma')) \\
\mathcal{B}'[\![q=p]\!](\sigma',\eta',\varsigma') & = & (\mathcal{P}'[\![q]\!](\sigma',\eta',\varsigma')\approx\mathcal{P}'[\![p]\!](\sigma',\eta',\varsigma'))(\sigma',\eta',\varsigma')
\end{array}
$$

Figure 5.12: Differences in the Semantics of Expressions

$\langle x := \texttt{malloc set}, (\sigma', \eta', \varsigma')\rangle \rhd' \langle \texttt{skip}, (\sigma'[x \mapsto \psi'], \eta', \varsigma'[\psi' \mapsto \emptyset])\rangle$ where $\psi' \in \mathit{SetLoc}'$ and $\psi' \notin (im(\sigma') \cup im(\eta') \cup dom(\varsigma') \cup \bigcup im(\varsigma'))$	[Malloc-Set']
$\langle x.sel := \texttt{malloc set}, (\sigma', \eta', \varsigma')\rangle \rhd' \langle \texttt{skip}, (\sigma', \eta'[(\sigma'(x), sel) \mapsto \psi'], \varsigma'[\psi' \mapsto \emptyset])\rangle$ where $\psi' \in \mathit{SetLoc}'$ and $\psi' \notin (im(\sigma') \cup im(\eta') \cup dom(\varsigma') \cup \bigcup im(\varsigma'))$	[Malloc-Set-Heap']
$\langle x.\texttt{insert}(s), (\sigma', \eta', \varsigma')\rangle \rhd' \langle \texttt{skip}, (\sigma', \eta', \varsigma'[\sigma'(x) \mapsto i])\rangle$ where $\varsigma'(\sigma'(x))$ def. and $i = \begin{cases} \varsigma'(\sigma'(x)), & \text{if } \exists z. z \in \varsigma'(\sigma'(x)) \wedge (z \approx \mathcal{X}'[\![s]\!](\sigma', \eta', \varsigma'))(\sigma', \eta', \varsigma') \\ \varsigma'(\sigma'(x)) \cup \{\mathcal{X}'[\![s]\!](\sigma', \eta', \varsigma')\}, & \text{otherwise} \end{cases}$	[Set-Insert']
$\langle x.\texttt{remove}(s), (\sigma', \eta', \varsigma')\rangle \rhd' \langle \texttt{skip}, (\sigma', \eta', \varsigma'[\sigma'(x) \mapsto i])\rangle$ where $\varsigma'(\sigma'(x))$ def. and $i = \{j \mid j \in \varsigma'(\sigma'(x)) \wedge \neg(j \approx \mathcal{X}'[\![s]\!](\sigma', \eta', \varsigma'))(\sigma', \eta', \varsigma')\}$	[Set-Remove']
$\langle x := y.\texttt{selectAndRemove}, (\sigma', \eta', \varsigma')\rangle \rhd' \langle \texttt{skip}, (\sigma'[x \mapsto el'], \eta', \varsigma'[\sigma'(y) \mapsto (\varsigma'(\sigma'(y)) \setminus \{el'\})])\rangle$ where $el' \in \varsigma'(\sigma'(y))$ and $el' \in (\mathit{Item}' \setminus \mathit{SetLoc}')$	[Set-SelectRemove']
$\langle x := y.\texttt{selectAndRemove}, (\sigma', \eta', \varsigma')\rangle \rhd' \langle \texttt{skip}, (\sigma', \eta', \varsigma'[\sigma'(y) \mapsto (\varsigma'(\sigma'(y)) \setminus \{el'\})][\sigma'(x) \mapsto \varsigma'(el')])\rangle$ where $el' \in \varsigma'(\sigma'(y))$ and $el' \in \mathit{SetLoc}'$	[Set-SelectRemove-Set']

Figure 5.13: Differences in Structural Operational Semantics

- $\equiv_{bb} \subseteq \mathbb{B} \times \mathbb{B}$
  $b_1 \equiv_{bb} b_2 \Leftrightarrow (b_1 \Leftrightarrow b_2)$
- $\equiv_{zz} \subseteq \mathbb{Z} \times \mathbb{Z}$
  $z_1 \equiv_{zz} z_2 \Leftrightarrow (z_1 = z_2)$
- $\equiv_{ll} \subseteq \mathrm{Loc} \times \mathrm{Loc}'$
  $\xi \equiv_{ll} \xi' \Rightarrow (\forall sel \in \pi_2(dom(\eta)).(\eta(\xi, sel) \equiv \eta'(\xi', sel) \vee (\eta(\xi, sel)\ undef. \wedge \eta'(\xi', sel)\ undef.)) \wedge \forall \xi''.(\xi \equiv_{ll} \xi'' \Rightarrow \xi' = \xi'') \wedge \forall \xi''.(\xi'' \equiv_{ll} \xi' \Rightarrow \xi = \xi''))$
- $\equiv_{slsl} \subseteq \mathrm{SetLoc} \times \mathrm{SetLoc}'$
  $\psi \equiv_{slsl} \psi' \Leftrightarrow \varsigma(\psi) \equiv_{ss} \varsigma'(\psi')$
- $\equiv_{ssl} \subseteq \mathrm{Set} \times \mathrm{SetLoc}'$
  $s \equiv_{ssl} \psi' \Leftrightarrow s \equiv_{ss} \varsigma'(\psi')$
- $\equiv_{ss} \subseteq \mathrm{Set} \times \mathrm{Set}'$
  $s \equiv_{ss} s' \Leftrightarrow (\forall i \in s. \exists i' \in s'. i \equiv i' \wedge \forall i' \in s'. \exists i \in s. i \equiv i')$

The conditions for $\equiv_{bb}$ and $\equiv_{zz}$ require the $\equiv$-relation to follow the usual semantics of equality for boolean and integer values. If locations correspond with respect to $\equiv$, their selector-fields also have to correspond. This requires the second heap η' to be homomorphic to the first heap η. The condition for $\equiv_{ss}$ is probably the most interesting, since it relates sets, which are represented differently in the two semantics. Elements of sets which are sets themselves need to have a corresponding set location in the other heap. Figure 5.14

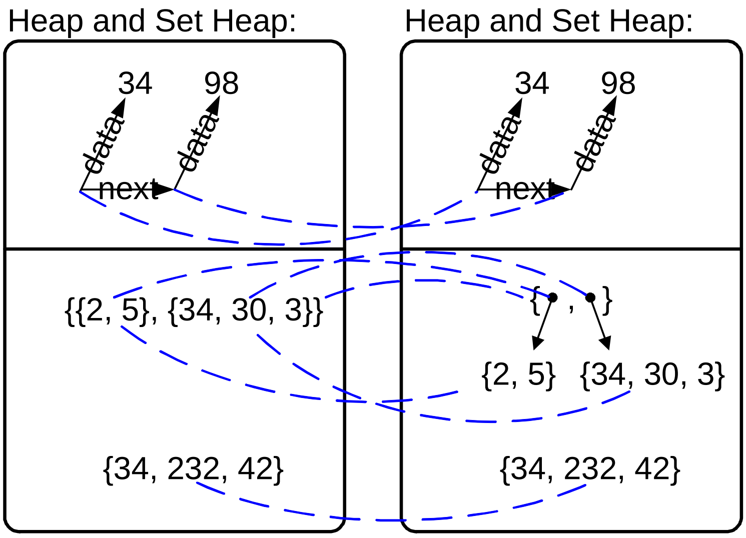


Figure 5.14: Example of Heap Correspondence Relation

gives an example of a *Heap Correspondence Relation*. We omitted the $\equiv_{zz}$ part to make it more readable.

Lemma 1 (Heap Correspondence Relation Stability) *If* $\equiv$ *is a* Heap Correspondence Relation *on* (η, ς) *and* (η', ς') *and* $\psi \in \text{SetLoc}, \psi' \in \text{SetLoc}'$ *and* $\psi \notin (im(\sigma) \cup im(\eta) \cup dom(\varsigma)), \psi' \notin (im(\sigma') \cup im(\eta') \cup dom(\varsigma') \cup \bigcup im(\varsigma'))$ *and* $i \in \text{Item}, i' \in \text{Item}'$ *with* $i \neq \psi, i \neq \psi', i \equiv i'$ *and* $x \in \text{Set}, x' \in \text{Set}'$,
then there exist Heap Correspondence Relations $\equiv'$ *and* $\equiv''$ *with*

$$(\eta, \varsigma[\psi \mapsto x]) \equiv' (\eta', \varsigma') \text{ and } i \equiv' i'$$

and

$$(\eta, \varsigma) \equiv'' (\eta', \varsigma'[\psi' \mapsto x']) \text{ and } i \equiv'' i'$$

This expresses that we can change unaliased locations of the set heap without losing the fact that a *Heap Correspondence Relation* exists. In addition, the relation between other items is preserved. We will refer to this lemma as the *Stability Lemma*.

Proof Sketch:

Give a new *Heap Correspondence Relation* by adding and removing appropriate elements from the previous relation. For the full proof see Appendix A. ■

Now we can go on to define correspondence of states.

Definition 16 (Corresponding States) *Two states* $(\sigma, \eta, \varsigma) \in \mathrm{State}, (\sigma', \eta', \varsigma') \in \mathrm{State}'$ *are called corresponding, denoted by* $(\sigma, \eta, \varsigma) \cong (\sigma', \eta', \varsigma')$*, iff there exists a* Heap Correspondence Relation $\equiv$ *on* $(\eta, \varsigma), (\eta', \varsigma')$ *with*

$$\sigma(x) \equiv \sigma'(x) \vee (\sigma(x) \textit{ undef.} \wedge \sigma'(x) \textit{ undef.}) \textit{ for all } x \in \mathit{dom}(\sigma)$$

The stack serves as an anchor to connect the two states. The existence of a *Heap Correspondence Relation* then requires the reachable part of the heap and the set heap of the first state to be homomorphically represented by the second state.

Lemma 2 (Set Injectivity) *If* $i, j \in \mathrm{Set}, i' \in \mathrm{Set}'$ *then*

$$i \equiv i' \wedge j \equiv i' \Rightarrow i = j$$

That is, sets are unique in Semantics I. For a proof see Appendix A.

The following lemma proves two properties of the relation between a *Heap Correspondence Relation* and the $\approx$-predicate defined in Figure 5.12. It will help in the proof of the *Expressions Coincide Lemma*.

Lemma 3 ($\equiv$ / $\approx$ Relation) *Let* $\equiv$ *be a* Heap Correspondence Relation *on* $(\eta, \varsigma), (\eta', \varsigma')$ *and let* σ, σ' *be arbitrary stacks. Then the following holds*

$$1.\quad i \equiv i' \wedge i \equiv j' \Rightarrow (i' \approx j')(\sigma', \eta', \varsigma')$$

and

$$2.\quad i \equiv i' \wedge (i' \approx j')(\sigma', \eta', \varsigma') \Rightarrow i \equiv j'$$

That is the following diagram commutes:

$$\begin{tikzcd} i \arrow[r, phantom, "\equiv"] \arrow[dr, phantom, "\equiv" sloped] & i' \arrow[d, phantom, "\approx" sloped] \\ & j' \end{tikzcd}$$

Proof Sketch:

Proof by case distinction on the type of i, i', j'. Induction where $i \in \mathit{Set}$.
The full proof is in the Appendix A. ■

Theorem 2 (Expressions Coincide) *If two states correspond by the previous definition, then all expressions evaluate to equivalent values in both semantics:*

$$(\sigma, \eta, \varsigma) \cong (\sigma', \eta', \varsigma') \Rightarrow$$
$$\mathcal{X}[\![s]\!](\sigma, \eta, \varsigma) \equiv \mathcal{X}'[\![s]\!](\sigma', \eta', \varsigma') \vee (\mathcal{X}[\![s]\!](\sigma, \eta, \varsigma) \textit{ undef.} \wedge \mathcal{X}'[\![s]\!](\sigma', \eta', \varsigma') \textit{ undef.})$$

For boolean and integer expressions this means that they evaluate to the same value, since $z_1 \equiv z_2 \Leftrightarrow z_1 \equiv_{zz} z_2 \Leftrightarrow z_1 = z_2$ and $b_1 \equiv b_2 \Leftrightarrow b_1 \equiv_{bb} b_2 \Leftrightarrow (b_1 \Leftrightarrow b_2)$.
The result will be used in the proof of the *Simulation Lemma*. In addition it shows that the definition of corresponding states is sensible.

Proof Sketch:

Proof by induction over the structure of the formula. Use the definition of a *Heap Correspondence Relation* and the previous lemma for most of the base cases. The step cases are trivial.
See Appendix A for a proof. ■

Definition 17 (Corresponding Statements) *Two statements S_1, S_2 correspond, $S_1 \sim S_2$, iff $S_2 = T(S_1)$, where the transformation $T : \mathrm{Stmt} \to \mathrm{Stmt}$ is defined as follows:*

`skip`	$\mapsto$	`skip`	
$p := s$	$\mapsto$	$p :=$ `malloc set`; $p := s$	if $p : t$ `set`
$p := s$	$\mapsto$	$p := s$	otherwise
S_1; S_2	$\mapsto$	$T(S_1)$; $T(S_2)$	
`if` b `then` S_1 `else` S_2	$\mapsto$	`if` b `then` $T(S_1)$ `else` $T(S_2)$	
`while` b `do` S	$\mapsto$	`while` b `do` $T(S)$	
$p :=$ `malloc`	$\mapsto$	$p :=$ `malloc`	
$p :=$ `malloc set`	$\mapsto$	$p :=$ `malloc set`	
x.`insert`(p)	$\mapsto$	$x_{temp} :=$ `malloc set`; $x_{temp} := x$; $x :=$ `malloc set`; $x := x_{temp}$; x.`insert`(p)	
x.`remove`(p)	$\mapsto$	$x_{temp} :=$ `malloc set`; $x_{temp} := x$; $x :=$ `malloc set`; $x := x_{temp}$; x.`remove`(p)	
$x := y$.`selectAndRemove`	$\mapsto$	$y_{temp} :=$ `malloc set`; $y_{temp} := y$; $y :=$ `malloc set`; $y := y_{temp}$; $x :=$ `malloc set`; $x := y$.`selectAndRemove`	if $x : t$ `set`
$x := y$.`selectAndRemove`	$\mapsto$	$y_{temp} :=$ `malloc set`; $y_{temp} := y$; $y :=$ `malloc set`; $y := y_{temp}$; $x := y$.`selectAndRemove`	otherwise

Before altering sets, they are being copied to a new location. This is to remedy the effects of aliasing that may occur in Semantics II. When inserting a set A into another set B only its set location is inserted. Changing set A would then also alter set B, which is not desired and different from Semantics I.

Definition 18 (Corresponding Configurations) *Two configurations correspond, iff their statements and their states correspond:*

$$\langle S_1, (\sigma_1, \eta_1, \varsigma_1)\rangle \simeq \langle S_2, (\sigma_2, \eta_2, \varsigma_2)\rangle$$
$$:\Leftrightarrow$$
$$S_1 \sim S_2 \wedge (\sigma_1, \eta_1, \varsigma_1) \cong (\sigma_2, \eta_2, \varsigma_2)$$

Semantics II allows aliasing. In order to establish a simulation relation between the two semantics, we need to be able to eliminate aliasing where desired. The following lemma shows how this can be achieved.

Lemma 4 (Aliasing Lemma) *Let* $(\sigma, \eta, \varsigma) \cong (\sigma', \eta', \varsigma')$.
If $\langle x_{temp} := \texttt{malloc set}; x_{temp} := x; x := \texttt{malloc set}; x := x_{temp}; S', (\sigma', \eta', \varsigma')\rangle \triangleright'^{*} \langle S', (\sigma'_2, \eta'_2, \varsigma'_2)\rangle$ *then*

$$(\sigma, \eta, \varsigma) \cong (\sigma'_2, \eta'_2, \varsigma'_2)$$

and

$$\sigma'_2(x) \notin (im(\sigma'_2[x \mapsto undef.]) \cup im(\eta'_2) \cup \bigcup im(\varsigma'_2))$$

This means that executing the series of statements $x_{temp} := \texttt{malloc set}; x_{temp} := x;$ $x := \texttt{malloc set}; x := x_{temp}$ on a state $(\sigma', \eta', \varsigma')$ that is corresponding to $(\sigma, \eta, \varsigma)$ will preserve this correspondence, i.e. $(\sigma, \eta, \varsigma) \cong (\sigma'_2, \eta'_2, \varsigma'_2)$. In addition the set location of x is not aliased in the new state $(\sigma_2, \eta_2, \varsigma_2)$.

For a proof see Appendix A.

Lemma 5 (Simulation Lemma) *The* Simulation Lemma *expresses that Semantics II can mimic the behaviour of Semantics I.*

If $\langle S_1, (\sigma_1, \eta_1, \varsigma_1)\rangle \simeq \langle S'_1, (\sigma'_1, \eta'_1, \varsigma'_1)\rangle$ *and* $\langle S_1, (\sigma_1, \eta_1, \varsigma_1)\rangle \triangleright \langle S_2, (\sigma_2, \eta_2, \varsigma_2)\rangle$ *then there exists a* $\langle S'_2, (\sigma'_2, \eta'_2, \varsigma'_2)\rangle$ *with* $\langle S_2, (\sigma_2, \eta_2, \varsigma_2)\rangle \simeq \langle S'_2, (\sigma'_2, \eta'_2, \varsigma'_2)\rangle$ *and* $\langle S'_1, (\sigma'_1, \eta'_1, \varsigma'_1)\rangle \triangleright'^{*} \langle S'_2, (\sigma'_2, \eta'_2, \varsigma'_2)\rangle$

This is illustrated by the following diagram:

$$\begin{array}{ccc} \langle S_1, (\sigma_1, \eta_1, \varsigma_1)\rangle & \triangleright & \langle S_2, (\sigma_2, \eta_2, \varsigma_2)\rangle \\ \simeq & & \simeq \\ \langle S'_1, (\sigma'_1, \eta'_1, \varsigma'_1)\rangle & \triangleright'^{*} & \langle S'_2, (\sigma'_2, \eta'_2, \varsigma'_2)\rangle \end{array}$$

Any $\triangleright$−step in Semantics I can be simulated in Semantics II by making one or possibly more steps.

Proof Sketch:

For every inference rule in Semantics I show that there are inference rules in Semantics II with an equivalent effect. The applicability of most of the corresponding rules follows from the *Expressions Coincide Theorem.* Use the *Aliasing Lemma* to show that the execution of the commands preceding set operations remove aliases.
For a full proof see Appendix A. ■

We define sensible initial states for both semantics:

Definition 19 (Initial States) *An* initial state $(\sigma, \eta, \varsigma)$ *of the first semantics has the following properties:*

$\sigma(x)$	$=$	0	*if* x : `int`
$\sigma(x)$	$=$	$\mathbf{0}$	*if* x : `bool`
$\sigma(x)$	*undef.*		*if* x : `loc` *or* $x : t$ `set`
$\eta(\xi, sel)$	*undef.*		$\forall \xi \in \mathrm{Loc}.\forall sel \in \mathrm{Sel}$
$\varsigma(\psi)$	*undef.*		$\forall \psi \in \mathrm{SetLoc}$

An initial state $(\sigma', \eta', \varsigma')$ *in Semantics II is defined similarly, exchanging* $\sigma, \eta, \varsigma, \xi, \psi$ *with their respective primed versions.*

Building on the *Simulation Lemma* and the preceding definition we are now ready to proof the main theorem of this chapter.

Theorem 3 (Simulation Theorem) Initial states *of both semantics correspond and Semantics II can mimic the behaviour of Semantics I.*

If $(\sigma, \eta, \varsigma)$ *and* $(\sigma', \eta', \varsigma')$ *are initial states, then* $(\sigma, \eta, \varsigma) \cong (\sigma', \eta', \varsigma')$. *If* $\langle S_1, (\sigma_1, \eta_1, \varsigma_1)\rangle \simeq \langle S_1', (\sigma_1', \eta_1', \varsigma_1')\rangle$ *and* $\langle S_1, (\sigma_1, \eta_1, \varsigma_1)\rangle \triangleright \langle S_2, (\sigma_2, \eta_2, \varsigma_2)\rangle$ *then there exists a* $\langle S_2', (\sigma_2', \eta_2', \varsigma_2')\rangle$ *with* $\langle S_2, (\sigma_2, \eta_2, \varsigma_2)\rangle \simeq \langle S_2', (\sigma_2', \eta_2', \varsigma_2')\rangle$ *and* $\langle S_1', (\sigma_1', \eta_1', \varsigma_1')\rangle \triangleright'^* \langle S_2', (\sigma_2', \eta_2', \varsigma_2')\rangle$

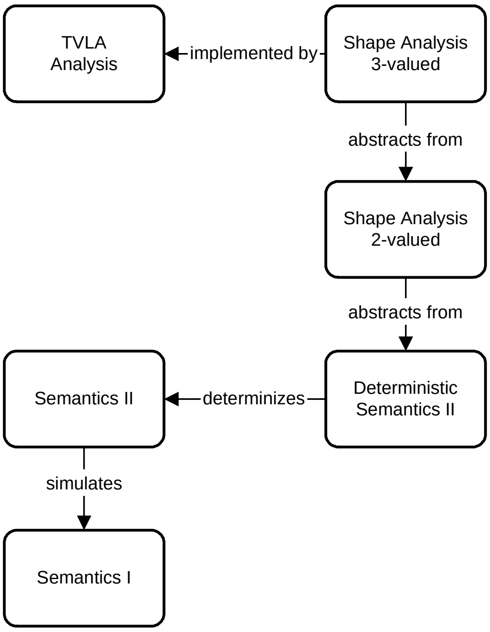
Proof:

The second part of the theorem is proven by the *Simulation Lemma*. It remains to show that initial states correspond. Obviously, a *Heap Correspondence Relation* $\equiv$ for (η, ς) and (η', ς') exists, since both η, η' and ς, ς' are completely undefined, thus putting no restrictions on $\equiv$. We also have $0 \equiv 0$ and $\mathbf{0} \equiv \mathbf{0}$ since this is required by the definition of a *Heap Correspondence Relation*. So, $\sigma(x) = 0 \equiv_{zz} 0 = \sigma'(x)$ for x : `int` and $\sigma(x) = \mathbf{0} \equiv_{bb} \mathbf{0} = \sigma'(x)$ for x : `bool`. Also, $\sigma(x)$ undef. and $\sigma'(x)$ undef. for x : `loc` or $x : t$ `set`.

■

6 Shape Analysis of RESET

To create a shape analysis for our Semantics II we perform two abstraction steps. The first step solely abstracts from values, i.e. booleans and integers. Only the structure of the heap (the shape) is preserved and represented by 2-valued logical structures. Using logical structures allows us to make another abstraction to a representation by 3-valued logical structures which makes our analysis computable since the domain will be finite. Our analysis fits into the framework of [SRW02] introduced in Chapter 2 and is implemented in TVLA. The following diagram illustrates the relations between the different semantics and our analysis:



6.1 Shape Analysis 2-valued

The 2-valued semantics abstracts from values, i.e. booleans and integers. It does however remain concrete for the structure of the heap and the set heap, since this is what we want to analyze in a Shape Analysis. The universe U comprises both locations in the heap and in the set heap. Alternatively, one could have two disjoint universes. In our case this would only complicate the definitions[1].

[1] We would need two predicates to represent the set heap, one for sets of locations and one for sets of sets, yielding even more predicate-update formulae.

Predicate	**Type**	**Intended Meaning**
$x(v)$ for each $x \in Var$	$U \to \mathbb{B}$	Pointer variable x points to heap cell v.
$sel(v_1, v_2)$ for each $sel \in Sel$	$U \times U \to \mathbb{B}$	The sel selector of v_1 points to v_2.
$isSet(v)$	$U \to \mathbb{B}$	v represents a set.
$isIn(v_1, v_2)$	$U \times U \to \mathbb{B}$	v_1 is in set v_2.
$leq(v_1, v_2)$	$U \times U \to \mathbb{B}$	An ordering relation on heap cells.
$selected(v)$	$U \times U \to \mathbb{B}$	v has been selected for removal or allocation.
$unallocated(v)$	$U \to \mathbb{B}$	v has not been allocated yet.

Figure 6.1: Representation of the State by Predicates

6.1.1 Domains

Unary and binary predicates are used to represent the state. These predicates are shown in Figure 6.1. For every variable a unary predicate is introduced that is true for the element of the heap or set heap it points to. This set of unary predicates corresponds to what was called the stack in the Semantics II. To distinguish between locations and set locations the unary predicate *isSet* is used. The heap is modeled by a collection of binary predicates, one for each selector. Such a predicate is true if the selector field of the first operand points to the second. Finally, the set heap is modeled by the binary predicate $isIn(v_1, v_2)$ which is true if v_1 is contained in v_2. Notice the subtle difference between the relation *isIn* and set membership. A set v can also be an element of a set v_{set} if it is equal to another set v_{el} such that $isIn(v_{el}, v_{set})$, but not $isIn(v, v_{set})$. Membership is determined by equality, not by identity. For sets of locations the two notions coincide, however, because equality on locations is identity.

Three more predicates are used, namely *leq*, *selected*, and *unallocated*, that are not directly connected to any of the constructs in the Semantics II. Previously, the $x := y$.`selectAndRemove`-statement was the source of nondeterminism. Our framework does not allow us to specify nondeterministic update formulae, though. To determinize this operation the predicate *leq* is introduced. It imposes a total ordering on the heap and the set heap. This allows us to deterministically select the “smallest” element of a set. The predicate *selected* will be explained in the section about the predicate-update formulae. Since we use a constant domain, all heap cells that will eventually be used have to be present right away. *unallocated* keeps track of the heap cells that have not yet been allocated or that have been released again.

Expression exp	**Type**	**Condition** $cond(exp)$
$x_1 = x_2$	$x_1, x_2 : \texttt{loc}$	$\forall v.(x_1(v) \Leftrightarrow x_2(v))$
`true`		$\mathbf{1}$
`false`		$\mathbf{0}$
$\neg exp_1$		$\neg cond(exp_1)$
$exp_1 \wedge exp_2$		$cond(exp_1) \wedge cond(exp_2)$
$exp_1 \vee exp_2$		$cond(exp_1) \vee cond(exp_2)$

Figure 6.2: Semantics of Expressions

6.1.2 Semantics of Expressions

The conditional statement and the while-loop require the evaluation of boolean expressions. Since we abstract from values, only expressions concerning locations or set locations have to be considered here. Equality of pointer variables is straightforward. It translates to equivalence of the two predicates representing the variables. Boolean combinations of expressions can also be handled easily. See Figure 6.2.

For sets, equality is more complicated. Two sets can be equal although they are stored at different locations in the set heap. In the Semantics II of the previous chapter this was handled by $\approx$, which recursively descended into the set heap. The following definition of predicate eq would be analogous.

$$\begin{array}{rcl} eq(v_1, v_2) & = & ((\forall v_3.isIn(v_3, v_1) \Rightarrow \exists v_4.(isIn(v_4, v_2) \wedge eq(v_3, v_4))) \wedge isSet(v_1) \\ & \wedge & (\forall v_4.isIn(v_4, v_2) \Rightarrow \exists v_3.(isIn(v_3, v_1) \wedge eq(v_3, v_4))) \wedge isSet(v_2)) \\ & \vee & (v_1 = v_2) \end{array}$$

Here however, we need to give an equivalent formula explicitly. It does not seem possible though for the general case of set equality using first-order logic. Fortunately, the maximal depth of sets can be determined statically by the type system. This allows us to expand the recursive definition sufficiently often. The $equal$-predicate is serving this purpose:

$$\begin{array}{rcl} equal_1(v_1, v_2) & = & ((\forall v_3.isIn(v_3, v_1) \Rightarrow \exists v_4.(isIn(v_4, v_2) \wedge v_3 = v_4)) \wedge isSet(v_1) \\ & \wedge & (\forall v_4.isIn(v_4, v_2) \Rightarrow \exists v_3.(isIn(v_3, v_1) \wedge v_3 = v_4)) \wedge isSet(v_2)) \\ & \vee & (v_1 = v_2) \\ equal_{n+1}(v_1, v_2) & = & ((\forall v_3.isIn(v_3, v_1) \Rightarrow \exists v_4.(isIn(v_4, v_2) \wedge equal_n(v_3, v_4))) \wedge isSet(v_1)) \\ & \wedge & (\forall v_4.isIn(v_4, v_2) \Rightarrow \exists v_3.(isIn(v_3, v_1) \wedge equal_n(v_3, v_4))) \wedge isSet(v_2)) \\ & \vee & (v_1 = v_2) \\ equal(v_1, v_2) & = & equal_n(v_1, v_2), \text{ where n is the maximal nesting depth of sets in the program.} \end{array}$$

The extra $(v_1 = v_2)$ in each definition of $equal_i$ acts as a shortcut. It allows $equal_i$ to be used for sets of depth i or less. Using $equal$ we can now go on to define the conditions for all the expressions concerning sets:

Expression *exp*	**Type**	**Condition** *cond*(*exp*)
$x_1 = x_2$	$x_1, x_2 : t$ `set`	$\exists v_1.\exists v_2.(x_1(v_1) \wedge x_2(v_2) \wedge equal(v_1, v_2))$
$x_1 \in x_2$		$\exists v_1.\exists v_2.(x_1(v_1) \wedge x_2(v_2) \wedge \exists v_{el}.(isIn(v_{el}, v_2) \wedge equal(v_1, v_{el})))$
$x_1 \subseteq x_2$		$\exists v_1.\exists v_2.(x_1(v_1) \wedge x_2(v_2) \wedge \forall v_{el1}.(isIn(v_{el1}, v_1)$ $\Rightarrow \exists v_{el2}.(isIn(v_{el2}, v_2) \wedge equal(v_{el1}, v_{el2}))))$

6.1.3 Semantics of Statements

The effect of statements is modeled by predicate-update formulae. These specify how the interpretation of the predicate symbols is altered by the execution of the particular statement. We only give update formulae for predicates that are changed by the respective statement, i.e. formulae that simply copy the previous interpretation are omitted.

The following table displays the predicate-update formulae for statements manipulating the heap. They are similar to those given in [SRW02].

Statement	**Type**	**Predicate-update formula**
$x := y$	$x :$ `loc`	$x'(v) = y(v)$
$x.sel := y$	$x.sel :$ `loc`	$sel'(v_1, v_2) = (sel(v_1, v_2) \wedge \neg x(v_1)) \vee (x(v_1) \wedge y(v_2))$
$x := y.sel$	$x :$ `loc`	$x'(v) = \exists v_1.(y(v_1) \wedge sel(v_1, v))$

We also split the handling of `malloc`-statements into two phases. In the first phase the heap cell to be allocated is selected. The *selected*-predicate is used to mark this heap cell. The smallest unallocated heap cell is chosen. The second update-formula is then responsible for assigning that heap cell and for removing it from the *unallocated*-predicate.

Statement	**Type**	**Predicate-update formula**
$x :=$ `malloc` or $x.sel :=$ `malloc` $- (1)$		$selected'(v) = unallocated(v)$ $\wedge \forall v_1.(unallocated(v_1) \Rightarrow leq(v, v_1))$
$x :=$ `malloc set` or $x.sel :=$ `malloc set` $- (1)$		$selected'(v) = isSet(v) \wedge unallocated(v)$ $\wedge \forall v_1.((isSet(v) \wedge unallocated(v_1)) \Rightarrow leq(v, v_1))$
$x :=$ `malloc` or $x :=$ `malloc set` $- (2)$		$unallocated'(v) = unallocated(v) \wedge \neg selected(v)$ $x'(v) = selected(v)$ $selected'(v) = \mathbf{0}$
$x.sel :=$ `malloc` or $x.sel :=$ `malloc set` $- (2)$		$unallocated'(v) = unallocated(v) \wedge \neg selected(v)$ $sel'(v_1, v_2) = (\neg x(v_1) \wedge sel(v_1, v_2))$ $\vee (x(v_1) \wedge selected(v_2))$ $selected'(v) = \mathbf{0}$

The following table shows the update formulae for statements manipulating sets. In these cases only the *isIn* predicate is changed. As in the Semantics II, assignments copy the contents of sets. Elements are inserted into a set if there is no other equal element already contained in it. When removing an element, all elements equal to it are removed from the set.

Statement	Type	Predicate-update formula
$x := y$	$x : t$ `set`	$isIn'(v_1, v_2) = \neg x(v_2) \wedge isIn(v_1, v_2)$ $\vee x(v_2) \wedge \exists v_3.(y(v_3) \wedge isIn(v_1, v_3))$
$x.sel := y$	$x.sel : t$ `set`	$isIn'(v_1, v_2) = \neg\exists v_3.(x(v_3) \wedge sel(v_3, v_2)) \wedge isIn(v_1, v_2)$ $\vee \exists v_3.(x(v_3) \wedge sel(v_3, v_2)) \wedge \exists v_4.(y(v_4) \wedge isIn(v_1, v_4))$
$x := y.sel$	$x : t$ `set`	$isIn'(v_1, v_2) = \neg x(v_2) \wedge isIn(v_1, v_2)$ $\vee x(v_2) \wedge \exists v_3.(y(v_3) \wedge \exists v_4.(sel(v_3, v_4) \wedge isIn(v_1, v_4)))$
$x.$`insert`(y)		$isIn'(v_1, v_2) = isIn(v_1, v_2)$ $\vee y(v_1) \wedge x(v_2) \wedge \neg\exists v_3.(isIn(v_3, v_2) \wedge equal(v_3, v_1))$
$x.$`remove`(y)		$isIn'(v_1, v_2) = isIn(v_1, v_2)$ $\wedge \neg(x(v_2) \wedge \exists v_3.(y(v_3) \wedge equal(v_1, v_3)))$

We split the $x := y.$`selectAndRemove`-statement into two steps to simplify the formulae. The first step selects the element while the second step actually removes it from y and assigns it to x. As mentioned before, we use the ordering relation *leq* to determinize the statement, by choosing the smallest element with respect to *leq*. This is done in the first step, by setting the *selected*-predicate to true for this element. Two different formulae are used for the second step depending on the type of set. When dealing with sets of locations we have to alter the stack. In the other case, only the set heap, represented by $isIn$, is changed. In both cases *selected* is reset to be universally false.

Statement	Type	Predicate-update formula
$x := y.$`selectAndRemove` $-(1)$		$selected'(v) = \exists v_{set}.(y(v_{set}) \wedge isIn(v, v_{set})$ $\wedge \forall v_{el}.(isIn(v_{el}, v_{set}) \Rightarrow leq(v, v_{el})))$
$x := y.$`selectAndRemove` $-(2)$	$x :$ `loc`	$x'(v) = selected(v)$ $isIn'(v_{el}, v_{set}) = isIn(v_{el}, v_{set}) \wedge \neg(y(v_{set}) \wedge selected(v_{el}))$ $selected'(v) = \mathbf{0}$
$x := y.$`selectAndRemove` $-(2)$	$x : t$ `set`	$isIn'(v_{el}, v_{set}) = (isIn(v_{el}, v_{set})$ $\vee (x(v_{set}) \wedge \exists v_{sel}.(selected(v_{sel}) \wedge isIn(v_{el}, v_{sel}))))$ $\wedge \neg(y(v_{set}) \wedge \exists v_{sel}.(selected(v_{sel}) \wedge equal(v_{el}, v_{sel})))$ $selected'(v) = \mathbf{0}$

6.2 Shape Analysis 3-valued

We could use the above predicates and predicate-update formulae directly to generate a 3-valued shape analysis using TVLA. In order to gain additional precision it is however necessary to add instrumentation predicates.

For our small case study we used three instrumentation predicates. See Figure 6.3 for their definition and intended meaning. We did not need to specify update formulae for these predicates. TVLA generated them automatically through finite differencing [RSL03]. This greatly reduced the burden on us. The $in[x]$ predicates are used as abstraction predicates. They serve a similar purpose as the $dle[x, sel]$ predicate family in Chapter 4 and may thus serve as a motivation for the *Abstraction Expressions* presented there.

Predicate	Defining Formula	Intended Meaning
$isElement(v_1, v_2)$	$\exists v.(isIn(v, v_2) \wedge equal(v_1, v))$	v_1 is an element of set v_2.
$isSubset(v_1, v_2))$	$isSet(v_1) \wedge isSet(v_2) \wedge \forall v.(isElement(v, v_1) \Rightarrow isElement(v, v_2))$	v_1 is a subset of v_2.
$in[x](v)$ for each $x \in Var$	$\exists v_1.(x(v_1) \wedge isElement(v, v_1)))$	v is an element of the set pointed to by x.

Figure 6.3: Instrumentation Predicates

For our case study these predicates were sufficient. Many other useful predicates are conceivable. For instance:

- Ternary predicates $isElementOfUnion, isElementOfIntersection$. Such predicates would be hard to visualize though.
- Alternatively, one could introduce unary predicates that represent the union or intersection of two specific sets. These could be visualized by additional nodes that are connecting the two sets to the elements.

6.3 Case Study - Intersection Program

To demonstrate that our shape analysis works in practice we conducted a case study. The task was to analyze the intersection program introduced in Chapter 5, which computes the intersection of two sets. Using the instrumentation predicates $in[X], in[Y]$ and $in[Z]$ we can formalize the property we want to prove in the following way:

$$\forall v.((in[X](v) \wedge in[Y](v)) \Leftrightarrow in[Z](v)).$$

An object v is a member of Z if and only if it is a member of both X and Y.
Figure 6.4 shows the source code of the program. The references to the contents of X are first copied into the temporary variable $Temp$. $Temp$ is used to iterate over the contents of X without destroying X itself. For every element of X we check whether it is also an element of Y. In this case it is inserted into Z.

We chose one three-valued input structure depicted in Figure 6.5 as input. An empty set pointed to by *empty* is kept to be able to check whether a set is empty by comparison. Unallocated sets are empty. The elements of X and Y are partitioned through the $in[x]$ predicates into those that are only contained in X, those only contained in Y and those contained in both X and Y.

Only one output structure is generated for this input. It is displayed in Figure 6.6. The only difference compared to the input structure is that elements that are contained in both

```
void intersection(Set X, Set Y)
{
    Temp := malloc set;
    Temp := X;
    Z := malloc set;
    while (Temp != Empty)
    {
        p := Temp.selectAndRemove;

        if (p ∈ Y)
            Z.insert(p);
    }

    p := NULL;
    Temp := NULL;
}
```

Figure 6.4: RESET Program Computing the Intersection

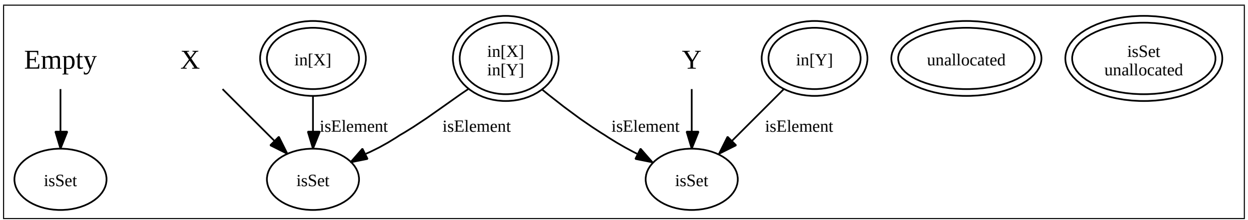


Figure 6.5: Input for Intersection Program

X and Y are now also contained in Z. So the program really computes the intersection of X and Y, the property is proven.

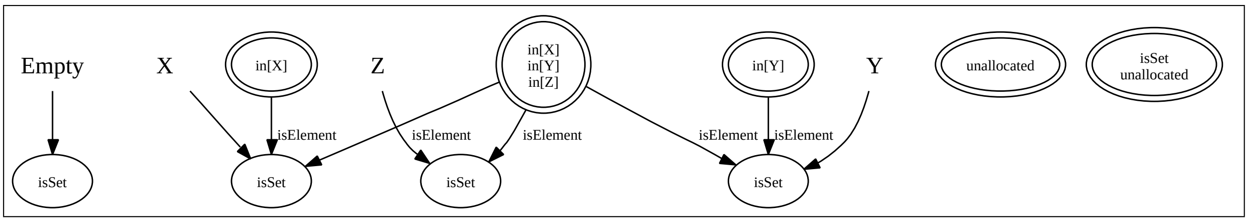


Figure 6.6: Output of Intersection Program

7 Modular Analysis

In this chapter we discuss modularity and modular analysis. We show its benefits, but also obstacles on the way to modularity. After describing techniques to overcome these obstacles we briefly investigate modular shape analysis.

7.1 Modularity

Modularity is an important concept in software engineering. Some of the advantages that a modular approach yields in the design process also translate to advantages of modular analyses. That is why it is useful to investigate what is meant by modularity in a more general sense before dealing with modular analysis.

According to Bertrand Meyer [Mey88] there are five essential criteria for a modular design method:

- Decomposability
  Large problems maybe decomposed into several less complex subproblems, connected by a simple structure, independent enough to be worked on concurrently.
- Composability
  Software elements can be composed to perform a desired task together. This is also related to reusability.
- Understandability
  A method favors modular understandability if it helps to produce modules that can be separately understood by a human reader.
- Continuity
  Small changes in the problem (specification) lead to small changes in the program in one or just few modules.
- Protection
  The effect of defects occurring at run-time remains confined to the module were it occurred. Errors should not propagate too far.

Meyer also names five rules for modularity that should be followed to achieve modularity as it is described above. We will list two of them that are related to modular analysis.

- Information Hiding
  Only a part of every module should be visible to the outside. This part is called the *interface*. Implementation details should be hidden. This rule follows primarily from the *Protection* aspect.

- Few Interfaces
  Every module should communicate with as few others as possible. Changes do not affect many other modules. This favors *Continuity* and *Protection*.

Object-oriented languages like Eiffel, C++, or Java allow to follow a modular design process. To a certain extent this is also possible in imperative languages.

7.2 Benefits of Modular Analysis

Modular analyses exploit the modular structure of programs to be analyzed. Figure 7.1 illustrates a possible modular structure of a program. The modules each follow some specification. A modular analysis would analyze each of the modules against its specification using only the specification of the modules it is using. This is depicted by the dashed arrows. An analysis of module C for instance, would prove its specification on the basis of the specifications of E and F. It would be independent of the concrete implementations of E and F.

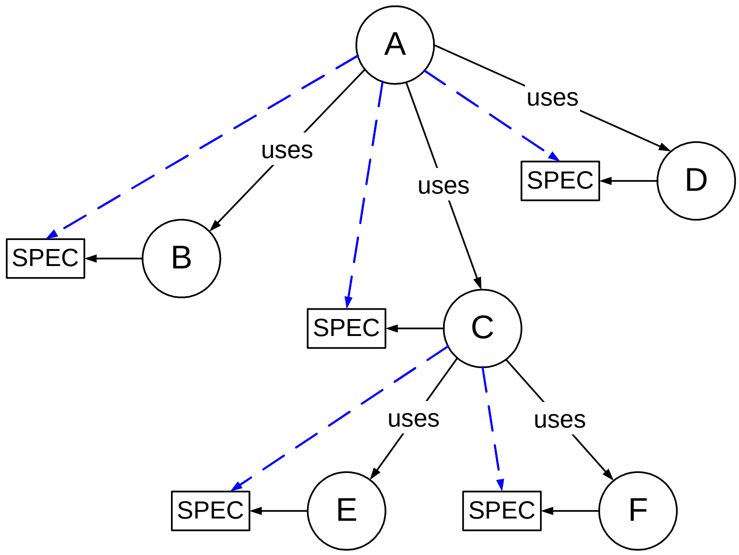


Figure 7.1: Sample Modular Structure

This separation into several analyses is related to the *Protection* and the *Understandability* criteria. Modules that are separately understandable by humans lend themselves naturally to an analysis following this structure. Errors in the implementation of modules that remain confined to the module where they occurred will likely be discovered in the analysis of the specific module, allowing the user to localize the problem. A modular analysis can also profit from the *Continuity* and the *Composability* aspects of modularity. A change in the implementation of one module only requires to reanalyze that particular module. The rest of the analysis remains valid as long as the specification does not change. Modules that are frequently reused have to be analyzed against their specification only once. This could be especially useful for widely used libraries.

In addition to these rather qualitative advantages, modular analyses are usually much faster than whole-program analyses. The simplicity of the specifications compared to their implementations results in smaller domains and thus earlier termination of the analysis algorithms.

7.3 Aliasing

Unfortunately, it is not always possible to perform modular analyses. Problems arise, where modules are not completely separated from each other. A modular view requires that changes to the state of a module can only be made by calls to the interface. This corresponds to the *Information Hiding* rule. However, this can usually not be guaranteed by the constructs of the programming language. When a memory location is reachable through different access paths, this is called aliasing. Aliasing allows to manipulate the heap at one place, causing problems at another. Figure 7.2 gives a simple example of this. x and y point to the same memory cell. If we manipulate $y.next$ this also has an impact on $x.next$. Aliasing only becomes a problem in conjunction with mutable locations.

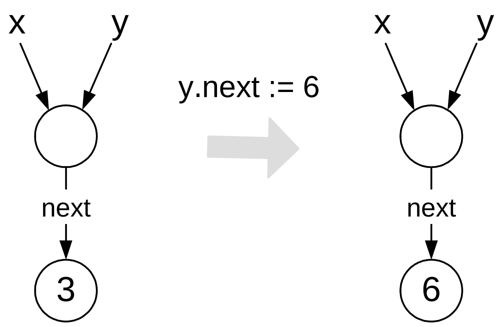


Figure 7.2: Simple Aliasing Example

If we have a notion of what is inside and what is outside of a module, we can distinguish two types of aliasing. Figure 7.3 gives an example involving a tree-based set module. The boundary of the set is marked by a rectangle, the interface by dots on the boundary. In this

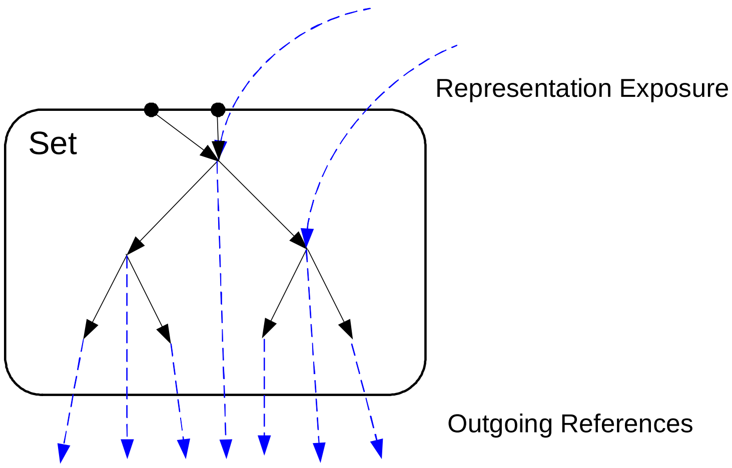


Figure 7.3: Representation Exposure and Outgoing References

example the data-elements of the set are considered to be outside. They are accessed by pointers from the tree structure. These pointers are called *Outgoing References*. Possibly, there are also paths leading into the module that are not passing through the interface. Such pointers, that are crossing the boundary from outside to inside are called *Incoming References*. Sometimes the inside of a module is called its *representation*. Then *Incoming References* are also referred to as *Representation Exposure* [Cla01].

Incoming references allow to manipulate the internal structure of the module. They could be used to destroy the tree structure or the ordering invariant. On the other hand such references may be useful. For instance, one might want to create an iterator. It would need to have read access to the internal structure to perform its task. Outgoing references are also problematic. In our example we could manipulate one of the data elements. This would in most cases violate the ordering invariant. Figure 7.4 shows the effect of manipulating set elements in a tree-based implementation (a), a list-based implementation (b) and in Semantics II (c) of Chapter 5. In all of the three cases set membership is changed and the data structure invariants are broken. The extent of the anomalies differs though:

- In the tree-based case the change of a data element can cause parts of the tree to become "invisible". A binary search in the tree would not discover the elements 7 and 8. The ordering invariant is broken. There are smaller elements than 11 in the right subtree.

- The list implementation is not as heavily damaged by changing one of its elements. In terms of set membership only the manipulated element is affected. In addition, the data structure invariant guaranteeing no duplicate elements is violated. This invariant could however be relaxed, for it is not necessary for a correct list-based implementation.

- Even if sets are primitives problems may arise. We are dealing with the Semantics II of Chapter 5. It is only possible to construct such a violation in the context of sets of sets though. This is because we are not able to manipulate primitive elements of sets. Two identical elements are created, both containing the 7 only (see Figure 7.4). In fact this corresponds exactly to the difference between the two semantics that we studied earlier.

The extent of the anomalies seems to be related to the complexity of the implementation. The more sophisticated the implementation, the greater the problems, as depicted in Figure 7.5. A modular analysis could follow two different approaches. It could either try to model these anomalies and internalize them, or it could rely on some mechanism to prevent the "bad" things from happening. We will continue to discuss the latter approach.

7.4 Ways to deal with Aliasing

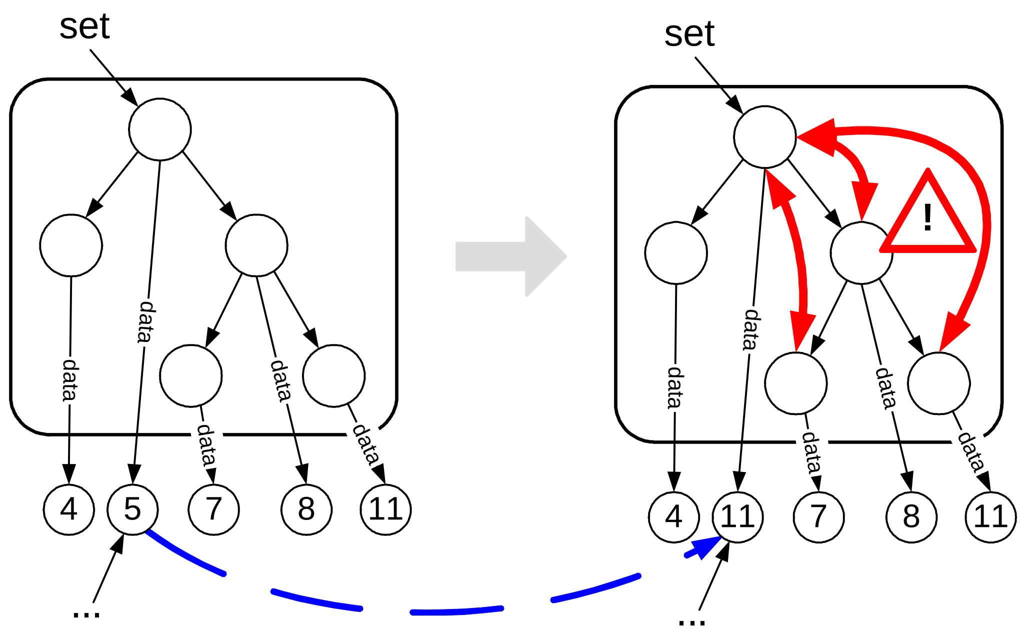
One possibility is to prevent aliasing altogether. This can be achieved by introducing constructs that guarantee static checkability by compilers and program analyzers. However, static checkability requires very conservative definitions. These constructs seem to impair the programmer too much [HLW$^+$92]. An example of such constructs is a swap statement that replaces normal assignment statements. In addition, many data structures rely on aliasing like doubly-linked lists.

Another rather basic approach is to allow aliases, but disallow the mutation of aliased objects. The transformation that we gave to generate equivalent programs in our set language is an example of alias prevention (Definition 17). Whenever we want to manipulate a set we would make a copy of it first and manipulate the copy instead.

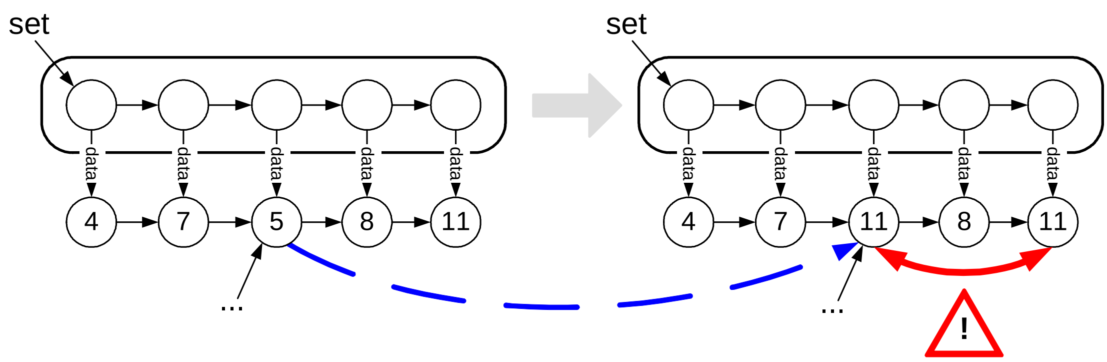
More sophisticated methods have been developed to limit aliasing in such a way that anomalies are prevented, at the same time being flexible enough to allow common programming patterns to be employed. Examples are *Islands* [Hog91], *Balloon Types* [Alm97], *Ownership Types* [CNP01, BLS03] and *Universes* [MPH01]. These methods all establish some sort of *encapsulation*. What is *encapsulation*?

"Encapsulation refers to building a capsule, in the case a conceptual barrier, around some collection of things." [WBWW90]

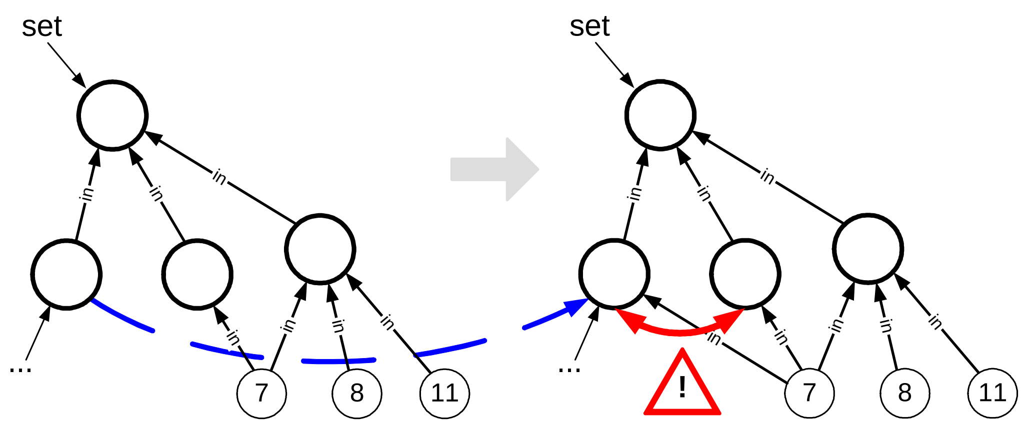
When talking about *incoming* and *outgoing* references we already had a notion of objects being inside or outside a module. This is formalized in different ways here. Constraints can then be imposed on references crossing the encapsulation boundaries. We may forbid write access or even read access to objects via access paths that cross the boundary. The difficulty is to provide a flexible yet statically checkable encapsulation scheme.



(a)



(b)



(c)

Figure 7.4: Anomalies through Outgoing References

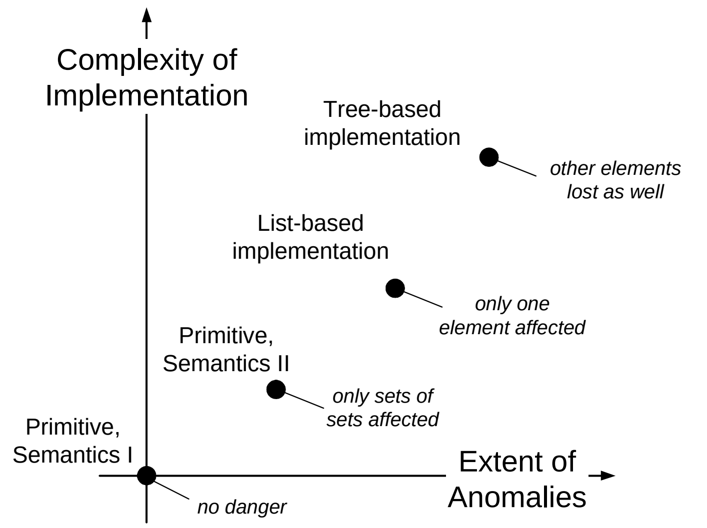
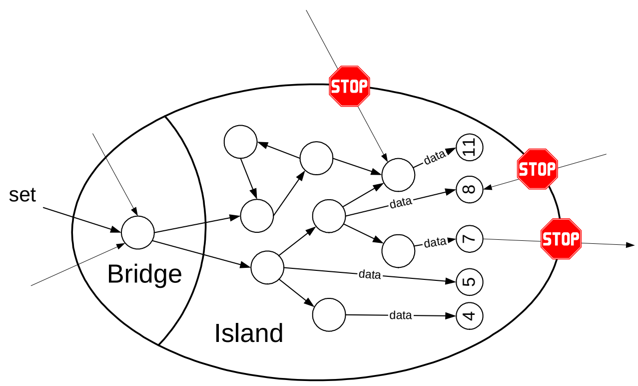


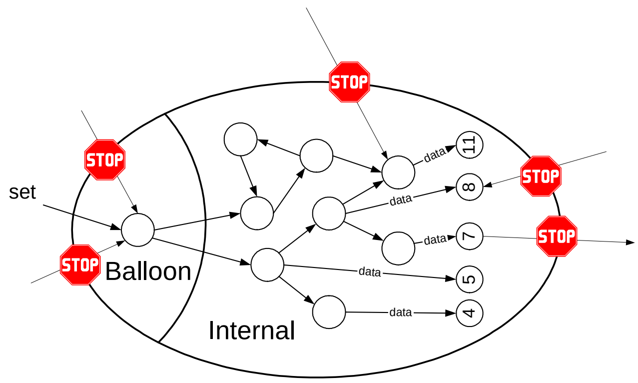
Figure 7.5: Extent of Anomalies

Noble et. al. [NBT+03] introduce a model of encapsulation to be able to compare different approaches. We give a short overview of the different approaches that partially stems from [NBT+03]. All of the attempts allow a nesting of encapsulation, so we will usually only discuss the basic case.

- *Islands* was the first such protection scheme. It provides *full encapsulation*, i.e. everything reachable from so-called *bridges* belongs to the island. References into the island that do not originate from the bridge are not allowed. References leaving the island are also prohibited. The island may only be accessed through its bridge. Aliasing is only allowed within the island.

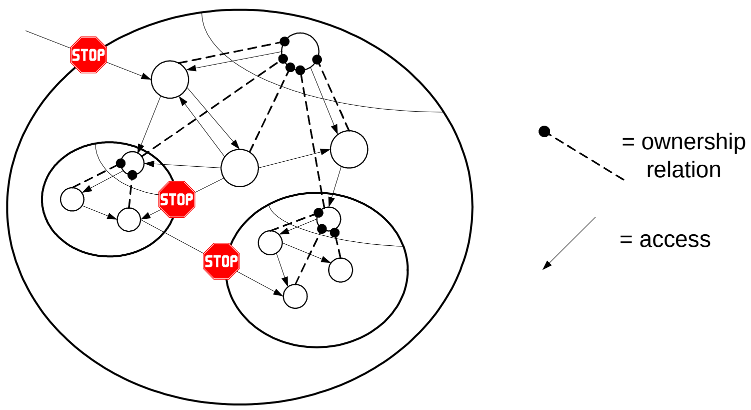


- *Balloons* is quite similar to *Islands*. It also provides full encapsulation. In contrast to *Islands* the entry to the balloon may not be aliased. The advantage of *Balloons* is that it needs less syntactic overhead than *Islands* to achieve encapsulation. It relies on an Abstract Interpretation to check whether the constraints imposed are met.



- *Ownership Types* do not necessarily enforce full encapsulation. An ownership relation *owner* between objects is established that forms a tree. Owners serve as entry points to the elements they encapsulate. In contrast to the full encapsulation schemes references may cross the boundaries from inside to outside. Entrance is still restricted to owners. Let **ownedby** be the transitive closure of the inverse of the *owner* relation. Then we can formalize this in the following way:

$$s \longrightarrow t \Rightarrow s \ \mathbf{ownedby}\ owner(t)$$



The additional flexibility gained by this is quite useful. For instance, it is now possible to differentiate between *arguments* and *representation*. Consider an unsorted linked-list. We are able to distinguish between elements stored in the list and the connecting structure. We can thus shield the structure, while keeping the elements available outside of it. For our set implementations this is not advisable as we have seen before. Our data structure invariants depend on the elements.

- *Universes* is similar to *Ownership Types*. It provides a little more flexibility by using read-only references. These may cross arbitrarily cross boundaries. Important programming patterns like iterators for existing containers can be created using read-only references. This was not possible with *Ownership Types*.

7.5 Modular Shape Analysis

We now want to informally explore how a modular shape analysis could look like and how our previous analyses relate to this.

An important question is how to express module specifications. On the one hand we need to be able to check that a module complies to its specification. On the other hand we want to use the specification as the basis for the analysis of programs that are using the module. It seems useful to specify the modules in the same language as the conventional shape analyses. This way we do not have to bridge an additional gap. In the shape analysis framework of [SRW02] first-order logic is used for this purpose. In this domain a module specification would consist of a number of predicates representing the state of the module and predicate-update formulae that model the effect of the module's methods. A disadvantage of this approach is that it is harder to write such specifications compared to algebraic specification

How would such a specification look like for a set module? In fact, the shape analysis developed in Chapter 6 contains predicates solely devoted to representing sets and predicate-update formulae to model the effect of the set methods. They may well serve as an example for a module specification. Using such a module specification in an analysis is easy. We replace the predicates used for the concrete implementation of the module by the predicates of the specification. Calls to module methods are interpreted by the predicate-update formulae of the specification instead of applying the methods of the implementation. In our set example an analysis can greatly benefit from this: The domain is reduced by using a smaller number of predicates. The effect of set methods can be computed by applying single predicate-update formulae. As we have seen in Chapter 4, a single invocation of the remove method could previously result in an analysis taking several hours.

Using a module specification requires that we have proven that the concrete implementation actually complies with it. For this purpose we have to somehow relate the domains of the implementation and the specification. Since both domains are specified using first-order logic, we define the predicates of the specification domain by formulae over the predicates used in the implementation. This resembles the definition of instrumentation predicates. We used the following two predicates to represent sets primitively (Chapter 6):

Predicate	**Type**	**Intended Meaning**
$isSet(v)$	$U \to \mathbb{B}$	v represents a set.
$isIn(v_1, v_2)$	$U \times U \to \mathbb{B}$	v_1 is in set v_2.

We can relate these predicates to the tree-based implementation like this:

Predicate	**Related Tree-based Expression**
$isSet(v)$	$isSet(v) \wedge treeNess \wedge inOrder$
$isIn(v_1, v_2)$	$downStar(v_1, v_2)$

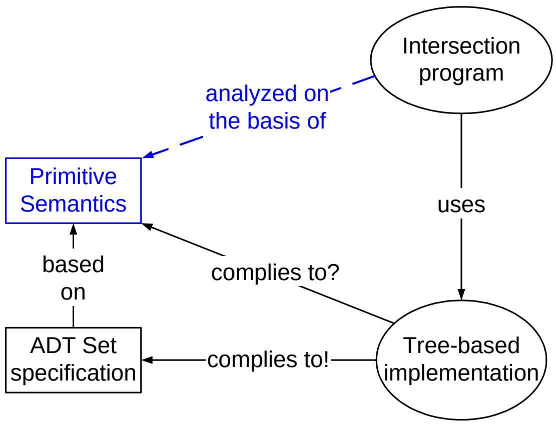


Figure 7.6: Does this constitute a sound Modular Shape Analysis?

where $treeNess$ and $inOrder$ capture the two data structure invariants for ordered trees and where $downStar(v_1, v_2)$ is defined as follows

Predicate	**Defining Formula**
$down(v_1, v_2)$	$left(v_1, v_2) \vee right(v_1, v_2)$
$downStar(v_1, v_2)$	$down^*(v_1, v_2)$

Interestingly, our TVLA analysis already used the instrumentation predicate $downStar$ which directly corresponds to $isIn$. After relating the domains we have to prove that the effect of the set methods in the implementation and the specification on related structures results in related structures again. In addition, we have to show that other operations cannot effect the structures in the implementation. We believe that these tasks should be dealt with separately. The former task could possibly be performed by a shape analysis similar to the one described in Chapter 4. The latter could be taken care of by employing an alias protection scheme like *Islands*. As we have seen in Figure 7.5 the primitive semantics was affected differently by aliasing than the implementations. So such a protection scheme is necessary.

Let us summarize how the analyses in Chapters 4 and 6 fit in here. In Chapter 4 we partially proved the conformance of list- and tree-based set implementations to the specification of the ADT Set defined in Chapter 3. Later we defined a semantics of an imperative language that contains sets as primitives. The definition was specifically designed to conform with the ADT Set specification. In Chapter 6 we created a shape analysis for this semantics and successfully applied it to a simple program. Although we took the detour via the ADT Set specification the resulting shape analysis seems to constitute a specification of the set implementations. We have not proven this though. Figure 7.6 illustrates this.

7.6 Assume/Guarantee Reasoning

While the approaches above try to prevent the negative effects of aliasing, another possibility would be to internalize these effects into the analysis. Such analyses are not really modular, but they might still help to make shape analysis algorithms scale better and they put less of a burden on the programmer to enter specifications.

In *Assume/Guarantee Reasoning* [YRS04, YSRS05] the effect of procedures on the heap is symbolically characterized. The programmer has to provide the precondition of procedures. Through abstract interpretation and the use of a theorem prover a precise symbolic representation of the effect of the procedure is then inferred. Inputs and outputs are connected in such a formula by using primed and unprimed versions of the predicates. When analyzing a program that uses such a procedure, the validity of the precondition is checked first. Then the precomputed effect is applied on the current state. Both steps involve the use of theorem provers. This limits the application to decidable logics. By staying in the domain of the implementation aliasing effects can be modeled without problems.

8 Conclusion

In this chapter we want to briefly recapitulate our contributions and discuss possible future work on that basis.

8.1 Contributions

We created a precise shape analysis for programs that are manipulating ordered trees. It is particularly tailored to invariants of the tree data structure. Choosing the right instrumentation predicates required a thorough understanding of the data structures involved. This meant identifying that reachability alone is not very interesting, but that the first edge on a path from one node to another is important. We implemented the analysis in TVLA [LA00, LAS00] and successfully applied it to methods of the tree-based set implementation. The analysis proved that the implementation complies to the axioms (3) and (4) of the ADT Set specification.

$$a \in s.\texttt{insert}(b) \leftrightarrow a =_{el} b \vee a \in s, \quad (3)$$
$$a \in s.\texttt{remove}(b) \leftrightarrow a \neq_{el} b \wedge a \in s \quad (4)$$

We used the *isElement*-predicate to relate different analyses. Our analyses of the insertion and removal methods established the two axioms in terms of *isElement*. Another analysis then established the equivalence between *isElement* and the set membership method $\cdot$`.insert(`$\cdot$`)`. Adapting existing analyses for singly-linked lists allowed us to show the same property for our list-based set implementation.

Inspired by a family of instrumentation predicates used in our tree analysis, we propose a new way of specifying abstractions by so-called "Abstraction Expressions". These expressions allow to not only use unary but also binary predicates in the abstraction specification. "Abstraction Expressions" have the same expressive power as *Canonical Abstraction*. However, we need a smaller number of predicates to express certain abstractions.

We also investigated the relation between the complexity of the domains of implementations and the extent of anomalies caused by aliasing. We found that the extent of anomalies rises with the complexity of the domains. Figure 8.1 illustrates this. Even the Semantics II of our RESET language shows some aliasing problems, although its domain is very simple. Problems occur only with sets of sets though.

We formally related Semantics I and Semantics II to identify where problems occur exactly. Relating the two semantics also hints at one way of overcoming aliasing problems, a technique known as *alias prevention*.

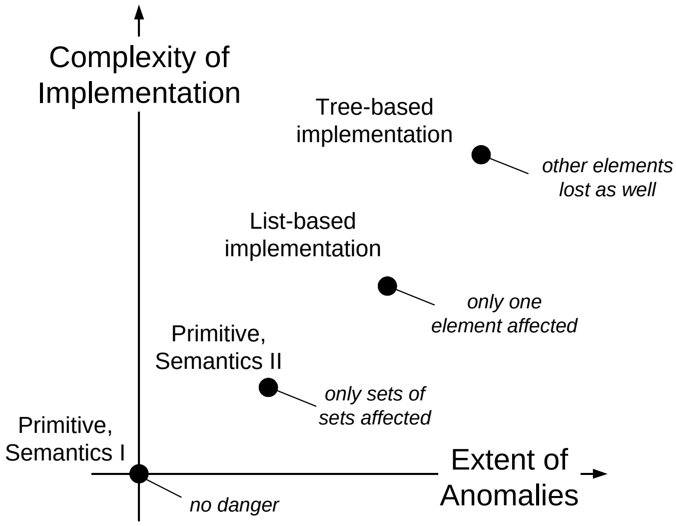


Figure 8.1: Extent of Anomalies

8.2 Future Work

In Chapter 4, we successfully analyzed a tree-based set implementation. Since the analysis is tailored to the underlying data structure and not to the specific algorithms employed, it might be possible to analyze other algorithms working on trees using the same abstraction.

The tree structure lends itself naturally to recursion. We could possibly combine recent work on interprocedural shape analysis [RS01] with our abstractions to be able to analyze recursive implementations. Modern data structure libraries usually contain more efficient set implementations using balanced trees, like AVL or red-black trees. They maintain even more complicated data structure invariants than the unbalanced tree implementation we analyzed. Algorithms on these structures can usually be implemented more easily using recursion, too. Extending our analysis to cope with the invariants of balanced trees might make such algorithms amenable as well.

Abstraction Expressions seem useful where we want to distinguish individuals if they differ by binary predicates originating from individuals that we distinguish. In our tree-based analysis, we could separate smaller and larger tree elements. In the shape analysis for RESET, we could use the set membership relation to separate individuals in terms of the sets they belong to. An implementation of the concept would allow deeper insight into the usefulness of the approach.

In Chapter 7 we discussed modular analysis. Aliasing was identified as an obstacle on the way to modular analyses. Different encapsulation schemes were briefly introduced that limit aliasing. It would be interesting to investigate such protection mechanism even further. How much do the constructs of the scheme constrain our design? Do the guarantees given by the encapsulation scheme suffice to perform sound modular shape analyses?

Modular Analysis requires module specifications. We started out using algebraic specification to specify the ADT Set. The technique proved convenient as a specification mechanism. However, formally relating implementations to algebraic specifications is not so easy. It is also not obvious how to base a shape analysis on such a specification. In Chapter 7 we proposed to stay in the first-order logic domain for the specification. This makes it easier to analyze programs on the basis of the specification. Figure 8.2 illustrates the situation. Can other specification techniques better cover the triangle? Can we possibly transform algebraic specifications into the domain of the analyses? Maybe we can automatically generate obligations for shape analysis that are necessary to show the compliance of implementations to algebraic specifications.

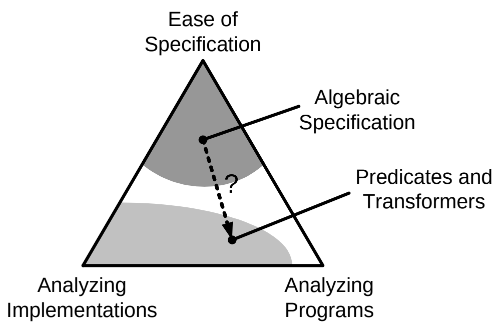


Figure 8.2: Properties of Specifications for Modular Analysis

List of Figures

Bibliography

[Alm97] Paulo Sérgio Almeida. Balloon types: Controlling sharing of state in data types. In *Proceedings ECOOP'97*, volume 1241 of *LNCS*, pages 32–59. Springer-Verlag, June 1997.

[BLS03] Chandrasekhar Boyapati, Barbara Liskov, and Liuba Shrira. Ownership types for object encapsulation. In *ACM Symposium on Principles of Programming Languages (POPL)*, New Orleans, Louisiana, January 2003.

[BRS+00] Michael Balser, Wolfgang Reif, Gerhard Schellhorn, Kurt Stenzel, and Andreas Thums. *A Practical Course on KIV*, 2000.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

[Cla01] David Clarke. *Object Ownership and Containment*. PhD thesis, University of New South Wales, Australia, July 2001.

[CNP01] David G. Clarke, James Noble, and John Potter. Simple ownership types for object containment. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 53–76, London, UK, 2001. Springer-Verlag.

[CWZ90] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *PLDI '90: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 296–310, New York, NY, USA, 1990. ACM Press.

[DF04] Robert DeLine and Manuel Fahndrich. Typestates for objects, 2004.

[Ebb94] Heinz-Dieter Ebbinghaus. *Einführung in die Mengenlehre*. BI Wissenschaftsverlag, 3., vollst. überarb. und erweiterte auflage edition, 1994.

[EM85] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification I*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1985.

[EM90] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification 2: Module Specifications and Constraints*. Springer-Verlag New York, Inc., New York, NY, USA, 1990.

[GH96] Rakesh Ghiya and Laurie J. Hendren. Is it a tree, a dag, or a cyclic graph? a shape analysis for heap-directed pointers in c. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–15, New York, NY, USA, 1996. ACM Press.

[HLW+92] John Hogg, Doug Lea, Alan Wills, Dennis deChampeaux, and Richard Holt. The geneva convention on the treatment of object aliasing. *SIGPLAN OOPS Mess.*, 3(2):11–16, 1992.

[Hog91] John Hogg. Islands: aliasing protection in object-oriented languages. In *OOPSLA '91: Conference proceedings on Object-oriented programming systems, languages, and applications*, pages 271–285, New York, NY, USA, 1991. ACM Press.

[LA00] Tal Lev-Ami. TVLA: A framework for kleene based static analysis. Master's thesis, Tel-Aviv University, Tel-Aviv, Israel, 2000.

[LARSW00] Tal Lev-Ami, Thomas Reps, Mooly Sagiv, and Reinhard Wilhelm. Putting static analysis to work for verification: A case study. In *ISSTA '00: Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 26–38, New York, NY, USA, 2000. ACM Press.

[LAS00] Tal Lev-Ami and Mooly Sagiv. TVLA: A system for implementing static analyses. In *SAS '00: Proceedings of the 7th International Symposium on Static Analysis*, pages 280–301, London, UK, 2000. Springer-Verlag.

[LEW97] Jacques Loeckx, Hans-Dieter Ehrich, and Markus Wolf. *Specification of abstract data types*. John Wiley & Sons, Inc., New York, NY, USA, 1997.

[LKR04] Patrick Lam, Viktor Kuncak, and Martin Rinard. Generalized typestate checking using set interfaces and pluggable analyses, 2004.

[Man03] Roman Manevich. Data structures and algorithms for efficient shape analysis. Master's thesis, Tel-Aviv University, School of Computer Science, Tel-Aviv, Israel, January 2003. Available at www.cs.tau.ac.il/rumster/msc_thesis.pdf.

[Mey88] Bertrand Meyer. *Object-Oriented Software Construction*, volume - of *Prentice Hall International Series in Computer Science*. Prentice Hall, New York - London - Toronto, 1988.

[Mic04] Sun Microsystems. Java 2 platform standard edition 5.0 api specification, 2004. Available at http://java.sun.com/j2se/1.5.0/docs/api/.

[MN99] Kurt Mehlhorn and Stefan Näher. *LEDA - A Platform for Combinatorial and Geometric Computing.* Cambridge University Press, Cambridge, 1999.

[MPH01] Peter Müller and Arnd Poetzsch-Heffter. Universes: A type system for alias and dependency control. Technical Report 279, Fernuniversität Hagen, 2001.

[MS96] David R. Musser and Atul Saini. *STL tutorial and reference guide*, volume - of *Addison-Wesley professional computing ser.* Addison-Wesley, 1996.

[NBT+03] James Noble, Robert Biddle, Ewan Tempero, Alex Potanin, and Dave Clarke. Towards a model of encapsulation. In *International Workshop on Aliasing, Confinement, and Ownership (IWACO)*, Darmstadt, Germany, July 2003.

[NN92] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: A Formal Introduction.* John Wiley & Sons, Inc., New York, NY, USA, 1992.

[NNH99] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[NVP98] James Noble, Jan Vitek, and John Potter. Flexible alias protection. In *ECOOP '98: Proceedings of the 12th European Conference on Object-Oriented Programming*, pages 158–185, London, UK, 1998. Springer-Verlag.

[Obe94] Arnold Oberschelp. *Allgemeine Mengenlehre.* BI Wissenschaftsverlag, 1994.

[RS01] Noam Rinetzky and Mooly Sagiv. Interprocedural shape analysis for recursive programs. *Lecture Notes in Computer Science*, 2027:133–149, 2001.

[RSL03] Thomas Reps, Shmuel Sagiv, and Alexey Loginov. Finite differencing of logical formulas for static analysis. In *ESOP*, pages 380–398, 2003.

[SRW99] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3–valued logic. In *Symposium on Principles of Programming Languages*, pages 105–118, 1999.

[SRW02] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3–valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, 2002.

[Wac05] Björn Wachter. Checking universally quantified temporal properties with three-valued analysis. Master's thesis, Universität des Saarlandes, March 2005.

[WBWW90] Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener. *Designing Object-Oriented Software.* Prentice Hall, Englewood Cliffs, New Jersey, 1990.

[Win93] Glynn Winskel. *The Formal Semantics of Programming Languages – An Introduction.* MIT Press, Cambridge, MA, USA, 1993.

[YRS04] Greta Yorsh, Thomas W. Reps, and Shmuel Sagiv. Symbolically computing most-precise abstract operations for shape analysis. In *TACAS*, pages 530–545, 2004.

[YSRS05] Greta Yorsh, Alexey Skidanov, Thomas Reps, and Mooly Sagiv. Automatic assume/guarantee reasoning for heap-manipulating programs (ongoing work), 2005.

A Proofs

Lemma 1:

Proof:

We will prove this constructively: $\equiv = \equiv' \setminus (\{\psi\} \times \mathit{SetLoc}') \cup \{(\psi, s') \mid (\forall i \in x.\exists i' \in x'.i \equiv i' \wedge \forall i' \in s'.\exists i \in x.i \equiv i')\}$. Since $\psi \notin (im(\sigma) \cup im(\eta) \cup dom(\varsigma))$, ψ does not occur in any of the requirements for $\equiv'_{ll}, \equiv'_{ssl}, \equiv'_{ss}$. This means that all elements of $\equiv_{ll}, \equiv_{ssl}$ and $\equiv_{ss}$ remain correct after changing ς. $\equiv'_{bb}$ and $\equiv'_{zz}$ are the same in any *Heap Correspondence Relation*, so we are left with $\equiv'_{slsl}$. The pairs with ψ in the first component have to be adjusted. This is achieved by the removal of all existing pairs and the addition of pairs according to the definition. No pairs regarding i or i' are removed either, proving the second part of the conjunction.

The proof for $\equiv''$ is analogous.

■

Lemma 2:

Proof:

Proof by induction over the type of set that i, j and i' represent.

- Base case: i, j, i' represent sets of primitive values $(\mathbb{B}, \mathbb{Z})$
  The definition of $\equiv$ can be simplified to plain set equality in this case.
  $i \equiv i' \Rightarrow i \equiv_{ss} i' \Rightarrow i = i'$ and $j \equiv i' \Rightarrow j \equiv_{ss} i' \Rightarrow j = i'$.
  So $i = j$.
- Base case: i, j, i' represent sets of locations.
  We prove $i \subseteq j$ and $j \subseteq i$:
  - $e \in i \Rightarrow \exists e' \in i'.e \equiv_{ll} e'$
    $e' \in i' \Rightarrow \exists e'' \in j.e'' \equiv_{ll} e'$
    Since $\equiv_{ll}$ is by definition injective, $e = e''$.
  - Completely analogous exchanging i and j.
- Step case: i, j, i' represent sets of sets.
  Again, we prove $i \subseteq j$ and $j \subseteq i$:

- $e \in i \Rightarrow \exists e' \in i'.e \equiv_{ssl} e' \Rightarrow e \equiv_{ss} \varsigma'(e')$
  $e' \in i \Rightarrow \exists e'' \in j.e'' \equiv_{ssl} e' \Rightarrow e'' \equiv_{ss} \varsigma'(e')$
  By induction hypothesis $e = e''$, so $e \in j$.
- Completely analogous exchanging i and j.

■

Lemma 3:

Proof:

1. - Case 1: $i, i', j' \in \mathbb{B}$
     $i \equiv i' \Rightarrow i \equiv_{bb} i' \Rightarrow (i \Leftrightarrow i')$
     $i \equiv j' \Rightarrow i \equiv_{bb} j' \Rightarrow (i \Leftrightarrow j')$
     $((i \Leftrightarrow i') \wedge (i \Leftrightarrow j')) \Rightarrow (i' \Leftrightarrow j') \Rightarrow (i' \approx j')(\sigma', \eta', \varsigma')$
   - Case 2: $i, i', j' \in \mathbb{Z}$
     Analogous to previous case.
   - Case 3: $i \in Loc, i', j' \in Loc'$
     $i \equiv i' \Rightarrow i \equiv_{ll} i' \Rightarrow \forall i''.(i \equiv_{ll} i'' \Rightarrow i' = i'')$
     $i \equiv j' \Rightarrow i \equiv_{ll} j' \Rightarrow i' = j' \Rightarrow (i' \approx j')(\sigma', \eta', \varsigma')$
   - Case 4: $i \in Set, i', j' \in SetLoc'$
     Proof by induction over the type of set represented by i.
     - Base case: i represents a set of locations or of primitive values
       It is sufficient to show that $\varsigma'(i') = \varsigma'(j')$, because this entails $(i' \approx j')(\sigma', \eta', \varsigma')$.
       * $\varsigma'(i') \subseteq \varsigma'(j')$:
         $e' \in \varsigma'(i') \Rightarrow \exists e \in i.e \equiv e'$
         $e \in i \Rightarrow \exists e'' \in \varsigma'(j').e \equiv e''$
         For $\mathbb{B}, \mathbb{Z}$ and Loc $\equiv$ is functional, that is e is related to at most one element, so $e' = e''$ and $(e' \approx e'')(\sigma', \eta', \varsigma')$.
       * $\varsigma'(j') \subseteq \varsigma'(i')$:
         Analogous to previous case.
     - Step case: i is a set of sets.
       We have to prove $(i' \approx j')(\sigma', \eta', \varsigma') = (\forall x \in \varsigma'(i').\exists z \in \varsigma'(j').(z \approx x)(\sigma', \eta', \varsigma')) \wedge (\forall x \in \varsigma'(j').\exists z \in \varsigma'(i').(z \approx x)(\sigma', \eta', \varsigma'))$. We will first prove the first part of the conjunction. The second part is analogous.
       * $\forall x \in \varsigma'(i').\exists z \in \varsigma'(j').(z \approx x)(\sigma', \eta', \varsigma')$
         $i \equiv i' \Rightarrow (e' \in \varsigma'(i') \Rightarrow \exists e \in i.e \equiv_{ssl} e')$
         $i \equiv j' \Rightarrow (e \in i \Rightarrow \exists e'' \in \varsigma'(j').e \equiv_{ssl} e'')$
         By induction hypothesis we can infer $(e' \approx e'')(\sigma', \eta', \varsigma')$. Since $e'' \in j'$ we have found a corresponding element for e'.
       * $\forall x \in \varsigma'(j').\exists z \in \varsigma'(i').(z \approx x)(\sigma', \eta', \varsigma')$
         Analogous to first part.

- Case 5: $i \in Set, i', j' \in Set'$
  We have to prove $(\forall e' \in i'.\exists e'' \in j'.(e' \approx e'')(\sigma', \eta', \varsigma')) \wedge (\forall e'' \in j'.\exists e' \in i'.(e' \approx e'')(\sigma', \eta', \varsigma'))$. We will separately prove the two parts of the conjunction.
  - $\forall e' \in i'.\exists e'' \in j'.e' \equiv e''$
    $e' \in i' \Rightarrow \exists e \in i.e \equiv e'$ and
    $e' \in i \Rightarrow \exists e'' \in j'.e \equiv e''$.
    Using the previous cases we can infer $(e' \approx e'')(\sigma', \eta', \varsigma')$.
  - $\forall e'' \in j'.\exists e' \in i'.e' \equiv e''$
    Analogous to previous case.
- Case 6: $i \in SetLoc, i', j' \in SetLoc'$
  $i \equiv_{slsl} i' \Rightarrow \varsigma(i) \equiv_{ss} \varsigma'(i')$ and
  $i \equiv_{slsl} j' \Rightarrow \varsigma(i) \equiv_{ss} \varsigma'(j')$
  By case 5 we can infer $(\varsigma'(i') \approx \varsigma'(j'))(\sigma', \eta', \varsigma')$. By definition of $\approx$ this is equivalent to $(i' \approx j')(\sigma', \eta', \varsigma')$.

2.
- Case 1: $i, i', j' \in \mathbb{B}$
  $i \equiv i' \Rightarrow (i \Leftrightarrow i')$
  $(i' \approx j')(\sigma', \eta', \varsigma') \Rightarrow (i' \Leftrightarrow j')$
  $((i \Leftrightarrow i') \wedge (i' \Leftrightarrow j')) \Rightarrow (i \Leftrightarrow j') \Rightarrow (i \equiv j'')$
- Case 2: $i, i', j' \in \mathbb{Z}$
  Analogous to previous case.
- Case 3: $i \in Loc, i', j' \in Loc'$
  $(i' \approx j')(\sigma', \eta', \varsigma') \Rightarrow i' = j'$
  $(i \equiv i' \wedge i' = j') \Rightarrow (i \equiv j')$
- Case 4: $i \in Set, i', j' \in SetLoc'$
  Proof by induction over the type of set represented by i.
  - Base case: i represents a set of locations or of primitive values
    Here, $\approx$ simplifies to set equality of the referenced sets.
    $(\varsigma'(i') \approx \varsigma'(j'))(\sigma', \eta', \varsigma') \Rightarrow \varsigma'(i') = \varsigma'(j')$. $i \equiv_{ssl} i'$ is equivalent to $i \equiv_{ss} \varsigma'(i')$. By the previous equality we get $i \equiv_{ss} \varsigma(j')$ which is again equivalent to $i \equiv_{ssl} j'$.
  - Step case: i represents a set of sets.
    We have to prove $i \equiv j' \Leftrightarrow i \equiv_{ss} \varsigma'(j') \Leftrightarrow (\forall e \in i.\exists e'' \in \varsigma'(j').e \equiv e'' \wedge \forall e'' \in \varsigma'(j').\exists e \in i.e \equiv e'')$ We will separately prove the two parts of the conjunction.
    * $\forall e \in i.\exists e'' \in \varsigma'(j').e \equiv e''$
      $e \in i \Rightarrow \exists e' \in \varsigma'(i').e \equiv_{ssl} e'$ and
      $e' \in \varsigma'(i') \Rightarrow \exists e'' \in \varsigma'(j').(e' \approx e'')(\sigma', \eta', \varsigma')$.
      By induction hypothesis we can infer $e \equiv e''$.
    * $\forall e'' \in \varsigma'(j').\exists e \in i.e \equiv e''$
      Analogous to previous case.

- Case 5: $i \in \mathit{Set}, i', j' \in \mathit{Set}'$
  We have to prove $i \equiv j' \Leftrightarrow (\forall i \in s.\exists i' \in s'.i \equiv i' \wedge \forall i' \in s'.\exists i \in s.i \equiv i')$.
  We will separately prove the two parts of the conjunction.
  - $\forall e \in i.\exists e'' \in j'.e \equiv e''$
    $e \in i \Rightarrow \exists e' \in i'.e \equiv e'$ and
    $e' \in i' \Rightarrow \exists e'' \in j'.(e' \approx e'')(\sigma', \eta', \varsigma')$.
    Using the previous cases we can infer $e \equiv e''$.
  - $\forall e'' \in j'.\exists e \in i.e \equiv e''$
    Analogous to previous case.
- Case 6: $i \in \mathit{SetLoc}, i', j' \in \mathit{SetLoc}'$
  $i \equiv_{slsl} i' \Rightarrow \varsigma(i) \equiv_{ss} \varsigma'(i')$ and
  $(i' \approx j')(\sigma', \eta', \varsigma') \Rightarrow (\varsigma'(i') \approx \varsigma'(j'))(\sigma', \eta', \varsigma')$.
  By case 5 we can follow $\varsigma(i) \equiv_{ss} \varsigma'(j')$ which implies $i \equiv_{slsl} j'$.

■

Theorem 2:

Proof:

Proof by induction over the structure of the formula.

- Base cases: $s = Num$ and $s = true$ and $s = false$
  Trivial.
- Base case: $s = x$
  By definition $\sigma(x) \equiv \sigma'(x)$ or $\sigma(x)$ and $\sigma'(x)$ are undefined.
  In the former case $\mathcal{X}[\![x]\!](\sigma, \eta, \varsigma) = \sigma(x) \equiv \sigma'(x) = \mathcal{X}'[\![x]\!](\sigma', \eta', \varsigma')$. If both values are undefined our condition is also fulfilled.
- Base case: $s = x.sel$
  Again $\sigma(x)$ and $\sigma'(x)$ maybe undefined. Then the condition is trivially true. Otherwise, by definition $\sigma(x) \equiv \sigma'(x)$ implies $\sigma(x) \equiv_{ll} \sigma'(x)$. This implies $\eta(\sigma(x), sel) \equiv \eta'(\sigma'(x), sel) \vee (\eta(\sigma(x), sel)$ undef. $\wedge\, \eta'(\sigma'(x), sel)$ undef.).

  - Case 1: $\eta(\sigma(x), sel)$ defined: $\mathcal{X}[\![x.sel]\!](\sigma, \eta, \varsigma) = \mathcal{P}[\![x.sel]\!](\sigma, \eta, \varsigma) = \eta(\sigma(x), sel) = \eta'(\sigma'(x), sel) = \mathcal{P}'[\![x.sel]\!](\sigma', \eta', \varsigma') = \mathcal{X}'[\![x.sel]\!](\sigma', \eta', \varsigma')$
  - Case 2: $\eta(\sigma(x), sel)$ undef.: $\mathcal{X}[\![x.sel]\!](\sigma, \eta, \varsigma) = \mathcal{P}[\![x.sel]\!](\sigma, \eta, \varsigma) =$ undef. $=$ undef. $= \mathcal{P}'[\![x.sel]\!](\sigma', \eta', \varsigma') = \mathcal{X}'[\![x.sel]\!](\sigma', \eta', \varsigma')$

By the previous two cases we know that either $\mathcal{P}[\![q]\!](\sigma, \eta, \varsigma) \equiv \mathcal{P}'[\![q]\!](\sigma', \eta', \varsigma')$ or $\mathcal{P}[\![q]\!](\sigma, \eta, \varsigma)$ and $\mathcal{P}'[\![q]\!](\sigma', \eta', \varsigma')$ are both undefined. In the latter case also the expression s will be undefined in both cases. Thus, we will only deal with the case that the values are defined in the sequel. This also holds for $\mathcal{P}[\![p]\!](\sigma, \eta, \varsigma)$ and $\mathcal{P}'[\![p]\!](\sigma', \eta', \varsigma')$.
Since $\mathcal{P}[\![q]\!](\sigma, \eta, \varsigma) \equiv \mathcal{P}'[\![q]\!](\sigma', \eta', \varsigma')$ also $\varsigma(\mathcal{P}[\![q]\!](\sigma, \eta, \varsigma)) \equiv \varsigma'(\mathcal{P}'[\![q]\!](\sigma', \eta', \varsigma'))$ $(*)$.
For the same reason $\varsigma(\mathcal{P}[\![p]\!](\sigma, \eta, \varsigma)) \equiv \varsigma'(\mathcal{P}'[\![p]\!](\sigma', \eta', \varsigma'))$ $(**)$.

- Base case: $s = q \in p$
  We will separately look at two cases:
  1. $\mathcal{P}[\![q]\!](\sigma, \eta, \varsigma) \in \mathit{SetLoc}$, that is q represents a set.
  2. $\mathcal{P}[\![q]\!](\sigma, \eta, \varsigma) \notin \mathit{SetLoc}$, q represents some primitive value or a location.

  1. Proof of "$\Rightarrow$":
     $\mathcal{X}[\![q \in p]\!](\sigma, \eta, \varsigma) \Rightarrow \varsigma(\mathcal{P}[\![q]\!](\sigma, \eta, \varsigma)) \in \varsigma(\mathcal{P}[\![p]\!](\sigma, \eta, \varsigma))$
     $(**) \Rightarrow \exists z \in \varsigma'(\mathcal{P}'[\![p]\!](\sigma', \eta', \varsigma')).\varsigma(\mathcal{P}[\![q]\!](\sigma, \eta, \varsigma)) \equiv_{ssl} z \Rightarrow$
     $\varsigma(\mathcal{P}[\![q]\!](\sigma, \eta, \varsigma)) \equiv_{ss} \varsigma'(z) \Rightarrow \mathcal{P}[\![q]\!](\sigma, \eta, \varsigma) \equiv_{slsl} z$
     By $\mathcal{P}[\![q]\!](\sigma, \eta, \varsigma) \equiv_{slsl} \mathcal{P}'[\![q]\!](\sigma', \eta', \varsigma')$ and the previous lemma:
     $(z \approx \mathcal{P}'[\![q]\!](\sigma', \eta', \varsigma'))(\sigma', \eta', \varsigma')$. So $\mathcal{X}'[\![q \in p]\!](\sigma', \eta', \varsigma')$.

     Proof of "$\Leftarrow$":
     $\mathcal{X}'[\![q \in p]\!](\sigma', \eta', \varsigma') \Rightarrow \exists z \in \varsigma'(\mathcal{P}'[\![p]\!](\sigma', \eta', \varsigma')).(z \approx \mathcal{P}'[\![q]\!](\sigma', \eta', \varsigma'))(\sigma', \eta', \varsigma')$.
     Since $\mathcal{P}[\![q]\!](\sigma, \eta, \varsigma) \equiv_{slsl} \mathcal{P}'[\![q]\!](\sigma', \eta', \varsigma')$ and by the previous lemma:
     $\mathcal{P}[\![q]\!](\sigma, \eta, \varsigma) \equiv_{slsl} z \Rightarrow \varsigma(\mathcal{P}[\![q]\!](\sigma, \eta, \varsigma)) \equiv_{ss} \varsigma'(z)$.
     $(**) \Rightarrow \exists y \in \varsigma(\mathcal{X}[\![p]\!](\sigma, \eta, \varsigma)).y \equiv_{ssl} z \Rightarrow y \equiv_{ss} \varsigma'(z)$. By the *Set Injectivity Lemma* we infer that $y = \varsigma(\mathcal{P}[\![q]\!](\sigma, \eta, \varsigma))$ and thus $\varsigma(\mathcal{P}[\![q]\!](\sigma, \eta, \varsigma)) \in \varsigma(\mathcal{P}[\![p]\!](\sigma, \eta, \varsigma))$, so $\mathcal{X}[\![q \in p]\!](\sigma, \eta, \varsigma)$.
  2. Here, $\mathcal{X}'[\![q \in p]\!](\sigma', \eta', \varsigma')$ simplifies to $\mathcal{P}'[\![q]\!](\sigma', \eta', \varsigma') \in \varsigma'(\mathcal{P}'[\![p]\!](\sigma', \eta', \varsigma'))$.
     Proof of "$\Rightarrow$":
     $\mathcal{X}[\![q \in p]\!](\sigma, \eta, \varsigma) \Rightarrow \mathcal{P}[\![q]\!](\sigma, \eta, \varsigma) \in \varsigma(\mathcal{P}[\![p]\!](\sigma, \eta, \varsigma))$
     $(**) \Rightarrow \exists z \in \varsigma'(\mathcal{P}'[\![p]\!](\sigma', \eta', \varsigma')).\mathcal{P}[\![q]\!](\sigma, \eta, \varsigma) \equiv z$
     Since we also know that $\mathcal{P}[\![q]\!](\sigma, \eta, \varsigma) \equiv \mathcal{P}'[\![q]\!](\sigma', \eta', \varsigma')$ we can infer by the previous lemma, that $(z \approx \mathcal{P}'[\![q]\!](\sigma', \eta', \varsigma'))(\sigma', \eta', \varsigma')$ and therefore $\mathcal{X}'[\![q \in p]\!](\sigma', \eta', \varsigma')$.

     Proof of "$\Leftarrow$":
     $\mathcal{X}'[\![q \in p]\!](\sigma', \eta', \varsigma') \Rightarrow \mathcal{P}'[\![q]\!](\sigma', \eta', \varsigma') \in \varsigma'(\mathcal{P}'[\![p]\!](\sigma', \eta', \varsigma'))$
     $(**) \Rightarrow \exists z \in \varsigma(\mathcal{P}[\![p]\!](\sigma, \eta, \varsigma)).z \equiv \mathcal{P}'[\![q]\!](\sigma', \eta', \varsigma')$. By the injectivity of $\equiv$ on $\mathbb{B}, \mathbb{Z}$ and *Loc* and the fact $\mathcal{P}[\![q]\!](\sigma, \eta, \varsigma) \equiv \mathcal{P}'[\![q]\!](\sigma', \eta', \varsigma')$ we infer $z = \mathcal{P}[\![q]\!](\sigma, \eta, \varsigma)$. So $\mathcal{X}[\![q \in p]\!](\sigma, \eta, \varsigma)$.
- Base case: $s = q \subseteq p$
  Proof of "$\Rightarrow$":
  $\mathcal{X}[\![q \in p]\!](\sigma, \eta, \varsigma) \Rightarrow \varsigma(\mathcal{P}[\![q]\!](\sigma, \eta, \varsigma)) \subseteq \varsigma(\mathcal{P}[\![p]\!](\sigma, \eta, \varsigma))$
  $(x \in \varsigma'(\mathcal{P}'[\![q]\!](\sigma', \eta', \varsigma')) \wedge (*)) \Rightarrow \exists z \in \varsigma(\mathcal{P}[\![q]\!](\sigma, \eta, \varsigma)).z \equiv x \Rightarrow$
  $(z \in \varsigma(\mathcal{A}[\![p]\!](\sigma, \eta, \varsigma)) \wedge (**)) \Rightarrow \exists z' \in \mathcal{A}'[\![q]\!](\sigma', \eta', \varsigma').z \equiv z'$.
  By the previous lemma: $(x \approx z')(\sigma', \eta', \varsigma')$. Thus, $\mathcal{X}'[\![q \in p]\!](\sigma', \eta', \varsigma')$

  Proof of "$\Leftarrow$":
  $\mathcal{X}'[\![q \in p]\!](\sigma', \eta', \varsigma') \Rightarrow \forall x \in \varsigma'(\mathcal{P}'[\![q]\!](\sigma', \eta', \varsigma')).\exists z \in \varsigma'(\mathcal{P}'[\![p]\!](\sigma', \eta', \varsigma')).(z \approx x)(\sigma', \eta', \varsigma')$
  $(x \in \varsigma(\mathcal{A}[\![q]\!](\sigma, \eta, \varsigma)) \wedge (*)) \Rightarrow \exists z \in \varsigma'(\mathcal{P}'[\![q]\!](\sigma', \eta', \varsigma')).x \equiv z$.

$\mathcal{X}'[\![q \in p]\!](\sigma, \eta, \varsigma) \Rightarrow \exists z' \in \varsigma'(\mathcal{P}'[\![p]\!](\sigma', \eta', \varsigma')).(z \approx z')(\sigma', \eta', \varsigma')$
By the previous lemma $x \equiv z'$. $(**) \Rightarrow \exists x' \in \varsigma(\mathcal{A}[\![p]\!](\sigma, \eta, \varsigma)).x' \equiv z'$

Case distinction depending on type of x:

- $x \in (\mathbb{B} \cup \mathbb{Z} \cup Loc)$:
  $\equiv$ is by definition injective on these values, so $x = x'$
- $x \in Set$:
  Then $x \equiv z' \Rightarrow x \equiv_{ssl} z' \Rightarrow x \equiv_{ss} \varsigma'(z')$ and
  $x' \equiv z' \Rightarrow x' \equiv_{ssl} z' \Rightarrow x' \equiv_{ss} \varsigma'(z')$. By the *Set Injectivity Lemma* $x = x'$.

So, $\mathcal{X}[\![q \in p]\!](\sigma, \eta, \varsigma)$.

- Step case: $s = \neg b_1$
  Again, either $\mathcal{B}[\![b_1]\!](\sigma, \eta, \varsigma)$ and $\mathcal{B}'[\![b_1]\!](\sigma', \eta', \varsigma')$ are undefined (then also $\mathcal{X}[\![s]\!](\sigma, \eta, \varsigma)$ and $\mathcal{X}'[\![s]\!](\sigma', \eta', \varsigma')$ are undefined) or the following holds:
  $\mathcal{X}[\![\neg b_1]\!](\sigma, \eta, \varsigma) = not\ \mathcal{B}[\![b_1]\!](\sigma, \eta, \varsigma) \underbrace{\Leftrightarrow}_{I.H.} not\ \mathcal{B}'[\![b_1]\!](\sigma', \eta', \varsigma') = \mathcal{X}'[\![\neg b_1]\!](\sigma', \eta', \varsigma')$
- Step case: $s = a_1\ op_a\ a_2$
  If $\mathcal{A}[\![a_1]\!](\sigma, \eta, \varsigma)$ or $\mathcal{A}[\![a_2]\!](\sigma, \eta, \varsigma)$ are undefined, then also their counterparts are undefined and thus both $\mathcal{X}[\![a_1\ op_a\ a_2]\!](\sigma, \eta, \varsigma)$ and $\mathcal{X}'[\![a_1\ op_a\ a_2]\!](\sigma', \eta', \varsigma')$ are undefined. Otherwise, by induction hypothesis:
  $\mathcal{A}[\![a_1]\!](\sigma, \eta, \varsigma) = \mathcal{A}'[\![a_1]\!](\sigma', \eta', \varsigma')$ and $\mathcal{A}[\![a_2]\!](\sigma, \eta, \varsigma) = \mathcal{A}'[\![a_2]\!](\sigma', \eta', \varsigma')$
  $\mathcal{X}[\![a_1\ op_a\ a_2]\!](\sigma, \eta, \varsigma) =$
  $\mathcal{A}[\![a_1]\!](\sigma, \eta, \varsigma)\ \mathbf{op_a}\ \mathcal{A}[\![a_2]\!](\sigma, \eta, \varsigma) = \mathcal{A}'[\![a_1]\!](\sigma', \eta', \varsigma')\ \mathbf{op_a}\ \mathcal{A}'[\![a_2]\!](\sigma', \eta', \varsigma') =$
  $\mathcal{X}'[\![a_1\ op_a\ a_2]\!](\sigma', \eta', \varsigma')$
- Step cases: $s = a_1\ op_r\ a_2$ and $s = b_1\ op_b\ b_2$
  Analogous to previous case.

■

Lemma 4:
Proof:

We prove the two claims separately.

- $(\sigma, \eta, \varsigma) \cong (\sigma'_2, \eta'_2, \varsigma'_2)$
  Let $\equiv$ be the *Heap Correspondence Relation* on (η, ς) and (η', ς'). We need to prove that there exists a *Heap Correspondence Relation* $\equiv'$ for (η, ς) and (η'_2, ς'_2) and that $\sigma(y) \equiv' \sigma'_2(y)$ for all $y \in dom(\sigma)$. By the *Stability Lemma* we easily infer that there exists a $\equiv'$ with $(\eta, s) \equiv' (\eta'_2, \varsigma'_2)$. The two `malloc set` statements introduce unaliased set locations, which are then manipulated. For all variables $y \in (dom(\sigma) \setminus \{x\}$ we know that $\sigma'_2(y) \neq \sigma'_2(x)$ by the condition for [Malloc-Set']. Again we can use the *Stability Lemma* to find that $\sigma(y) \equiv' \sigma'_2(y)$. So it only remains to show that $\sigma(x) \equiv' \sigma'_2(x)$.

One easily sees that $\varsigma'(\sigma'(x)) = \varsigma_2'(\sigma_2'(x))$. By the fact that $(\sigma, \eta, \varsigma) \cong (\sigma', \eta', \varsigma')$ we know that $\sigma(x) \equiv \sigma'(x)$ and thus $\varsigma(\sigma(x)) \equiv \varsigma'(\sigma'(x))$. Since $\varsigma'(\sigma'(x)) \neq \sigma_2'(x)$ we infer by the *Stability Lemma* $\varsigma(\sigma(x)) \equiv' \varsigma'(\sigma'(x))$. From this and the previous fact we conclude $\varsigma(\sigma(x)) \equiv' \varsigma_2'(\sigma_2'(x))$ which is by definition equivalent to $\sigma(x) \equiv \sigma_2'(x)$.

- $\sigma_2'(x) \notin (im(\sigma_2'[x \mapsto undef.]) \cup im(\eta_2') \cup \bigcup im(\varsigma_2'))$ $\sigma_2'(x) = \psi_2'$. The inference rule [Malloc-Set] ensures that ψ_2' does not occur in the state also [Assignment-Set] does only change the contents of $\varsigma(\psi_2')$ which proves the claim.

$\blacksquare$

Lemma 5:
Proof:

We will show that whenever an inference rule in Semantics I applies, there are inference rules in Semantics II that will have an equivalent effect. The proof is by induction over the structure of the statements.

The following inference rules do not manipulate the state, hence the *Heap Correspondence Relation* that is valid before the step remains valid after it.

- $\langle \texttt{skip};\, S, (\sigma, \eta, \varsigma)\rangle \triangleright \langle S, (\sigma, \eta, \varsigma)\rangle$ [Skip-Elimination]
  Let $\langle \texttt{skip};\, S, (\sigma, \eta, \varsigma)\rangle \simeq \langle \texttt{skip};\, T(S), (\sigma', \eta', \varsigma')\rangle$. Then [Skip-Elimination'] applies and we get $\langle S, (\sigma, \eta, \varsigma)\rangle \simeq \langle T(S), (\sigma', \eta', \varsigma')\rangle$.

- $\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2, (\sigma, \eta, \varsigma)\rangle \triangleright \langle S_1, (\sigma, \eta, \varsigma)\rangle$ where $\mathcal{B}[\![b]\!](\sigma, \eta, \varsigma) = \mathbf{1}$ [If-True]
  Let $\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2, (\sigma, \eta, \varsigma)\rangle \simeq \langle \texttt{if}\ b\ \texttt{then}\ T(S_1)\ \texttt{else}\ T(S_2), (\sigma', \eta', \varsigma')\rangle$. By the *Expressions Coincide Lemma* $\mathcal{B}[\![b]\!](\sigma, \eta, \varsigma) = \mathcal{B}'[\![b]\!](\sigma', \eta', \varsigma')$. So [If-True'] also applies in Semantics II and we get $\langle S_1, (\sigma, \eta, \varsigma)\rangle \simeq \langle T(S_1), (\sigma', \eta', \varsigma')\rangle$.

- $\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2, (\sigma, \eta, \varsigma)\rangle \triangleright \langle S_2, (\sigma, \eta, \varsigma)\rangle$ where $\mathcal{B}[\![b]\!](\sigma, \eta, \varsigma) = \mathbf{0}$ [If-False]
  Analogous to previous case.

- $\langle \texttt{while}\ b\ \texttt{do}\ S, (\sigma, \eta, \varsigma)\rangle \triangleright \langle S;\ \texttt{while}\ b\ \texttt{do}\ S, (\sigma, \eta, \varsigma)\rangle$ where $\mathcal{B}[\![b]\!](\sigma, \eta, \varsigma) = \mathbf{1}$ [While-True]
  Analogous to [If-True] case.

- $\langle \texttt{while}\ b\ \texttt{do}\ S, (\sigma, \eta, \varsigma)\rangle \triangleright \langle \texttt{skip}, (\sigma, \eta, \varsigma)\rangle$ where $\mathcal{B}[\![b]\!](\sigma, \eta, \varsigma) = \mathbf{0}$ [While-False]
  Analogous to [If-True] case.

The following inference rules change the state. Therefore we need to show that the resulting states still correspond. Either by showing that the previous *Heap Correspondence Relation* is still valid or by giving an adjusted version.

- $$\langle x := s, (\sigma, \eta, \varsigma)\rangle \triangleright \langle \texttt{skip}, (\sigma[x \mapsto \mathcal{X}[\![s]\!](\sigma, \eta, \varsigma)], \eta, \varsigma)\rangle \quad \text{[Assignment]}$$
$$\text{if } \mathcal{X}[\![s]\!](\sigma, \eta, \varsigma) \in (\mathit{Item} \setminus \mathit{SetLoc})$$
Let $\langle x := s, (\sigma, \eta, \varsigma)\rangle \simeq \langle x := s, (\sigma', \eta', \varsigma')\rangle$. Then [Assignment'] applies for $\langle x := s, (\sigma', \eta', \varsigma')\rangle$. We need to show that $\langle \texttt{skip}, (\sigma[x \mapsto \mathcal{X}[\![s]\!](\sigma, \eta, \varsigma)], \eta, \varsigma)\rangle \simeq \langle \texttt{skip}, (\sigma'[x \mapsto \mathcal{X}'[\![s]\!](\sigma', \eta', \varsigma')], \eta', \varsigma')\rangle$ The previous *Heap Correspondence Relation* remains valid. Heap and Set heap are not changed by the inference and $\sigma(x) \equiv \sigma'(x) \Leftrightarrow \mathcal{X}[\![s]\!] \equiv \mathcal{X}'[\![s]\!]$ by the *Expressions Coincide Lemma*.

- $$\langle x.\mathit{sel} := s, (\sigma, \eta, \varsigma)\rangle \triangleright \langle \texttt{skip}, (\sigma, \eta[(\sigma(x), \mathit{sel}) \mapsto \mathcal{X}[\![s]\!](\sigma, \eta, \varsigma)], \varsigma)\rangle \quad \text{[Assignment-Heap]}$$
$$\text{if } \mathcal{X}[\![s]\!](\sigma, \eta, \varsigma) \in (\mathit{Item} \setminus \mathit{SetLoc})$$
Again the corresponding rule [Assignment-Heap'] applies and the *Heap Correspondence Relation* remains valid. $\eta(\sigma(x), \mathit{sel})$ and $\eta'(\sigma'(x), \mathit{sel})$ have been changed. We have to verify that $\sigma(x) \equiv_{ll} \sigma'(x)$, which was true before the inference step by definition, i.e. all selector-fields agreed or were undefined. By *Expressions Coincide Lemma* $\mathcal{X}[\![s]\!] \equiv \mathcal{X}'[\![s]\!]$ and so $\eta(\sigma(x), \mathit{sel})$ and $\eta'(\sigma'(x), \mathit{sel})$ after the inference step.

- $$\langle x := s, (\sigma, \eta, \varsigma)\rangle \triangleright \langle \texttt{skip}, (\sigma, \eta, \varsigma[\sigma(x) \mapsto \varsigma(\mathcal{X}[\![s]\!](\sigma, \eta, \varsigma))])\rangle \quad \text{[Assignment-Set]}$$
$$\text{if } \mathcal{X}[\![s]\!](\sigma, \eta, \varsigma) \in \mathit{SetLoc}$$
The configuration corresponding to $\langle x := s, (\sigma, \eta, \varsigma)\rangle$ is of the form $\langle x := \texttt{malloc set}; x := s, (\sigma', \eta', \varsigma')\rangle$. Application of [Seq. Composition'] with [Malloc-Set'] and [Assignment-Set'] result in the configuration $\langle \texttt{skip}, \sigma'[x \mapsto \psi'], \eta', \varsigma'[\psi' \mapsto \varsigma'(\mathcal{X}'[\![s]\!](\sigma', \eta', \varsigma'))])\rangle$.
We can use the *Stability Lemma* to prove that there exists a *Heap Correspondence Relation* $\equiv'$ for $(\eta, \varsigma[\sigma(x) \mapsto \varsigma(\mathcal{X}[\![s]\!](\sigma, \eta, \varsigma))])$ and $(\eta', \varsigma'[\psi' \mapsto \varsigma'(\mathcal{X}'[\![s]\!](\sigma', \eta', \varsigma'))])$, since ψ' which was introduced by `malloc set` is not aliased. The *Stability Lemma* also gives us $\sigma(y) \equiv' \sigma'[x \mapsto \psi'](y)$ because $\sigma'[x \mapsto \psi'](y) \neq \psi'$ and $\sigma(y) \neq \sigma(x)$ (aliasing is impossible in the first semantics).
So it remains to show $\sigma(x) \equiv' \sigma'[x \mapsto \psi'](x) = \psi'$. The requirements for a *Heap Correspondence Relation* give us $\sigma(x) \equiv' \psi' \Leftrightarrow \varsigma(\mathcal{X}[\![s]\!](\sigma, \eta, \varsigma)) = \varsigma[\sigma(x) \mapsto \varsigma(\mathcal{X}[\![s]\!](\sigma, \eta, \varsigma))](\sigma(x)) \equiv' \varsigma'[\psi' \mapsto \varsigma'(\mathcal{X}'[\![s]\!](\sigma', \eta', \varsigma'))](\psi') = \varsigma'(\mathcal{X}'[\![s]\!](\sigma', \eta', \varsigma'))$. By the *Expressions Coincide Lemma* $\mathcal{X}[\![s]\!](\sigma, \eta, \varsigma) \equiv \mathcal{X}'[\![s]\!](\sigma', \eta', \varsigma')$. $\mathcal{X}[\![s]\!](\sigma, \eta, \varsigma) \neq \sigma(x)$ and $\mathcal{X}'[\![s]\!](\sigma', \eta', \varsigma') \neq \psi'$ so also $\mathcal{X}[\![s]\!](\sigma, \eta, \varsigma) \equiv' \mathcal{X}'[\![s]\!](\sigma', \eta', \varsigma')$ by the *Stability Lemma*. The conditions for $\equiv'$ finally give us $\varsigma(\mathcal{X}[\![s]\!](\sigma, \eta, \varsigma)) \equiv' \varsigma'(\mathcal{X}'[\![s]\!](\sigma', \eta', \varsigma'))$ which is equivalent to $\sigma(x) \equiv' \sigma'[x \mapsto \psi'](x)$.

- $$\langle x.\mathit{sel} := s, (\sigma, \eta, \varsigma)\rangle \triangleright \langle \texttt{skip}, (\sigma, \eta, \varsigma[\eta(\sigma(x), \mathit{sel}) \mapsto \varsigma(\mathcal{X}[\![s]\!](\sigma, \eta, \varsigma))])\rangle \quad \text{[Assignment-Heap-Set]}$$
$$\text{if } \mathcal{X}[\![s]\!](\sigma, \eta, \varsigma) \in \mathit{SetLoc}$$
Similar to proof of [Assignment-Set].

- $$\langle x := \texttt{malloc}, (\sigma, \eta, \varsigma)\rangle \triangleright \langle \texttt{skip}, (\sigma[x \mapsto \xi], \eta, \varsigma)\rangle \quad \text{[Malloc]}$$
$$\text{where } \xi \in \mathit{Loc} \text{ and } \xi \notin (\mathit{im}(\sigma) \cup \mathit{dom}(\eta) \cup \mathit{im}(\eta) \cup \bigcup \mathit{im}(\varsigma))$$
The same inference rule can be used in Semantics II. Let $\equiv$ be the *Heap Correspondence Relation* between the two states prior to the execution of the rule and let ξ

and ξ' be the two new locations introduced. Then $\equiv' = \equiv \cup (\xi, \xi')$ is a *Heap Correspondence Relation* for the resulting states. Since ξ and ξ' are new locations they do not occur in $\equiv$ and thus do not violate any of rules involving other elements. The requirement for $\xi \equiv_{ll} \xi'$ is also fulfilled, since all selector-fields are undefined. Finally, $\psi = \sigma(x) \equiv' \sigma'(x) = \psi'$, so the resulting states are corresponding.

- $\langle x.sel := \texttt{malloc}, (\sigma, \eta, \varsigma)\rangle \triangleright \langle \texttt{skip}, (\sigma, \eta[(\sigma(x), sel) \mapsto \xi], \varsigma)\rangle$ [Malloc-Heap]
  where $\xi \in Loc$ and $\xi \notin (im(\sigma) \cup dom(\eta) \cup im(\eta) \cup \bigcup im(\varsigma))$

Analogous to the previous case.

- $\langle x := \texttt{malloc set}, (\sigma, \eta, \varsigma)\rangle \triangleright \langle \texttt{skip}, (\sigma[x \mapsto \psi], \eta, \varsigma[\psi \mapsto \emptyset])\rangle$ [Malloc-Set]
  where $\psi \in SetLoc$ and $\psi \notin (im(\sigma) \cup im(\eta) \cup dom(\varsigma))$

The corresponding inference rule [Malloc-Set'] applies, so it remains to show that $(\sigma[x \mapsto \psi], \eta, \varsigma[\psi \mapsto \emptyset]) \cong (\sigma'[x \mapsto \psi'], \eta', \varsigma'[\psi' \mapsto \emptyset])$. The *Stability Lemma* can be applied to infer the existence of $\equiv'$ for $(\eta, \varsigma[\psi \mapsto \emptyset])$ and $\eta', \varsigma'[\psi' \mapsto \emptyset])$. By the *Stability Lemma* we can also follow that $\sigma[x \mapsto \psi](y) \equiv' \sigma[x \mapsto \psi'](y)$ for $y \in (dom(\sigma[x \mapsto \psi]) \setminus \{x\})$, since $\sigma[x \mapsto \psi](y) \neq \psi$ and $\sigma'[x \mapsto \psi'](y) \neq \psi'$. Finally, $\sigma[x \mapsto \psi](x) = \psi \equiv' \psi' = \sigma'[x \mapsto \psi'](x)$ because $\varsigma[\psi \mapsto \emptyset](\psi) = \emptyset \equiv_{ss} \emptyset = \varsigma'[\psi' \mapsto \emptyset](\psi')$.

- $\langle x.sel := \texttt{malloc set}, (\sigma, \eta, \varsigma)\rangle \triangleright \langle \texttt{skip}, (\sigma, \eta[(\sigma(x), sel) \mapsto \psi], \varsigma[\psi \mapsto \emptyset])\rangle$ [Malloc-Set-Heap]
  where $\psi \in SetLoc$ and $\psi \notin (im(\sigma) \cup im(\eta) \cup dom(\varsigma))$

Analogous to previous case.

- $\langle x.\texttt{insert}(s), (\sigma, \eta, \varsigma)\rangle \triangleright \langle \texttt{skip}, (\sigma, \eta, \varsigma[\sigma(x) \mapsto (\varsigma(\sigma(x)) \cup \{i\})])\rangle$ [Set-Insert]
  where $i = \begin{cases} \varsigma(\mathcal{X}[\![s]\!](\sigma, \eta, \varsigma)), & \text{if } \mathcal{X}[\![s]\!](\sigma, \eta, \varsigma) \in SetLoc \\ \mathcal{X}[\![s]\!](\sigma, \eta, \varsigma), & \text{otherwise} \end{cases}$

The configuration corresponding to $\langle x.\texttt{insert}(s), (\sigma, \eta, \varsigma)\rangle$ is $\langle x_{temp} := \texttt{malloc set};$ $x_{temp} := x;\ x := \texttt{malloc set};\ x := x_{temp};\ x.\texttt{insert}(s), (\sigma', \eta', \varsigma')\rangle$.
By the *Aliasing Lemma* we can execute the first four commands of the sequence and get a new configuration $\langle x.\texttt{insert}(s), (\sigma'_2, \eta'_2, \varsigma'_2)\rangle$, where $(\sigma, \eta, \varsigma) \cong (\sigma'_2, \eta'_2, \varsigma'_2)$ and $\sigma'_2(x)$ is not aliased.

For the resulting configuration [Set-Insert'] is applicable. We distinguish two cases:

1. $\exists z.z \in \varsigma'_2(\sigma'_2(x)) \wedge (z \approx \mathcal{X}'[\![s]\!](\sigma'_2, \eta'_2, \varsigma'_2))(\sigma'_2, \eta'_2, \varsigma'_2)$:
   Take such a z. Since $\sigma(x) \equiv \sigma'_2(x)$ we have $\varsigma(\sigma(x)) \equiv_{ss} \varsigma'_2(\sigma'_2(x))$, which implies $\exists x \in \varsigma(\sigma(x)).x \equiv z$.
   Because of $(z \approx \mathcal{X}'[\![s]\!])$ and the $\equiv / \approx$ *Relation Lemma* we get $x \equiv \mathcal{X}'[\![s]\!]$.
   By the *Expressions Coincide Lemma* $\mathcal{X}[\![s]\!] \equiv \mathcal{X}'[\![s]\!]$. We have to look at two cases here:
   - $\mathcal{X}'[\![s]\!] \in SetLoc'$ From $\mathcal{X}[\![s]\!] \equiv_{slsl} \mathcal{X}'[\![s]\!]$ we can follow $\varsigma(\mathcal{X}[\![s]\!]) \equiv_{ss} \varsigma'_2(\mathcal{X}'[\![s]\!])$ and from $z \equiv_{ssl} \mathcal{X}'[\![s]\!]$ we follow $z \equiv_{ss} \varsigma'_2(\mathcal{X}'[\![s]\!])$. By injectivity of $\equiv_{ss}$ we

infer $z = \varsigma(\mathcal{X}[\![s]\!])$. That is $\varsigma(\mathcal{X}[\![s]\!])$ is already part of the set and [Set-Insert] has no effect on the state.

- $\mathcal{X}'[\![s]\!] \notin SetLoc'$
  Since $\equiv \setminus \equiv_{slsl}$ is injective (by *Set Injectivity Lemma* for $\equiv_{ss}$ and trivially for the other parts of the relation) we get $x = \mathcal{X}[\![s]\!]$.
  This means that [Set-Insert] has no effect, since x is already an element of the set.

[Set-Insert'] does not change the state either in this case. Thus, the existing *Heap Correspondence Relation* remains valid.

2. $\neg\exists z.z \in \varsigma'_2(\sigma'_2(x)) \wedge (z \approx \mathcal{X}'[\![s]\!](\sigma'_2, \eta'_2, \varsigma'_2))(\sigma'_2, \eta'_2, \varsigma'_2)$:
In this case the elements are not part of the sets before and will be inserted. By the *Stability Lemma* there exists a *Heap Correspondence Relation*$\equiv'$ for $(\eta, \varsigma[\sigma(x) \mapsto (\varsigma(\sigma(x)) \cup \{i\})])$ and $(\eta'_2, \varsigma'_2[\sigma'_2(x) \mapsto (\varsigma'_2(\sigma'_2(x)) \cup \{\mathcal{X}'[\![s]\!](\sigma'_2, \eta'_2, \varsigma'_2)\})])$ where $i = \begin{cases} \varsigma(\mathcal{X}[\![s]\!](\sigma, \eta, \varsigma)), & \text{if } \mathcal{X}[\![s]\!](\sigma, \eta, \varsigma) \in SetLoc \\ \mathcal{X}[\![s]\!](\sigma, \eta, \varsigma), & \text{otherwise} \end{cases}$.
The application of the *Stability Lemma* is possible because $\sigma'_2(x)$ and $\sigma(x)$ are not aliased. As in the previous proofs we need to show that $\sigma(y) \equiv' \sigma'_2(y)$ for all $y \in dom(\sigma)$. For $y \in (dom(\sigma) \setminus \{x\})$ this also follows from the *Stability Lemma* as $\sigma(y) \neq \sigma(x)$ and $\sigma'_2(y) \neq \sigma'_2(x)$. This leaves us with $\sigma(x) \equiv' \sigma'_2(x)$ to prove. This is by definition equivalent to $(\varsigma(\sigma(x)) \cup \{i\}) = \varsigma[\sigma(x) \mapsto (\varsigma(\sigma(x)) \cup \{i\})](\sigma(x)) \equiv'_{ss} \varsigma'_2[\sigma'_2(x) \mapsto (\varsigma'_2(\sigma'_2(x)) \cup \{\mathcal{X}'[\![s]\!](\sigma'_2, \eta'_2, \varsigma'_2)\})](\sigma'_2(x)) = (\varsigma'_2(\sigma'_2(x)) \cup \{\mathcal{X}'[\![s]\!](\sigma'_2, \eta'_2, \varsigma'_2)\})$. We know that $\varsigma(\sigma(x)) \equiv'_{ss} \varsigma'_2(\sigma'_2(x))$. So showing $i \equiv \mathcal{X}'[\![s]\!](\sigma'_2, \eta'_2, \varsigma'_2)$ suffices to close the proof. In fact, this follows from the *Expressions Coincide Theorem*, since i is either $\varsigma(\mathcal{X}[\![s]\!](\sigma, \eta, \varsigma))$ or $\mathcal{X}[\![s]\!](\sigma, \eta, \varsigma)$.

- $$\langle x.\texttt{remove}(s), (\sigma, \eta, \varsigma)\rangle \triangleright \langle \texttt{skip}, (\sigma, \eta, \varsigma[\sigma(x) \mapsto (\varsigma(\sigma(x)) \setminus \{i\})])\rangle \quad \text{[Set-Remove]}$$
  $$\text{where } i = \begin{cases} \varsigma(\mathcal{X}[\![s]\!](\sigma, \eta, \varsigma)), & \text{if } \mathcal{X}[\![s]\!](\sigma, \eta, \varsigma) \in SetLoc \\ \mathcal{X}[\![s]\!](\sigma, \eta, \varsigma), & \text{otherwise} \end{cases}$$
  Analogous to previous case replacing $\cup$ with $\setminus$.

- $$\langle x := y.\texttt{selectAndRemove}, (\sigma, \eta, \varsigma)\rangle \triangleright \langle \texttt{skip}, (\sigma, \eta, \varsigma[\sigma(y) \mapsto (\varsigma(\sigma(y)) \setminus \{el\})][\sigma(x) \mapsto el])\rangle \quad \text{[Set-SelectRemove-Set]}$$
  $$\text{where } el \in \varsigma(\sigma(y)) \text{ and } el \in Set$$
  The configuration corresponding to $\langle x := y.\texttt{selectAndRemove}, (\sigma, \eta, \varsigma)\rangle$ is $\langle y_{temp} := \texttt{malloc set};\ y_{temp} := y;\ y := \texttt{malloc set};\ y := y_{temp};\ x := \texttt{malloc set};$ $x := y.\texttt{selectAndRemove}, (\sigma', \eta', \varsigma')\rangle$. By the *Aliasing Lemma* we can execute the first four statements of the series to obtain a new configuration $\langle x := \texttt{malloc set};\ x := y.\texttt{selectAndRemove}, (\sigma'_2, \eta'_2, \varsigma'_2)\rangle$ with $(\sigma, \eta, \varsigma) \cong (\sigma'_2, \eta'_2, \varsigma'_2)$ and $\sigma'_2(y)$ not aliased. Combining the effects of [Malloc-Set'] and [Set-SelectRemove-Set'] we get $\langle \texttt{skip}, (\sigma'_2[x \mapsto \psi'], \eta'_2, \varsigma'_2[\sigma'_2(y) \mapsto (\varsigma'_2(\sigma'_2(y)) \setminus \{el'\})][\psi' \mapsto \varsigma'_2(el')])\rangle$, where $el' \in \varsigma'_2(\sigma'_2(y))$ and $\psi' \notin (im(\sigma'_2) \cup im(\eta'_2) \cup dom(\varsigma'_2) \cup \bigcup im(\varsigma'_2))$. By the fact that $(\sigma, \eta, \varsigma) \cong (\sigma'_2, \eta'_2, \varsigma'_2)$ we conclude that $\varsigma(\sigma(y)) \equiv_{ss} \varsigma'_2(\sigma'_2(y))$. We assume that $el \equiv el'$ in the following. This is possible due to the nondeterminism of the

[Set-SelectRemove-Set'] rule.

We need to prove that the resulting configurations correspond. The statements `skip` and `skip` obviously correspond. By the *Stability Lemma* and the fact that $\sigma(x), \sigma(y), \sigma_2'(x)$ and $\sigma_2'(y)$ are not aliased we infer that there exists a *Heap Correspondence Relation*$\equiv'$ for $(\eta, \varsigma[\sigma(y) \mapsto (\varsigma(\sigma(y)) \setminus \{el\})][\sigma(x) \mapsto el])$ and $(\eta_2', \varsigma_2'[\sigma_2'(y) \mapsto (\varsigma_2'(\sigma_2'(y)) \setminus \{el'\})][\psi' \mapsto \varsigma_2'(el')])$. The *Stability Lemma* also allows us to infer $\sigma(z) \equiv' \sigma_2'[x \mapsto \psi'](z)$ for $z \in (dom(\sigma) \setminus \{x, y\})$. We still need to prove $\sigma(x) \equiv' \sigma_2'[x \mapsto \psi'](x) = \psi'$ and $\sigma(y) \equiv' \sigma_2'[x \mapsto \psi'](y)$. We know that $el \equiv' el'$ and that $\varsigma[\sigma(y) \mapsto (\varsigma(\sigma(y)) \setminus \{el\})][\sigma(x) \mapsto el](\sigma(x)) = el$ and $\varsigma_2'[\sigma_2'(y) \mapsto (\varsigma_2'(\sigma_2'(y)) \setminus \{el'\})][\psi' \mapsto \varsigma_2'(el')](\sigma_2'(x)) = el'$ which proves $\sigma(x) \equiv' \sigma_2'[x \mapsto \psi'](x)$. $\sigma(y) \equiv' \sigma_2'[x \mapsto \psi'](y)$ is equivalent to $(\varsigma(\sigma(y)) \setminus \{el\}) \equiv_{ss}' (\varsigma_2'(\sigma_2'(y)) \setminus \{el'\})$. We know that $\varsigma(\sigma(y)) \equiv_{ss}' \varsigma_2'(\sigma_2'(y))$ and $el \equiv el'$ which obviously implies $(\varsigma(\sigma(y)) \setminus \{el\}) \equiv_{ss}' (\varsigma_2'(\sigma_2'(y)) \setminus \{el'\})$.

- $\langle x := y.\texttt{selectAndRemove}, (\sigma, \eta, \varsigma)\rangle \triangleright \langle \texttt{skip}, (\sigma[x \mapsto el], \eta, \varsigma[\sigma(y) \mapsto (\varsigma(\sigma(y)) \setminus \{el\})])\rangle$ [Set-SelectRemove]
  where $el \in \varsigma(\sigma(y))$ and $el \in (\mathit{Item} \setminus \mathit{SetLoc})$

  Analogous to previous case.

The last missing inference rule is [Seq. Composition]. This is the only "real" step case of the proof, i.e. the only case that relies on the induction hypothesis.

- $$\frac{\langle S_1, (\sigma_1, \eta_1, \varsigma_1)\rangle \triangleright \langle S_2, (\sigma_2, \eta_2, \varsigma_2)\rangle}{\langle S_1;\, S, (\sigma_1, \eta_1, \varsigma_1)\rangle \triangleright \langle S_2;\, S, (\sigma_2, \eta_2, \varsigma_2)\rangle} \quad \text{[Seq. Composition]}$$
  By induction hypothesis we know that if $\langle S_1, (\sigma, \eta, \varsigma)\rangle \triangleright \langle S_2, (\sigma_2, \eta_2, \varsigma_2)\rangle$ and $\langle S_1, (\sigma_1, \eta_1, \varsigma_1)\rangle \simeq \langle T(S_1), (\sigma_1', \eta_1', \varsigma_1')\rangle$ then there exists some $\langle S_2, (\sigma_2, \eta_2, \varsigma_2)\rangle \simeq \langle T(S_2), (\sigma_2', \eta_2', \varsigma_2')\rangle$ with $\langle T(S_1), (\sigma_1', \eta_1', \varsigma_1')\rangle \triangleright^* \langle T(S_2), (\sigma_2', \eta_2', \varsigma_2')\rangle$.
  For every inference step from the series of inferences from the induction hypothesis we can apply the [Seq. Composition']-rule. This yields $\langle T(S_1);\, T(S), (\sigma_1', \eta_1', \varsigma_1')\rangle \triangleright^* \langle T(S_2);\, T(S), (\sigma_2', \eta_2', \varsigma_2')\rangle$.

■

B Source Code

B.1 C Implementations

B.1.1 List-based Implementation

Structure Declarations

```
typedef struct List
{
  void* data;
  struct List* next;
} List;

typedef struct Set
{
  List* list;
  int (*compare)(void*, void*);
  int size;
} Set;


Set* emptySet(int (*comp)(void*, void*));
        //comp should return 0 iff the parameters have the same value
void insertElement(Set* set, void* element);
void* removeElement(Set* set, void* element);
int isElement(Set* set, void* element);
void addSet(Set* set1, Set* set2);
void subSet(Set* set1, Set* set2);
int isSubset(Set* set1, Set* set2);
Set* copySet(Set* set);
int sizeOf(Set* set);
```

Implementation

```
#include <stdio.h>

#include "set.h"

Set* emptySet(int (*comp)(void*, void*))
```

```
{
    Set* emptySet;

    emptySet = (Set*)malloc(sizeof(Set));
    emptySet->compare = comp;
    emptySet->list = 0;
    emptySet->size = 0;

    return emptySet;
}

int isEmpty(Set* set)
{
  return (set->list == 0);
}

void insertElement(Set* set, void* element)
{
  List* list = set->list;
  List* prev = 0;

  while (list != 0)
    {
      if (compare(list->data, element) == 0)
        return;

      prev = list;
      list = list->next;
    }

  List* newList = (List*)malloc(sizeof(List));
  newList->data = element;
  newList->next = 0;
  set->size++;

  if (prev == 0) //list is empty
    {
      set->list = newList;
    }
  else //append item to list
    {
      prev->next = newList;
    }
}
```

```
void* removeElement(Set* set, void* element)
{
  List* temp;
  List* list = set->list;

  if (list == 0)
    return;

  if (compare(list->data, element) == 0)
    {
      set->size--;
      set->list = list->next;
      free(list);
    }
  else
    while (list->next != 0)
      {
        if (compare(list->next->data, element) == 0)
          {
            void* deletedElement = list->next->data;
            set->size--;
            temp = list->next->next;
            free(list->next);
            list->next = temp;
            return deletedElement;
          }
        list = list->next;
      }
}


int isElement(Set* set, void* element)
{
    List* list = set->list;

    while (list != 0)
    {
        if (compare(list->data, element) == 0)
            return 1;

        list = list->next;
    }

    return 0;
```

```
}

void addSet(Set* set1, Set* set2)
{
  List* list = set2->list;
  while (list != 0)
    {
      insertElement(set1, list->data);
      list = list->next;
    }
}

void subSet(Set* set1, Set* set2)
{
  List* list = set2->list;
  while (list != 0)
    {
      removeElement(set1, list->data);
      list = list->next;
    }
}

int isSubset(Set* set1, Set* set2)
{
  List* list = set1->list;
  while (list != 0)
    {
      if (!isElement(set2, list->data))
        return 0;
      list = list->next;
    }

  return 1;
}

Set* copySet(Set* set2)
{
  Set* newset = emptySet(set2->compare);
  addSet(newset, set2);
  return newset;
}

int sizeOf(Set* set)
{
  return set->size;
```

```
}

int compare(void* a, void* b)
{
  return (*((int*)a) - *((int*)b));
}


int main(int argc, char** argv)
{
  Set* mySet, *mySet2;
  mySet = emptySet(&compare);
  mySet2 = emptySet(&compare);

  int a, b, c;
  a = 34;
  b = 344;
  c = 3423;

  insertElement(mySet2, &b);

  insertElement(mySet, &a);

  insertElement(mySet, &a);
  insertElement(mySet, &b);
  insertElement(mySet, &c);
  removeElement(mySet, &a);

  if (isElement(mySet, &a))
    printf("a in mySet\n");
  else
    printf("a not in mySet\n");

  if (isElement(mySet, &b))
    printf("b in mySet\n");
  else
    printf("b not in mySet\n");

  if (isSubset(mySet2, mySet))
    printf("mySet2 is subset of mySet\n");
  else
    printf("mySet2 is not subset of mySet\n");

  if (isSubset(copySet(mySet), mySet))
    printf("mySet is subset of mySet\n");
```

```
  else
    printf("mySet is not subset of mySet\n");

  printf("Size of mySet: %i\n", sizeOf(mySet));
  printf("Size of mySet2: %i\n", sizeOf(mySet2));


  free(mySet);
  free(mySet2);
  return 1;
}
```

B.1.2 Tree-based Implementation

Structure Declarations

```
typedef struct Tree
{
  void* data;
  struct Tree* left;
  struct Tree* right;
} Tree;

typedef struct Set
{
  Tree* tree;
  int (*compare)(void*, void*);
  int size;
} Set;


Set* emptySet(int (*comp)(void*, void*));
        //comp should return 0 iff the parameters have the same value
int isEmpty(Set* set);
void insertElement(Set* set, void* element);
void removeElement(Set* set, void* element);
int isElement(Set* set, void* element);
void addSet(Set* set1, Set* set2);
void subSet(Set* set1, Set* set2);
int isSubset(Set* set1, Set* set2);
Set* copySet(Set* set);
int sizeOf(Set* set);
```

Implementation

```
#include <stdio.h>
#include <stdlib.h>
#include "set.h"

Set* emptySet(int (*comp)(void*, void*))
{
    Set* emptySet;

    emptySet = (Set*)malloc(sizeof(Set));
    emptySet->compare = comp;
    emptySet->size = 0;

    return emptySet;
}

int isEmpty(Set* set)
{
  return (set->tree == 0);
}


void insertElement(Set* set, void* element)
{
    if (!isElement(set, element))
    {
      set->size++;
      Tree* tree = set->tree;
      Tree* previous = tree;
      int compresult;

      while (tree != 0)  //find suitable position for new element
        {
          previous = tree;
          compresult = compare(tree->data, element);
          if (compresult < 0)
            tree = tree->left;
          else if (compresult > 0)
            tree = tree->right;
        }

      tree = (Tree*)malloc(sizeof(tree));
      tree->data = element;
      tree->left = 0;
      tree->right = 0;
```

```
      if (previous == 0) //first element to be inserted...
        {
          set->tree = tree;
        }
      else
        {
          if (compresult < 0)
            previous->left = tree;
          else if (compresult > 0)
            previous->right = tree;
        }
    }
}

void removeElement(Set* set, void* element)
{
    Tree* tree = set->tree;
    Tree* previous = 0;
    int oldcompresult = 0;

    while (tree != 0) //find element...
      {
        int compresult = compare(tree->data, element);
        if (compresult == 0) //we found the element.
        {
          set->size--;
          if ((tree->right == 0) && (tree->left == 0)) //it had not successors
            {
              if (previous == 0)
                set->tree = 0;
              else if (previous->left == tree)
                previous->left = 0;
              else
                previous->right = 0;
            }
          else if (tree->right == 0) //only one successor
            {
              if (previous == 0)
                set->tree = tree->left;
              else if (previous->left == tree)
                previous->left = tree->left;
              else
                previous->right = tree->left;
            }
          else if (tree->left == 0) //only one successor
```

```
    {
      if (previous == 0)
        set->tree = tree->right;
      else if (previous->left == tree)
        previous->left = tree->right;
      else
        previous->right = tree->right;
    }
  else
    {  //position has two subtrees: either find largest element to the left
        //or smallest to the right; i chose left here
      Tree* subtree = tree->left;
      Tree* previous2 = 0;
      while (subtree->right != 0) //finding largest element to the
              //left of the element that is being removed
        {
          previous2 = subtree;
          subtree = subtree->right;
        }
      if (previous2 != 0)     //remove element from predecessor
        previous2->right = 0;

      subtree->left = tree->left;   //attach former subtrees of removed element
      subtree->right = tree->right;
      if (subtree->left == subtree) //otherwise we would introduce a cycle
        subtree->left = 0;


      if (previous == 0)            //link it to predecessor of removed element
        set->tree = subtree;
      else if (previous->left == tree)
        previous->left = subtree;
      else
        previous->right = subtree;
    }

  free(tree);

  return;
}

previous = tree;

if (compresult < 0) //traversing the tree in search of the element.
  tree = tree->left;
```

```
        else
          tree = tree->right;
      }
}


int isElement(Set* set, void* element)
{
    Tree* tree = set->tree;
    int depth = 0;

    while (tree != 0)
      {
        if (compare(tree->data, element) == 0)
          return 1;
        else if (compare(tree->data, element) < 0)
          tree = tree->left;
        else
          tree = tree->right;
      }

    return 0;
}


void addTree(Tree* tree, Set* set)   //adds the contents of the tree to the set recursively
{
  if (tree != 0)
    {
      insertElement(set, tree->data);

      addTree(tree->left, set);
      addTree(tree->right, set);
    }
}

void addSet(Set* set1, Set* set2)
{
  addTree(set1->tree, set2);
}

int isSubsetTree(Tree* tree, Set* set)  //checks whether elements of the tree
        //are contained in the set
{
  if (tree != 0)
```

```
    {
      if (!isElement(set, tree->data))
        return 0;

      printf("tada\n");

      return (isSubsetTree(tree->left, set) && isSubsetTree(tree->right, set));
    }

  return 1;
}

int isSubset(Set* set1, Set* set2)
{
  return isSubsetTree(set1->tree, set2);
}


Set* copySet(Set* set2)  //copies a set (shallow copy)
{
  Set* newset = emptySet(set2->compare);
  addSet(newset, set2);
  return newset;
}

int sizeOf(Set* set)
{
  return set->size;
}


int compare(void* a, void* b)
{
  return -(*((int*)a) - *((int*)b));
}

void printDepth(char* text, Set* set, void* element) //help method for debugging
{
  int result = isElement(set, element);

  if (result)
    {
      printf(text);
      printf(" holds in depth ");
      printf("%i\n", result);
```

```
    }
  else
  {
    printf(text);
    printf(" does not hold\n");
  }
}

void printSubset(char* text, Set* set1, Set* set2)
{
  int result = isSubset(set1, set2);

  if (result)
    {
      printf(text);
      printf(" holds");
    }
  else
  {
    printf(text);
    printf(" does not hold\n");
  }
}


int drawTreeLayer(Tree* tree, int layer)  //draws parts of the tree that have same depth
{
  if (tree == 0)
    return 0;

  if (layer == 0)
  {
    printf("%i ", *(int*)tree->data);
    return 1;
  }

  return (drawTreeLayer(tree->left, layer-1) + drawTreeLayer(tree->right, layer-1));
}

void drawTree(Tree* tree, char* text)
{
  printf(text);
  printf("\n");
  int layer = 0;
  while (drawTreeLayer(tree, layer) != 0)
```

```
    {
      layer++;
      printf("\n");
    }
  printf("\n");
}

int main(int argc, char** argv)
{
  int a, b, c, d;
  a = 34;
  b = 344;
  c = 23;
  d = 333;

  Set* mySet, *mySet2;
  mySet = emptySet(&compare);
  mySet2 = emptySet(&compare);

  isSubset(mySet, mySet2);
  printf("tada\n");

  insertElement(mySet2, &d);

  insertElement(mySet, &a);
  insertElement(mySet, &a);
  insertElement(mySet, &c);
  insertElement(mySet, &b);

  drawTree(mySet->tree, "mySet->tree:");
  addSet(mySet, mySet2);
  drawTree(mySet2->tree, "mySet2->tree:");

  removeElement(mySet, &a);
  insertElement(mySet, &a);


  printDepth("a in mySet", mySet, &a);
  printDepth("b in mySet", mySet, &b);
  printDepth("c in mySet", mySet, &c);

  printSubset("mySet subset of mySet2", mySet, mySet2);
  printSubset("mySet2 subset of mySet", mySet2, mySet);
  printSubset("mySet subset of mySet", mySet, mySet);
  printSubset("mySet2 subset of mySet2", mySet2, mySet2);
```

```
  drawTree(mySet->tree, "mySet->tree:");

  srand(time());
  Set* randomSet = emptySet(&compare);  //creating random binary tree
  int i;
  for (i = 0; i < 1000; i++)
    {
      int* randomNumber;
      randomNumber = (int*)malloc(sizeof(int));
      *randomNumber = rand()/10000000;

      insertElement(randomSet, randomNumber);
    }
  drawTree(randomSet->tree, "randomSet->tree:");

  /*  if (isSubset(copySet(mySet), mySet))
    printf("mySet is subset of mySet\n");
  else
  printf("mySet is not subset of mySet\n");*/

  printf("Size of mySet: %i\n", sizeOf(mySet));
  printf("Size of mySet2: %i\n", sizeOf(mySet2));
  printf("Size of randomSet: %i\n", sizeOf(randomSet));

  free(mySet);
  free(mySet2);
  free(randomSet);
  return 1;
}
```

B.2 TVLA Analyses

B.2.1 List-based Implementation

Predicates

```
//////////////////
// Core Predicates

// For every program variable z there is a unary predicate that holds for
// list elements pointed by z.
// The unique property is used to convey the fact that the predicate can hold
// for at most one individual.
```

```
// The pointer property is a visualization hint for graphical renderers.
foreach (z in PVar) {
  %p z(v_1) unique pointer
}

// The predicate isSet is true for heap cells that represent sets
%p isSet(v)


// The predicate next represents the n field of the list data type.
%p n(v_1, v_2) function

// The predicate deq represents the equality of the data fields of the two list elements
%p deq(v_1, v_2) reflexive transitive symmetric

/////////////////////////////////////////////
// Instrumentation (i.e., derived) predicates

// The is[n] predicate holds for list elements pointed by two different
// list elements.
%i is[n](v) = E(v_1, v_2) (v_1 != v_2 & n(v_1, v) & n(v_2, v))

// The c[v] predicate holds for elements that reside on a cycle
// along the n field.
%i c[n](v) =  E(v_1) (n(v_1, v) & n*(v, v_1))

// The t[n] predicate records transitive reflexive reachability between
// list elements along the n field.
%i t[n](v_1, v_2) = n*(v_1, v_2) transitive reflexive

// Integrity constraints for transitive reachability
%r !t[n](v_1, v_2) ==> !n(v_1, v_2)
%r !t[n](v_1, v_2) ==> v_1 != v_2
%r E(v_1) (t[n](v_1, v_2) & t[n](v_1, v_3) & !t[n](v_2, v_3)) ==> t[n](v_3, v_2)

// For every program variable z the predicate r[n,z] holds for individual
// v when v is reachable from variable z along the n field (more formally,
// the corresponding list element is reachable from z).
foreach (z in PVar) {
  %i r[n,z](v) = E(v_1) (z(v_1) & t[n](v_1, v))

  %r (r[n,z](v_1) & r[n,z](v_2) & !t[n](v_1, v_2)) ==> t[n](v_2, v_1)
}
```

```
//The noeq[deq] predicate expresses that an element is different from all the
//other elements that can be reached by a sequence of next-pointers (forward or backward)
%i noeq[deq,n](v) = A(v_1) (((t[n](v_1, v) | t[n](v, v_1)) & v_1 != v)
        -> (!deq(v_1, v) & !deq(v, v_1)))

%r ((t[n](v_1,v_2) | t[n](v_2,v_1)) & v_1 != v_2 & noeq[deq,n](v_2)) ==> !deq(v_2, v_1)
%r ((t[n](v_1,v_2) | t[n](v_2,v_1)) & v_1 != v_2 & noeq[deq,n](v_2)) ==> !deq(v_1, v_2)
%r (t[n](v_1,v_2) & noeq[deq,n](v_2)) ==> noeq[deq,n](v_1)
%r (t[n](v_2,v_1) & noeq[deq,n](v_2)) ==> noeq[deq,n](v_1)
%r A(v)((t[n](v,v_1) & v != v_1) -> !deq(v,v_1)) ==> noeq[deq,n](v_1)
%r A(v)((t[n](v_1,v) & v != v_1) -> !deq(v,v_1)) ==> noeq[deq,n](v_1)
%r (noeq[deq,n](v) & deq(v_1, v) & v != v_1) ==> !t[n](v_1,v)

// The predicate validSet is true for heap cells that represent valid sets
%i validSet(v) = isSet(v) & noeq[deq,n](v)

//The binary predicate isElement expresses that v_1 is element of set v_2
%i isElement(v_1, v_2) = isSet(v_2) & E(v)(t[n](v_2,v) & deq(v_1,v) & v != v_2)

%r t[n](v,v_2) & isSet(v) & v != v_2 ==> isElement(v_2, v)


// The predicate or[n,z,l] is used to take a snapshot of the part of the
// heap reachable from pointer variable z via dereferences of field n
// when the program reaches the program label l.
// (See Copy_Reach_L in actions.tvp.)
foreach (z in PVar) {
    %p or[n,z](v)
}
```

Actions

```
%action uninterpreted() {
  %t "uninterpreted"
}

%action skip() {
  %t "skip"
}

%action Copy_Reach_L(lhs) {
  %t "storeReach(" + lhs + ")"
  {
    or[n,lhs](v) = r[n,lhs](v)
```

```
  }
}


////////////////////////////////////////////////////////////////////////////
// Actions for statements manipulating pointer variables and pointer fields

%action Set_Null_L(lhs) {
  %t lhs + " = NULL"
  {
    lhs(v) = 0
  }
}

%action Copy_Var_L(lhs, rhs) {
  %t lhs + " = " + rhs
  %f { rhs(v) }
  {
    lhs(v) = rhs(v)
  }
}

%action Malloc_L(lhs) {
  %t lhs + " = (L) malloc(sizeof(struct node)) "
  %new
  {
    lhs(v) = isNew(v)
    t[n](v_1, v_2) = (isNew(v_1) ? v_1 == v_2 : t[n](v_1, v_2))
    r[n, lhs](v) = isNew(v)
    foreach(z in PVar-{lhs}) {
      r[n,z](v) = r[n,z](v)
    }


    is[n](v) = is[n](v)
    c[n](v) = c[n](v)

    deq(v_1, v_2) = (isNew(v_1) & isNew(v_2)) |          //reflexive...
                    (v_1 != v_2  & (isNew(v_1) | isNew(v_2))? 1/2 : deq(v_1, v_2))

    noeq[deq,n](v) =  (isNew(v) ? 1 : noeq[deq,n](v))
    isElement(v_1,v_2) = (isNew(v_1) ? 1/2 : isElement(v_1,v_2))
    validSet(v) = (isNew(v) ? 0 : validSet(v))

  }
```

```
}

%action Free_L(lhs) {
  %t "free(" + lhs + ")"
  %f { lhs(v) }
  %message (E(v, v_1) lhs(v) & n(v, v_1)) ->
           "Internal Error! " + lhs + "->" + n + " != NULL"
  {
    c[n](v) = c[n](v)
    t[n](v_1, v_2) = t[n](v_1, v_2)
    r[n, lhs](v) = r[n, lhs](v)
        foreach(z in PVar) {
      r[n,z](v) = r[n,z](v)
        }
    is[n](v) = is[n](v)
    noeq[deq,n](v) = noeq[deq,n](v)
    isElement(v_1,v_2) = isElement(v_1,v_2)
    validSet(v) = validSet(v)
   }
  %retain !lhs(v)
}

%action Get_Next_L(lhs, rhs) {
  %t lhs + " = " + rhs + "->" + n
  %f { E(v_1, v_2) rhs(v_1) & n(v_1, v_2) &  t[n](v_2, v) }
  %message (!E(v) rhs(v)) ->
           "Illegal dereference to\n" + n + " component of " + rhs
  {
    lhs(v) = E(v_1) rhs(v_1) & n(v_1, v)
    r[n,lhs](v) = r[n,rhs](v) & (c[n](v) | !rhs(v))
  }
}

%action Set_Data_L(lhs, rhs) {
  %t lhs + "->data = " + rhs + "->data"
  %f { lhs(v) }
  %message (!E(v) rhs(v)) ->
           "Illegal dereference to\n" + data + " component of " + rhs
  %message (!E(v) lhs(v)) ->
           "Illegal dereference to\n" + data + " component of " + lhs
  {
    deq(v_1, v_2) = (lhs(v_1) & E(v)(rhs(v) & deq(v, v_2)))     //comp. x->data with s.th.
                  | (lhs(v_2) & E(v)(rhs(v) & deq(v_1, v)))     //comp. x->data with s.th.
                  | (!lhs(v_1) & !lhs(v_2) & deq(v_1, v_2))
  }
```

```
}

%action Set_Next_Null_L(lhs) {
  %t lhs + "->" + n + " = NULL"
  %f {
       lhs(v),
       // optimized change-formula for t[n] update-formula
       E(v_1, v_2) lhs(v_1) & n(v_1, v_2) & t[n](v_2, v)
     }
  %message (!E(v) lhs(v)) -> "Illegal dereference to\n" +
                             n + " component of " + lhs

  {
    n(v_1, v_2) = n(v_1, v_2) & !lhs(v_1)
    r[n,lhs](v) = lhs(v)
    foreach(z in PVar-{lhs}) {
      r[n,z](v) = (c[n](v) & r[n,lhs](v)?
                  z(v) | E(v_1) z(v_1) & TC (v_1, v) (v_3, v_4) (n(v_3, v_4) & !lhs(v_3)) :
                  r[n,z](v) & ! (E(v_1) r[n,z](v_1) & lhs(v_1) & r[n,lhs](v) & !lhs(v)))
    }
    c[n](v) = c[n](v) & ! (E( v_1) lhs(v_1) & c[n](v_1) & r[n,lhs](v))
  }
}

%action Set_Next_L(lhs, rhs) {
  %t lhs + "->" + n + " = " + rhs
  %f {
       lhs(v), rhs(v),
       // optimized change-formula for t[n] upate-formula
       E(v_4) rhs(v_4) & t[n](v_4, v_2)
     }
  %message (E(v_1, v_2) lhs(v_1) & n(v_1, v_2)) ->
          "Internal Error! " + lhs + "->" + n + " != NULL"
  %message (E(v_1, v_2) lhs(v_1) & rhs(v_2) & t[n](v_2, v_1)) ->
          "A cycle may be introduced\nby assignment " + lhs + "->" + n + "=" + rhs
  {
    n(v_1, v_2) = n(v_1, v_2) | lhs(v_1) & rhs(v_2)
    foreach(z in PVar) {
      r[n,z](v) = r[n,z](v) | E(v_1) r[n,z](v_1) & lhs(v_1) & r[n,rhs](v)
    }
    c[n](v) = c[n](v) | (E(v_1) lhs(v_1) & r[n,rhs](v_1) & r[n,rhs](v))
  }
}
```

```
/////////////////////////////////////////////////////////////////
// Actions needed to simulate program conditions involving pointer
// equality tests.

%action Is_Not_Null_Var(lhs) {
  %t lhs + " != NULL"
  %f { lhs(v) }
  %p E(v) lhs(v)
}

%action Is_Null_Var(lhs) {
  %t lhs + " == NULL"
  %f { lhs(v) }
  %p !(E(v) lhs(v))
}

%action Is_Eq_Var(lhs, rhs) {
  %t lhs + " == " + rhs
  %f { lhs(v), rhs(v) }
  %p A(v) lhs(v) <-> rhs(v)
}

%action Is_Not_Eq_Var(lhs, rhs) {
  %t lhs + " != " + rhs
  %f { lhs(v), rhs(v) }
  %p !A(v) lhs(v) <-> rhs(v)
}

/////////////////////////////////////////////////////////////////
// Actions needed to simulate program conditions involving comparisons
// of data elements.

%action Data_Eq(lhs, rhs) {
  %t lhs + ".data == " + rhs + ".data"
  %f { lhs(v_1) & rhs(v_2) & deq(v_1, v_2) }
  %p E(v_1, v_2) (lhs(v_1) & rhs(v_2) & deq(v_1, v_2))
  {
   // deq(v_1,v_2) = (((lhs(v_1) & rhs(v_2)) | (rhs(v_1) & lhs(v_2))) ? 1 : deq(v_1, v_2))
   deq(v_1, v_2) = ((lhs(v_1) & rhs(v_2)) | (rhs(v_1) & lhs(v_2)))
                 | (lhs(v_1) & deq(v_1,v_2) & E(v)(rhs(v) & deq(v,v_2)))
                 | (lhs(v_2) & deq(v_1,v_2) & E(v)(rhs(v) & deq(v_1,v)))
                 | (rhs(v_1) & deq(v_1,v_2) & E(v)(lhs(v) & deq(v,v_2)))
                 | (rhs(v_2) & deq(v_1,v_2) & E(v)(lhs(v) & deq(v_1,v)))
                 | (v_1 == v_2)
  }
```

```
}

%action Data_Not_Eq(lhs, rhs) {
  %t lhs + ".data != " + rhs + ".data"
  %f { lhs(v_1) & rhs(v_2) & deq(v_1, v_2) }
  %p !E(v_1, v_2) (lhs(v_1) & rhs(v_2) & deq(v_1, v_2))
  {
    deq(v_1,v_2) = (((lhs(v_1) & rhs(v_2)) | (rhs(v_1) & lhs(v_2))) ? 0 : deq(v_1, v_2))
  }
}

/////////////////////////////////////////
// Actions for testing various properties

%action Assert_ListInvariants(lhs) {
  %t "AssertListInvariants(" + lhs + ")"
  %f { lhs(v) }
  %p E(v)(r[n,lhs](v) & (c[n](v) | !noeq[deq,n](v)))
  %message ( E(v)(r[n,lhs](v) & (c[n](v) | !noeq[deq,n](v))) ) ->
           "The list pointed by " + lhs + " may be cyclic or may contain duplicates!"
}

%action Assert_No_Leak(lhs) {
  %t "assertNoLeak(" + lhs + ")"
  %f { lhs(v) }
  %p E(v) !r[n,lhs](v) & !(E(v1) element(v1) & deq(v, v1))
  %message ( E(v) !r[n,lhs](v) & !(E(v1)element(v1) & deq(v, v1)) ) -> //only the element
                                //that is to be inserted/removed should not be reachable.
           "There may be a list element not reachable from variable " + lhs + "!"
}

%action Assert_Permutation_L(lhs) {
  %t "AssertPermutation(" + lhs + ")"
  %p !(A(v) (newList(v) | E(v1)(element(v1) & deq(v, v1))
          | (r[n,lhs](v) <-> or[n,lhs](v))))
        //either it used to be here before or it is the newly inserted element
  %message !(A(v) (newList(v) | E(v1)(element(v1) & deq(v, v1)) | (r[n,lhs](v)
                                 <-> or[n,lhs](v)))) ->
           "Unable to prove that the list pointed-to by " + lhs +
           "is a permutation of the original list "
}

%action Assert_Element_Removed(set, element) {
  %t "AssertElementRemoved(" + set + ", " + element + ")"
  %p E(vel, vset)(element(vel) & set(vset) & isElement(vel, vset))
```

```
  %message (E(vel, vset)(element(vel) & set(vset) & isElement(vel, vset))) ->
        "Element " + element + " has not been removed from set " + set + "."
}

%action Assert_Element_Inserted(set, element) {
  %t "AssertElementInserted(" + set + ", " + element + ")"
  %p E(vel, vset)(element(vel) & set(vset) & !isElement(vel, vset))
  %message (E(vel, vset)(element(vel) & set(vset) & !isElement(vel, vset))) ->
        "Element " + element + " has not been inserted into set " + set + "."
}

%action Is_Not_Element(element, set) {
  %t "Is_Not_Element(" + set + ", " + element + ")"
  %p !E(vel, vset)(element(vel) & set(vset) & isElement(vel, vset))
  %message (!E(vel, vset)(element(vel) & set(vset) & isElement(vel, vset))) ->
        "Element " + element + " is not element of set " + set + "."
}

%action Is_Element(element, set) {
  %t "Is_Element(" + set + ", " + element + ")"
  %p E(vel, vset)(element(vel) & set(vset) & isElement(vel, vset))
  %message (E(vel, vset)(element(vel) & set(vset) & isElement(vel, vset))) ->
        "Element " + element + " is element of set " + set + "."
}
```

Input Structures

```
// An empty list (x points to NULL).
%n = {setstart, el}
%p = {
   deq = {el->el, setstart->setstart}
   noeq[deq, n] = {setstart, el}
   set = {setstart}
   element = {el}
   t[n] = {setstart->setstart, el->el}
   r[n,set] = {setstart}
   r[n,element] = {el}

   isSet = {setstart}
   validSet = {setstart}
}

// An acyclic singly-linked list with a single element pointed by set.
%n = {setstart, head, el}
```

```
%p = {
       n = {setstart->head}
       deq = {setstart->setstart, head->head, el->el, head->el:1/2, el->head:1/2}
       noeq[deq,n] = {setstart, head, el}
       set = {setstart}
       element = {el}
       t[n] = {el->el, setstart->setstart, setstart->head, head->head}
       r[n,set] = {setstart, head}
       r[n,element] = {el}

       isSet = {setstart}
       validSet = {setstart}
       isElement = {head->setstart, el->setstart:1/2}
}

// An acyclic singly-linked list with two or more elements pointed by program set.
%n = {setstart, head, tail, el}
%p = {
       sm = {tail:1/2}
       n = {setstart->head, head->tail:1/2, tail->tail:1/2}
       deq = {el->el, setstart->setstart, head->head, tail->tail:1/2 , el->head:1/2,
                       head->el:1/2, el->tail:1/2, tail->el:1/2}
       noeq[deq,n] = {setstart, head, tail, el}
       set = {setstart}
       element = {el}
       t[n] = {el->el, setstart->setstart, setstart->head, setstart->tail, head->head,
                       head->tail, tail->tail:1/2}
       r[n,set] = {setstart, head, tail}
       r[n,element] = {el}

       isSet = {setstart}
       validSet = {setstart}
       isElement = {head->setstart, tail->setstart, el->setstart:1/2}
}
```

Insertion

```
/*

#include <stdio.h>

#include "set.h"

void insertElement(Set* set, void* element)
{
```

```
  List* list = set->list;
  List* prev = 0;

  while (list != 0)
    {
      if (compare(list->data, element) == 0)
        return;

      prev = list;
      list = list->next;
    }

  List* newList = (List*)malloc(sizeof(List));
  newList->data = element;
  newList->next = 0;
  set->size++;

  if (prev == 0) //list is empty
    {
      set->list = newList;
    }
  else //append item to list
    {
      prev->next = newList;
    }
}
*/
///////
// Sets

%s PVar {set, list, prev, newList, element, temp}

#include "predicates.tvp"

%%

#include "actions.tvp"

%%

/////////////////////////////////////////////////////////////////////////////
// Transition system for a function that creates an element with a specified
// value and inserts it at the end of the list if it is not already contained in the list.
```

```
//including data field...
L0  Copy_Reach_L(set)             L1
L1  Get_Next_L(list, set)         L2          // List* list = set->list;
L2  Set_Null_L(prev)              L3          // List* prev = 0;

L3  Is_Not_Null_Var(list)         L4          // while (list != 0)
L3  Is_Null_Var(list)             L12         //
L4  Data_Eq(list, element)        exit        // if (compare(list, element) == 0)
L4  Data_Not_Eq(list, element)    L8          //
L8  Copy_Var_L(prev, list)        L9          // prev = list;
L9  Get_Next_L(temp, list)        L10         // temp = list->next;
L10 Copy_Var_L(list, temp)        L11         // list = temp;
L11 Set_Null_L(temp)              L3          // temp = 0;

L12 Malloc_L(newList)             L13     // List* newList = (List*)malloc(sizeof(List));
L13 Set_Data_L(newList, element)  L14         // newList->data = element;
L14 Set_Next_Null_L(newList)      L15         // newList->next = 0;
                                              //  set->size++;
L15 Is_Null_Var(prev)                L16         //  if (prev == 0) //list is empty
L15 Is_Not_Null_Var(prev)            L17         //    else //append item to list
L16 Set_Next_L(set, newList)         exit        // set->list = newList;
L17 Set_Next_L(prev, newList)        exit        // prev->next = newList;

exit  Set_Null_L(prev)               exit2
exit2 Set_Null_L(list)               exitfinal

exitfinal Assert_Permutation_L(set)            error
exitfinal Assert_ListInvariants(list)          error
exitfinal Assert_No_Leak(set)                  error
exitfinal Assert_Element_Inserted(set, element) error

%% L0, exitfinal, error
```

Removal

```
/*


void* removeElement(Set* set, void* element)
{
  List* list = set->list;
  List* prev = 0;
  List* temp;
```

```
  while (list != 0)
    {
      if (compare(list->data, element) == 0)
      {
        set->size--;
        void* deletedElement = list->data;
        if (prev == 0)
                set->list = list->next;
        else
                prev->next = list->next;
        free(list);
        return deletedElement;
      }

      prev = list;
      list = list->next;
    }
}
*/
///////
// Sets

%s PVar {set, list, prev, element, newList, temp}

#include "predicates.tvp"

%%

#include "actions.tvp"

%%

//////////////////////////////////////////////////////////////////////////////
// Transition system for a function that creates an element with a specified
// value and inserts it at the end of the list if it is not already contained in the list.

L0  Copy_Reach_L(set)               L1
L1  Get_Next_L(list, set)           L2           // List* list = set->list;
L2  Set_Null_L(prev)                L3           // List* prev = 0;

L3  Is_Not_Null_Var(list)           L4           // while (list != 0)
L3  Is_Null_Var(list)               exit         //
L4  Data_Eq(list, element)          L5           // if (compare(list, element) == 0)
L4  Data_Not_Eq(list, element)      L14          //
```

```
L5  Get_Next_L(temp, list)         L6         //temp = list->next;
L6  Is_Null_Var(prev)              L7         //if (prev == 0)
L6  Is_Not_Null_Var(prev)          L9         //else
L7  Set_Next_Null_L(set)           L8         //set->list = 0;
L8  Set_Next_L(set, temp)          L11        //set->list = temp (==list->next);
L9  Set_Next_Null_L(prev)          L10        //prev->next = 0;
L10 Set_Next_L(prev, temp)         L11        //prev->next = temp (==list->next);
L11 Set_Null_L(temp)               L12        //temp = 0;
L12 Set_Next_Null_L(list)          L13        //list->next = 0;
L13 skip()                         exit       //free(list) omitted for demonstration purpose

L14 Copy_Var_L(prev, list)         L15        // prev = list;
L15 Get_Next_L(temp, list)         L16        // temp = list->next;
L16 Copy_Var_L(list, temp)         L17        // list = temp;
L17 Set_Null_L(temp)               L3          // temp = 0;

exit  Set_Null_L(prev)                 exit2
exit2 Set_Null_L(list)                 exitfinal


exitfinal Assert_Permutation_L(set)            error
exitfinal Assert_ListInvariants(list)          error
exitfinal Assert_No_Leak(set)                  error
exitfinal Assert_Element_Removed(set, element) error

%% L0, exitfinal, error
```

Membership Test

```
///////
// Sets

%s PVar {set, list, element}

#include "predicates.tvp"

%%

#include "actions.tvp"

%%
/*
int isElement(Set* set, void* element)
{
    List* list = set->list;
```

```
    while (list != 0)
    {
        if (compare(list->data, element) == 0)
            return 1;

        list = list->next;
    }

    return 0;
}
*/


//including data field...
L0      Copy_Reach_L(set)               L1
L1      Get_Next_L(list, set)           L2            // List* list = set->list;

L2      Is_Not_Null_Var(list)           L3            // while (list != 0)
L2      Is_Null_Var(list)               exitnotfound  //
L3      Data_Eq(list, element)          exitfound     // if (compare(list, element) == 0)
L3      Data_Not_Eq(list, element)      L4            // (else)
L4      Get_Next_L(list, list)          L2            // list = list->next;

exitfound    Is_Not_Element(element, set)      exitfounderror
exitnotfound Is_Element(element, set)          exitnotfounderror


%% L0, exitfound, exitnotfound, exitfounderror, exitnotfounderror
```

B.2.2 Tree-based Implementation

Predicates

```
#include "pred_tree.tvp"


/**********************************************/
/*************** Core Predicates *************/

%p dle(v_1, v_2) transitive reflexive

%p isSet(v)

/**********************************************************/
/****************** Instrumentation Predicates *************/
```

```
%i cmp[dle,left](v_1, v_2) = dle(v_2, v_1) & !dle(v_1, v_2) {}
%i cmp[dle,right](v_1, v_2) = dle(v_1, v_2) & !dle(v_2, v_1) {}

foreach (x in TRVar) {
    %i dle[x,left](v) = E(v1) (x(v1) & dle(v, v1) & !dle(v1, v))
    %i dle[x,right](v) = E(v1) (x(v1) & !dle(v, v1) & dle(v1, v))
}

%i inOrder[dle]()  = A(v2, v4)(downStar[left](v2, v4)) -> (dle(v4, v2) & !dle(v2, v4))
                & A(v2, v4)(downStar[right](v2, v4)) -> (dle(v2, v4) & !dle(v4, v2))  {}


//v1 is element of set v2
%i isElement(v1, v2) = isSet(v2) & E(vequal)(downStar(v2, vequal)
        & dle(vequal, v1) & dle(v1, vequal) & vequal != v2)


/*************************************************/
/**************** Consistency Rules **************/
%r !dle(v_1, v_2) ==> dle(v_2, v_1)

foreach (x in TRVar) {
    %r dle[x,left](v) & x(v1) ==> !dle(v1, v)
    %r dle[x,right](v) & x(v1) ==> !dle(v, v1)
}

/*%r !deq(v1, v2) & dle(v1, v2) ==> !dle(v2, v1)

%r E(v) deq(v1, v) & !dle(v, v2) ==> !dle(v1, v2)
%r E(v) deq(v1, v) & !dle(v2, v) ==> !dle(v2, v1)*/

%r dle(v1, v2) & dle(v2, v1) & dle(v1, v3) ==> dle(v2, v3)
%r dle(v1, v2) & dle(v2, v1) & dle(v3, v1) ==> dle(v3, v2)

%r E(v1)(!dle(v, v1) & dle(v2, v1)) ==> !dle(v, v2)
%r E(v1)(!dle(v1, v) & dle(v1, v2)) ==> !dle(v2, v)

%r isSet(v) & downStar(v, v1) & v1 != v ==> isElement(v1, v)

foreach (x in {element}) {
    %r dle[x,left](v) & x(v1) ==> dle(v, v1)
    %r dle[x,left](v) & x(v1) ==> !dle(v1, v)
    %r dle[x,right](v) & x(v1) ==> !dle(v, v1)
    %r dle[x,right](v) & x(v1) ==> dle(v1, v)
}
```

```
%r E(v1)(dle(v1, v2) & dle(v2, v1) & !dle(v1, v3)) ==> !dle(v2, v3)

%r inOrder[dle]() & downStar[right](v2, v4) ==> !dle(v4, v2)
%r inOrder[dle]() & downStar[left](v2, v4) ==> !dle(v2, v4)

//%r inOrder[dle]() & r[set](v1) & r[set](v2) & v1 != v2 ==> !deq(v1, v2)

%r inOrder[dle]() & E(v2) !dle(vel, v1) & downStar[left](v1, v2) ==> !dle(vel, v2)
%r inOrder[dle]() & E(v2) !dle(v1, vel) & downStar[right](v1, v2) ==> !dle(v2, vel)

%r treeNess() & downStar(v, v1) & !downStar[left](v, v1) & v != v1
        ==> downStar[right](v, v1)
%r treeNess() & downStar(v, v1) & !downStar[right](v, v1) & v != v1
        ==> downStar[left](v, v1)

//////////////////
// Core Predicates

// For every program variable z there is a unary predicate that holds for
// list elements pointed by z.
// The unique property is used to convey the fact that the predicate can hold
// for at most one individual.
// The pointer property is a visualization hint for graphical renderers.
foreach (z in PVar) {
    %p z(v_1) unique pointer
}

// For every field there is a corresponding binary predicate.
foreach (sel in TSel) {
  %p sel(v_1, v_2) function {}
}

// This predicate stores the original reachability of nodes.
foreach (z in PVar) {
    %p or[z](v)
}

/////////////////////////////
// Instrumentation Predicates

// The down predicate represents the union of selector predicates.
%i down(v1, v2) = |/{ sel(v1, v2) : sel in TSel } {}

// The downStar predicate records reflexive transitive reachability
// between tree nodes along the union of the selector fields.
```

```
%i downStar(v1, v2) = down*(v1, v2) transitive reflexive {}

foreach (sel in TSel) {
        %i downStar[sel](v1, v2) = E(v)(sel(v1, v) & down*(v, v2))         transitive
}

// For every program variable z the predicate r[z] holds for individual
// v when v is reachable from variable z along the selector fields.
foreach (x in PVar) {
  %i r[x](v) = E(v1) (x(v1) & downStar(v1, v))
}

%i treeNess() = A(v1, v2, v)((downStar[left](v,v1) & downStar[right](v,v2))
                        -> (!downStar(v1, v2) & !downStar(v2, v1))) {}




///////////////////////////////////////////////////////////////
// Additional integrity constraints

// down predicate
foreach (sel in TSel) {
  %r !down(v_1, v_2) ==> !sel(v_1, v_2)
}

// Binary reachability (downStar predicate)
%r !downStar(v_1, v_2) ==> !down(v_1, v_2)
%r (E(v_1) downStar(v_1, v_2) & !downStar(v_1, v_3)) ==> !downStar(v_2, v_3)

// Unary reachability (r[z] predicates)
foreach (x in PVar) {
      %r r[x](v_1) & !r[x](v_2) ==> !downStar(v_1, v_2)
      %r r[x](v_1) & !r[x](v_2) ==> !down(v_1, v_2)
}

// The treeness conditions
foreach (sel in TSel) {
 foreach (complementSel in TSel- {sel}) {
  // %r (E(v_1, v_2, v_3) sel(v_1, v_2)& complementSel(v_1, v_3) &
  //                      downStar(v_2, v_4) & downStar(v_3, v_5)) ==> v_4 != v_5
  // commented-out for efficiency
  // Useful consequences of the above rule which TVLA did not generate
/*  %r (E(v_2, v_4) treeNess() & sel(v_1, v_2) & downStar(v_2, v_4) & downStar(v_3, v_4))
    ==> !complementSel(v_1, v_3)
```

```
  %r (E(v_2) treeNess() & sel(v_1, v_2) & downStar(v_2, v_3))
    ==> !complementSel(v_1, v_3)*/

  %r (E(v_4) treeNess() & downStar[sel](v_1, v_4) & downStar(v_3, v_4))
    ==> !downStar[complementSel](v_1, v_3)

  %r treeNess() & downStar[sel](v1, v2) & downStar[complementSel](v1, v3) ==> v2 != v3

  %r treeNess() & sel(v_1, v_2) ==> !complementSel(v_1, v_2)
  %r treeNess() & downStar[sel](v_1, v_2) ==> !downStar[complementSel](v_1, v_2)
  }
%r (E(v_1, v_2) treeNess() & sel(v_1, v_2) & downStar(v_2, v_3) &
                downStar(v_1, v_4) & v_4 != v_1 & !downStar(v_2, v_4))
    ==> !downStar(v_4, v_3)
%r (E(v_1, v_2) treeNess() & sel(v_1, v_2) & downStar(v_2, v_3) &
                downStar(v_1, v_4) & v_4 != v_1 & !downStar(v_2, v_4))
     ==> !downStar(v_3, v_4)
}

// consequences of the acyclicity assumption
%r downStar(v_1, v_2) ==> !down(v_2, v_1)
foreach (sel in TSel) {
  %r sel(v_1, v_2) ==> !downStar(v_2, v_1)
  foreach (complementSel in TSel- {sel}) {
    %r sel(v_1, v_2) ==> !complementSel(v_2, v_1)
  }
}
```

Actions

```
// Binary-search Tree Actions

%action uninterpreted() {
  %t "uninterpreted"
}

%action skip() {
  %t "skip"
}

%action Copy_Reach_T(lhs) {
  %t "storeReach(" + lhs + ")"
  {
    or[lhs](v) = r[lhs](v)
  }
```

```
}


///////////////////////////////////////////////////////////
// Actions encoding program statements that involve boolean
// program variables.

%action Is_True(x1) {
    %t x1
    %p x1()
}

%action Is_False(x1) {
    %t "!" + x1
    %p !x1()
}

%action Set_True(x1) {
    %t x1 + " = true"
    {
        x1() = 1
    }
}

%action Set_False(x1) {
    %t x1 + " = false"
    {
        x1() = 0
    }
}


//////////////////////////////////////////////////////////////////
// Actions encoding program conditions involving pointer equality.

%action Is_Not_Null_Var(x1) {
    %t x1 + " != null"
    %f { x1(v) }
    %p E(v) x1(v)
}

%action Is_Null_Var(x1) {
    %t x1 + " == null"
    %f { x1(v) }
    %p !(E(v) x1(v))
```

```
}

%action Is_Eq_Var(x1, x2) {
    %t x1 + " == " + x2
    %f { x1(v), x2(v) }
    %p A(v) x1(v) <-> x2(v)
}

%action Is_Not_Eq_Var(x1, x2) {
    %t x1 + " != " + x2
    %f { x1(v), x2(v) }
    %p !A(v) x1(v) <-> x2(v)
}


///////////////////////////////////////////////////////////////
// Actions encoding program statements that involve comparisons
// of the data fields.

%action Greater_Data_T(x1, x2) {
    %t x1 + "->data > " + x2 + "->data"
    %f { x1(v_1) & x2(v_2) & dle(v_1, v_2) }
    %p !E(v_1, v_2) x1(v_1) & x2(v_2) & dle(v_1, v_2)
}

%action Less_Equal_Data_T(x1, x2) {
    %t x1 + "->data <= " + x2 + "->data"
    %f { x1(v_1) & x2(v_2) & dle(v_1, v_2) }
    %p E(v_1, v_2) x1(v_1) & x2(v_2) & dle(v_1, v_2)
}

%action Greater_Equal_Data_T(x1, x2) {
    %t x1 + "->data >= " + x2 + "->data"
    %f { x1(v_1) & x2(v_2) & dle(v_2, v_1) }
    %p E(v_1, v_2) x1(v_1) & x2(v_2) & dle(v_2, v_1)
}

%action Less_Data_T(x1, x2) {
    %t x1 + "->data < " + x2 + "->data"
    %f { x1(v_1) & x2(v_2) & dle(v_2, v_1) }
    %p !E(v_1, v_2) x1(v_1) & x2(v_2) & dle(v_2, v_1)
}

%action Equal_Data_T(x1, x2) {
    %t x1 + "->data == " + x2 + "->data"
```

```
    %f { x1(v_1) & x2(v_2) & dle(v_2, v_1) & dle(v_1, v_2) }
    %p E(v_1, v_2) x1(v_1) & x2(v_2) & dle(v_2, v_1) & dle(v_1, v_2)
}

%action Not_Equal_Data_T(x1, x2) {
    %t x1 + "->data != " + x2 + "->data"
    %f { x1(v_1) & x2(v_2) & dle(v_2, v_1) & dle(v_1, v_2) }
    %p !E(v_1, v_2) x1(v_1) & x2(v_2) & dle(v_2, v_1) & dle(v_1, v_2)
}

//////////////////////////////////////////////////////////////
// Actions encoding program statements that manipulate pointer
// variables and pointer fields.

// x1 = (Tree) malloc(sizeof(struct node))
%action Malloc_T(x1) {
    %t x1 + " = (Tree) malloc(sizeof(struct node)) "
    %new
    {
        x1(v) = isNew(v)
        r[x1](v) = isNew(v)
        foreach (x in PVar-{x1}) {
           r[x](v) = (isNew(v) ? 0 : r[x](v))
        }
        down(v1, v2) = ((isNew(v1) | isNew(v2)) ? 0 : down(v1, v2))
        downStar(v1, v2) = downStar(v1, v2) | (isNew(v1) & v1 == v2)
        foreach (sel in TSel) {
                downStar[sel](v1, v2) = downStar[sel](v1, v2)
        }
        dle(v_1, v_2) =
                (v_1 == v_2 ) |
                (v_1 != v_2  & (isNew(v_1) | isNew(v_2))? 1/2: dle(v_1, v_2))
        foreach (var in TRVar-{x1}) {
                dle[var, left](v) = (isNew(v) ? 1/2 : dle[var, left](v))
                dle[var, right](v) = (isNew(v) ? 1/2 : dle[var, right](v))
        }
        foreach (var in TRVar - (TRVar-{x1})) {
                dle[x1, left](v) = (isNew(v) ? 0 : 1/2)
                dle[x1, right](v) = (isNew(v) ? 0 : 1/2)
        }

        foreach (sel in TSel) {
                cmp[dle,sel](v_1, v_2) =
                     !(isNew(v_1) & isNew(v_2)) &
                     (v_1 != v_2  & (isNew(v_1) | isNew(v_2))? 1/2: cmp[dle,sel](v_1, v_2))
```

```
                sel(v1, v2) = ((isNew(v1) | isNew(v2)) ? 0 : sel(v1, v2))
        }
        isElement(v1, v2) = (isNew(v2) ? 0 : (isNew(v1) & isSet(v2) ? 1/2
: isElement(v1, v2)))

        inOrder[dle]() = inOrder[dle]()
        treeNess() = treeNess()
    }
}

// x1 = NULL
%action Set_Null_T(x1) {
    %t x1 + " =(T) NULL"
    {
        x1(v) = 0
        r[x1](v) = 0

        inOrder[dle]() = inOrder[dle]()
        treeNess() = treeNess()
    }
}

// x1 = x2
%action Copy_Var_T(x1, x2) {
    %t x1 + " = (T)" + x2
    %f { x2(v), r[x2](v) }
    {
        x1(v) = x2(v)
        r[x1](v) = r[x2](v)

        inOrder[dle]() = inOrder[dle]()
        treeNess() = treeNess()
    }
}

// x1 = x2->sel
%action Get_Sel_T(x1, x2, sel) {
        %t x1 + " = (T)" + x2 + "->" + sel
        %f {
               E(v_1, v_2) x2(v_1) & sel(v_1, v_2) & downStar(v_2, v),
               E(v_1) x2(v_1) & left(v_1, v),
               E(v_1) x2(v_1) & right(v_1, v)
      }
      %message !(E(v) x2(v)) -> "a possibly illegal dereference to ->" + sel
```

```
                            + " component of " + x2 + "\n"
        {
                x1(v) = E(v1) x2(v1) & sel(v1, v)
                r[x1](v) = E(v_1,v_2) x2(v_1) & sel(v_1, v_2) &
                              downStar(v_2, v)
                inOrder[dle]() = inOrder[dle]()
                treeNess() = treeNess()
        }

}

// x1->sel = NULL
%action Set_Sel_Null_T(x1, sel) {
    %t x1 + "->" + sel + " = (T) NULL"
    %f { x1(v), // change-formula for sel(v_1, v_2)
        E(v_1) x1(v_1) & sel(v_1, v_2),
        E(v_1, v_2) x1(v_1) & sel(v_1, v_2) & downStar(v_2, v)
                        // for reachability and downStar

        }
    %message !(E(v) x1(v)) -> "a possibly illegal dereference to ->" + sel
                                  + " component of " + x1 + "\n"
    {
        sel(v_1, v_2) = sel(v_1, v_2) & !x1(v_1)
        down(v_1, v_2) = ((x1(v_1) & sel(v_1, v_2)) ? 0 : down(v_1, v_2))
        downStar(v_1, v_2) =
           ((downStar(v_1, v_2) &
           E(v_3, v_4) downStar(v_1, v_3) & x1(v_3) & sel(v_3, v_4) & downStar(v_4, v_2))?
                0 : downStar(v_1, v_2))

        foreach (s in TSel - {sel}) {
                downStar[s](v_1, v_2) = ((downStar[s](v_1, v_2) &
                           E(v_3, v_4) downStar[s](v_1, v_3) & x1(v_3) & sel(v_3, v_4)
                                   & downStar(v_4, v_2)) ? 0 : downStar[s](v_1, v_2))
        }

        downStar[sel](v_1, v_2) = ((downStar[sel](v_1, v_2) &
             E(v_3, v_4) (downStar[sel](v_1, v_3) | v_1 == v_3) & x1(v_3) & sel(v_3, v_4)
                           & downStar(v_4, v_2)) ? 0 : downStar[sel](v_1, v_2))



        r[x1](v) = r[x1](v) & !(E(v_1, v_2) x1(v_1) & sel(v_1, v_2) & downStar(v_2, v))
        foreach (x2 in PVar - {x1}) {
                r[x2](v) = r[x2](v) & !(E(v_1, v_2)(x1(v_1) & r[x2](v_1) &
```

```
                                        sel(v_1, v_2) & downStar(v_2, v)))
        }
        inOrder[dle]() = inOrder[dle]()
        treeNess() = treeNess()
     }
}

// assert(x1->sel==NULL); x1->sel = x2
%action Set_Sel_T(x1, sel, x2) {
    %t x1 + "->" + sel + " = (T)" + x2
    %f { x1(v), x2(v),
        E(v_4) x2(v_4) &  downStar(v_4, v_2)
     }
    %message !(E(v) x1(v)) -> "a possibly illegal dereference to ->" + sel
                + " component of " + x1 + "\n"
    %message (E(v_1, v_2) x1(v_1) & sel(v_1, v_2)) -> "an internal error assuming "
                + x1 + "->" + sel + "==NULL\n"

    // Checks for creation of a cycle.
    %message (E(v_1, v_2)
                x1(v_1) & x2(v_2) & downStar(v_2, v_1)) ->
       "A cycle may be introduced by assignment " + x1 + "->" + sel + "=" + x2 + "\n"

    {
        sel(v_1, v_2) = sel(v_1, v_2) |  x1(v_1) & x2(v_2)
        down(v_1, v_2) = down(v_1, v_2) | x1(v_1) & x2(v_2)

        foreach (x3 in PVar) {
                r[x3](v) = r[x3](v) | E(v_1) x1(v_1) & r[x3](v_1) & r[x2](v)
        }

        treeNess() = treeNess() & !E(v1, v2)(x2(v2) & downStar(v1, v2) & v1 != v2)

        downStar(v_1, v_2) =
           (E(v_3, v_4) x1(v_3) & x2(v_4) & downStar(v_1, v_3) &
            downStar(v_4, v_2) ? 1: downStar(v_1, v_2))
        foreach (s in TSel - {sel}) {
                downStar[s](v_1, v_2) =
    (E(v_3, v_4) x1(v_3) & x2(v_4) & downStar[s](v_1, v_3) &
                                downStar(v_4, v_2) ? 1: downStar[s](v_1, v_2))
        }
        downStar[sel](v_1, v_2) =
                (E(v_3, v_4) x1(v_3) & x2(v_4) & (downStar[sel](v_1, v_3) | v_1 == v_3) &
                        downStar(v_4, v_2) ? 1: downStar[sel](v_1, v_2))
```

```
        inOrder[dle]() = inOrder[dle]() & A(v1, v2, v3)((x1(v1) & x2(v2) & downStar(v2, v3))
             -> (cmp[dle, sel](v1,v3)
                        & A(v)(   (downStar[left](v, v1) -> (dle(v3, v) & !dle(v, v3)))
                                & (downStar[right](v, v1) -> (!dle(v3, v) & dle(v, v3))))))
    }
}

// free(x1)
%action Free_T(x1) {
    %t "free(" + x1 + ") "
    %f { x1(v) }
    %message (E(v_1, v_2) x1(v_1) & (|/{ sel(v_1, v_2) : sel in TSel })) ->
        "Internal Error! assume that the selectors of " + x1 + "are all NULL"
    %retain !x1(v)
}

%action Set_Data_T(lhs, rhs) {
  %t lhs + "->data = " + rhs + "->data"
  %f { lhs(v) }
  %message (!E(v) rhs(v)) ->
           "Illegal dereference to\n" + data + " component of " + rhs
  %message (!E(v) lhs(v)) ->
           "Illegal dereference to\n" + data + " component of " + lhs
  {
            dle(v_1, v_2) = (lhs(v_1) & E(v)(rhs(v) & dle(v, v_2)))     //comp. x->data with
                          | (lhs(v_2) & E(v)(rhs(v) & dle(v_1, v)))     //comp. x->data with
                          | (!lhs(v_1) & !lhs(v_2) & dle(v_1, v_2))

        inOrder[dle]() = inOrder[dle]() & A(v1, v2)((lhs(v1) & rhs(v2)) -> (
                          A(v3)(downStar[left](v1, v3) -> (dle(v3, v2) & !dle(v2, v3)))
                //nodes reachable from lhs are in order with the new value of lhs which is in
                        & A(v3)(downStar[right](v1, v3) -> (!dle(v3, v2) & dle(v2, v3)))
                        & A(v)((downStar[left](v, v1) -> (dle(v2, v) & !dle(v, v2)))
                //nodes reaching lhs are still in order
& (downStar[right](v, v1) -> (!dle(v2, v) & dle(v, v2))))))

        treeNess() = treeNess()
  }
}


//////////////////////////////////////////
// Actions for testing various properties.

%action Is_Sorted_Data_T() {
```

```
    %t "Is Data in tree " + root + " in ascending order?"
    %p A(v) inOrder[dle]()
}

%action Is_Not_Sorted_Data_T() {
    %t "Is Data in tree NOT in ascending order?"
    %p !inOrder[dle]()
    %message !inOrder[dle]() ->
            "Unable to prove that the tree is still in order"
}

%action Is_Element(el, s) {
    %t "Is " + el + " an element of set " + s + "?"
    %p E(v1, v2)(el(v1) & s(v2) & isElement(v1,v2))
      %message (E(v1, v2)(el(v1) & s(v2) & isElement(v1,v2)))  ->
           "Unable to prove that " + el +
           " is not an element of " + set
}

%action Is_Not_Element(el, s) {
    %t "Is " + el + " not an element of set " + s + "?"
    %p !(E(v1, v2)(el(v1) & s(v2) & isElement(v1,v2)))
  %message !(E(v1, v2)(el(v1) & s(v2) & isElement(v1,v2)))  ->
           "Unable to prove that " + el +
           " is an element of " + set
}

%action Assert_Permutation_T(set, element) {
  %t "AssertPermutation(" + set + ", " + element + ")"
  %p !(A(v) ((E(vel) dle(v, vel) & dle(vel, v) & element(vel))
          | (r[set](v) <-> or[set](v))))
  %message !(A(v) ((E(vel) dle(v, vel) & dle(vel, v) & element(vel))
          | (r[set](v) <-> or[set](v)))) ->
           "Unable to prove that the tree pointed-to by " + set +
           " is a permutation of the original tree "
}
```

Input Structures

```
//an empty set
%n = {setstart, el}
%p = {
        set = {setstart}
        element = {el}
```

```
        downStar = {el->el, setstart->setstart}
        r[set] = {setstart}
        r[element] = {el}

        inOrder[dle] = 1

        treeNess = 1

        dle = {setstart->setstart, el->setstart, el->el}

        dle[element, right] = {setstart}

        cmp[dle,right] = {el->setstart}
        cmp[dle,left] = {setstart->el}

        isSet = {setstart}
}

//a one-elementary set
%n = {setstart, u, el}
%p = {
        set = {setstart}
        element = {el}

        left = {setstart->u}

        down = {setstart->u}
        downStar = {u->u, el->el, setstart->setstart, setstart->u}
            downStar[left] = {setstart->u}
        r[set] = {setstart, u}
        r[element] = {el}

        inOrder[dle] = 1

        treeNess = 1

        dle = {setstart->setstart, u->setstart, el->setstart,
               u->u, el->el, el->u:1/2, u->el:1/2}

        dle[element, left] = {u:1/2}
        dle[element, right] = {setstart, u:1/2}

        cmp[dle,right] = {u->setstart, el->setstart, el->u:1/2, u->el:1/2}
        cmp[dle,left] = {setstart->u, setstart->el, el->u:1/2, u->el:1/2}
```

```
        isSet = {setstart}
        isElement = {u->setstart, el->setstart:1/2}
}

//a non-empty set
%n = {setstart, u, us, el}
%p = {
        sm = {us:1/2}
        set = {setstart}
        element = {el}

        left = {u->us:1/2, us->us:1/2, setstart->u}
        right = {u->us:1/2, us->us:1/2}

        down = {setstart->u, u->us:1/2,us->us:1/2}
        downStar = {u->u, u->us, us->us:1/2, el->el,
                    setstart->setstart, setstart->u, setstart->us}
            downStar[left] = {setstart->u, setstart->us, u->us:1/2, us->us:1/2}
        downStar[right] = {u->us:1/2, us->us:1/2}
        r[set] = {setstart, u, us}
        r[element] = {el}

        inOrder[dle] = 1

        treeNess = 1

        dle = {setstart->setstart, u->setstart, us->setstart, el->setstart,
               u->u, u->us:1/2, us->u:1/2, us->us:1/2,
               el->el, el->u:1/2, el->us:1/2, u->el:1/2, us->el:1/2}

        dle[element, left] = {u:1/2, us:1/2}
        dle[element, right] = {setstart, u:1/2, us:1/2}

        cmp[dle,right] = {u->setstart, us->setstart, el->setstart,
                          u->us:1/2, us->u:1/2, us->us:1/2,
                          el->u:1/2, el->us:1/2, u->el:1/2, us->el:1/2}
        cmp[dle,left] = {setstart->u, setstart->us, setstart->el,
                         u->us:1/2, us->u:1/2, us->us:1/2,
                         el->u:1/2, el->us:1/2, u->el:1/2, us->el:1/2}

        isSet = {setstart}
        isElement = {u->setstart, us->setstart, el->setstart:1/2}
}
```

Insertion

```
%s PVar {set, tree, previous, element, root}
%s TRVar {element}
%s TSel {left, right}

#include "pred_sort.tvp"
%%
#include "actions_sort.tvp"
%%

/*
void insertElement(Set* set, void* element)
{
  Tree* tree = set->tree;
  Tree* previous = tree;

  while (tree != 0)  //find suitable position for new element
    {
      previous = tree;
      if (compare(tree->data, element->data) < 0)
        tree = tree->left;
      else if (compare(tree->data, element->data) > 0)
        tree = tree->right;
      else if (compare(tree->data, element->data) == 0) //element is already contained...
        return;
    }

  set->size++;

  tree = (Tree*)malloc(sizeof(tree));
  tree->data = element;
  tree->left = 0;
  tree->right = 0;
  if (previous == 0) //first element to be inserted... (tree was empty)
    {
      set->tree = tree;
    }
  else
    {
      if (compare(previous->data, element->data) < 0)
        previous->left = tree;
      else if (compare(previous->data, element->data) > 0)
        previous->right = tree;
    }
}
```

```
*/



L0  Copy_Reach_T(set)             L1

L1  Get_Sel_T(tree, set, left)    L2    // Tree* tree = set->tree;
                                //left denotes tree for sets... a new selection predicate wou
L2  Copy_Var_T(previous, tree)    L3    // Tree* previous = tree;

L3  Is_Not_Null_Var(tree)         L4    // while (tree != 0)
L3  Is_Null_Var(tree)             L81   //
L4  Copy_Var_T(previous, tree)    L5    // previous = tree;
L5  Greater_Data_T(tree, element) L6    // if (compare(tree->data, element->data) < 0)
L5  Less_Data_T(tree, element)    L7    // if (compare(tree->data, element->data) > 0)
L5  Equal_Data_T(tree, element) exit    // if (compare(tree->data, element->data) == 0)

L6  Get_Sel_T(tree, tree, left)   L3    // tree = tree->left;
L7  Get_Sel_T(tree, tree, right)  L3    // tree = tree->right;

                                        // set->size++;


L81 Set_Null_T(tree)              L8
L8  Malloc_T(tree)                L9    // tree = (Tree*)malloc(sizeof(tree));
L9  Set_Data_T(tree, element)     L10   // tree->data = element->data
L10 Set_Sel_Null_T(tree, left)    L11   // tree->left = 0;
L11 Set_Sel_Null_T(tree, right)   L12   // tree->right = 0;

L12 Is_Null_Var(previous)         L13   // if (previous == 0)
L12 Is_Not_Null_Var(previous)     L15   // else
L13 Set_Sel_Null_T(set, left)     L14   // set->tree = 0;
L14 Set_Sel_T(set, left, tree)    exit1 // set->tree = tree;
L15 Greater_Data_T(previous, element) L16a //if (compare(previous->data, element->data)<0)
L15 Less_Data_T(previous, element) L17a //if (compare(previous->data, element->data)>0)
L16a Set_Sel_Null_T(previous, left) L16b    // previous->left = 0;
L16b Set_Sel_T(previous, left, tree) exit1  // previous->left = tree;
L17a Set_Sel_Null_T(previous, right) L17b   // previous->right = tree;
L17b Set_Sel_T(previous, right, tree) exit1 // previous->right = tree;


exit Set_Null_T(tree)                 exit1 // tree = 0
exit1 Set_Null_T(previous)            exit2 // previous = 0;


exit2 Is_Not_Element(element, set)       error
```

```
exit2 Is_Not_Sorted_Data_T()            error
exit2 Assert_Permutation_T(set, element) error

%% L1, exit2, error
```

Removal

```
%s PVar {root, set, tree, treeRight, treeLeft, following, previous, previous2,
        element, temp, subtree}
%s TRVar {element}
%s TSel {left, right}

#include "pred_sort.tvp"
%%
#include "actions_sort.tvp"
%%

/*
void removeElement(Set* set, void* element)
{
    Tree* treeRight = 0;
    Tree* treeLeft = 0;
    Tree* following = 0;
    Tree* tree = set->tree;
    Tree* previous = 0;
    Tree* previous2 = 0;
    Tree* temp = 0;
    Tree* subtree = 0;

    while (tree != 0) //find element...
      {
        if (compare(tree->data, element) == 0) //we found the element.
        {
          treeLeft = tree->left;
          treeRight = tree->right;
          tree->left = 0;
          tree->right = 0;
          set->size--;

          if ((treeRight == 0) && (treeLeft == 0))
            following = 0;
          else if (treeRight == 0)
            following = treeLeft;
          else if (treeLeft == 0)
            following = treeRight;
```

```
    if ((treeRight == 0) || (treeLeft == 0))
      {
        if (previous == 0)
          set->tree = following;
        else
          {
            temp = previous->left;
            if (temp == tree)
              previous->left = following;
            else
              previous->right = following;
            temp = 0;
          }
      }

    following = 0;

    if ((treeRight != 0) && (treeLeft != 0))
      {  //position has two subtrees: either find largest element to the left
          //or smallest to the right; i chose left here
        subtree = treeLeft;
        previous2 = 0;
        temp = subtree->right;
        while (temp != 0)   //finding largest element to the left of the element that
                                  //is being removed
          {
            previous2 = subtree;
            subtree = temp;
            temp = subtree->right;
          }
        temp = 0;
        if (previous2 != 0)         //remove element from predecessor
        {
          temp = subtree->left;
          subtree->left = 0;
          previous2->right = 0;
          previous2->right = temp;
                temp = 0;
          }

        subtree->right = 0;
        if (treeLeft != subtree)      //otherwise we would introduce a cycle
                subtree->left = treeLeft;   //attach former subtrees of removed element
```

```
            subtree->right = treeRight;


            if (previous == 0)          //link it to predecessor of removed element
              set->tree = subtree;
            else
              {
                temp = previous->left;
                if (temp == tree)
                  previous->left = subtree;
                else
                  previous->right = subtree;
                temp = 0;
              }
          }

        free(tree);

        return;
      }

      previous = tree;

      if (compare(tree->data, element) < 0) //traversing the tree in search of the element.
        tree = tree->left;
      else
        tree = tree->right;
    }
}

*/

/*

*/

//including data field...
L0       Copy_Reach_T(set)                    Lentry1

Lentry1  Set_Null_T(treeRight)                Lentry2       // Tree* treeRight = 0;
Lentry2  Set_Null_T(treeLeft)                 Lentry3       // Tree* treeLeft = 0;
Lentry3  Set_Null_T(following)                Lentry4       // Tree* following = 0;
Lentry4  Get_Sel_T(tree, set, left)           Lentry5       // Tree* tree = set->tree;
//left denotes tree for sets... a new selection predicate would make things way more complica
Lentry5  Set_Null_T(previous)                 Lentry6       // Tree* previous = 0;
```

```
Lentry6  Set_Null_T(previous2)              Lentry7      // Tree* previous2 = 0;
Lentry7  Set_Null_T(temp)                   Lentry8      // Tree* temp = 0;
Lentry8  Set_Null_T(subtree)                Lwhile       // Tree* subtree = 0;


Lwhile   Is_Not_Null_Var(tree)              Lbody        // while (tree != 0) find element.
Lwhile   Is_Null_Var(tree)                  exit         //
Lbody    Not_Equal_Data_T(tree, element) Lnotfound
        //else (if (compare(tree->data, element->data) != 0))
Lbody    Equal_Data_T(tree, element)        Lfound
        //if (compare(tree->data, element->data) == 0)

Lfound   Get_Sel_T(treeLeft, tree, left)    Lf1          // treeLeft = tree->left;
Lf1      Get_Sel_T(treeRight, tree, right)  Lf1b         // treeRight = tree->right
Lf1b     Set_Sel_Null_T(tree, left)         Lf1c         // tree->left = 0;
Lf1c     Set_Sel_Null_T(tree, right)        Lf2          // tree->right = 0;
                                                         // set->size--;
Lf2      Is_Null_Var(treeRight)             Lf3          //
Lf2      Is_Not_Null_Var(treeRight)         Lf5          //
Lf3      Is_Null_Var(treeLeft)              LfNull //if ((treeRight == 0) && (treeLeft == 0)
Lf3      Is_Not_Null_Var(treeLeft)          LfRight      // else if (treeRight == 0)
Lf5      Is_Null_Var(treeLeft)              LfLeft       // else if (treeLeft == 0)
Lf5      Is_Not_Null_Var(treeLeft)          LfAfterAll   // else...
LfNull   Set_Null_T(following)              LfAfter      // following = 0;
LfRight  Copy_Var_T(following, treeLeft)    LfAfter      // following = treeLeft;
LfLeft   Copy_Var_T(following, treeRight)   LfAfter      // following = treeRight;

LfAfter  Is_Null_Var(previous)              LfAfter2     // if (previous == 0)
//if ((treeRight == 0) || (treeLeft == 0)) is ensured...
LfAfter  Is_Not_Null_Var(previous)          LfAfter3     // else
LfAfter2 Set_Sel_Null_T(set, left)          LfAfter2b    // set->tree = 0;
LfAfter2b Set_Sel_T(set, left, following)   LfAfterAll   // set->tree = following
LfAfter3 Get_Sel_T(temp, previous, left)    LfAfter4     // temp = previous->left;
LfAfter4 Is_Eq_Var(temp, tree)              LfAfter5     // if (temp == tree)
LfAfter4 Is_Not_Eq_Var(temp, tree)          LfAfter7     // else
LfAfter5 Set_Sel_Null_T(previous, left)     LfAfter6     // previous->left = 0;
LfAfter6 Set_Sel_T(previous, left, following) LfAfterAll //previous->left = following;
LfAfter7 Set_Sel_Null_T(previous, right)    LfAfter8     // previous->right = 0;
LfAfter8 Set_Sel_T(previous, right, following) LfAfterAll //previous->right = following;
LfAfterAll Set_Null_T(temp)                 LfAfterAlla  // temp = 0;

LfAfterAlla Set_Null_T(following)           LfR1         // following = 0;
LfR1     Is_Not_Null_Var(treeRight)         LfR2         // if (treeRight != 0)
LfR1     Is_Null_Var(treeRight)             exit         // else
LfR2     Is_Not_Null_Var(treeLeft)          LfRin        // if (treeLeft != 0)
```

```
LfR2      Is_Null_Var(treeLeft)               exit          // else

LfRin     Copy_Var_T(subtree, treeLeft)       LfRin2        // subtree = treeLeft;
LfRin2    Set_Null_T(previous2)               LfRin3        // previous2 = 0;
LfRin3    Get_Sel_T(temp, subtree, right)     Lwhile2       // temp = subtree->right;

Lwhile2   Is_Not_Null_Var(temp)               Lw2body       // while (temp != 0)
Lwhile2   Is_Null_Var(temp)                   Lfbodyend     // else
Lw2body   Copy_Var_T(previous2, subtree)      Lw22          // previous2 = subtree;
Lw22      Copy_Var_T(subtree, temp)           Lw23          // subtree = temp;
Lw23      Get_Sel_T(temp, subtree, right)     Lwhile2       // temp = subtree->right;
Lfbodyend Set_Null_T(temp)                    Lfb2          // temp = 0;


Lfb2      Is_Not_Null_Var(previous2)          Lfb3          // if (previous2 != 0)
Lfb2      Is_Null_Var(previous2)              Lfb5          // else
Lfb3      Get_Sel_T(temp, subtree, left)      Lfb3aa        // temp = subtree->left;
Lfb3aa    Set_Sel_Null_T(subtree, left)       Lfb3a         // subtree->left = 0;
Lfb3a     Set_Sel_Null_T(previous2, right)    Lfb3b         // previous2->right = 0;
Lfb3b     Set_Sel_T(previous2, right, temp)   Lfb3c         // previous2->right = temp;
Lfb3c     Set_Null_T(temp)                    Lfb5          // temp = 0;


Lfb5      Set_Sel_Null_T(subtree, right)      Lfb6          // subtree->right = 0;
Lfb6      Is_Not_Eq_Var(treeLeft, subtree)    Lfb7          // if (treeLeft != subtree)
Lfb6      Is_Eq_Var(treeLeft, subtree)        Lfb8          // else
Lfb7      Set_Sel_T(subtree, left, treeLeft)  Lfb8          // subtree->left = treeLeft;
Lfb8      Set_Sel_T(subtree, right, treeRight) Lfb9         // subtree->right = treeRight;



Lfb9      Is_Null_Var(previous)               Lfb10         // if (previous == 0)
Lfb9      Is_Not_Null_Var(previous)           Lfb12         // else
Lfb10     Set_Sel_Null_T(set, left)           Lfb11         // set->tree = 0;
Lfb11     Set_Sel_T(set, left, subtree)       exit          // set->tree = subtree;
Lfb12     Get_Sel_T(temp, previous, left)     Lfb13         // temp = previous->left;
Lfb13     Is_Eq_Var(temp, tree)               Lfb14         // if (previous == 0)
Lfb13     Is_Not_Eq_Var(temp, tree)           Lfb15         // else
Lfb14     Set_Sel_Null_T(previous, left)      Lfb14a        // previous->left = 0;
Lfb14a    Set_Sel_T(previous, left, subtree)  Lfbt          // previous->left = subtree;
Lfb15     Set_Sel_Null_T(previous, right)     Lfb15a        // previous->right = 0;
Lfb15a    Set_Sel_T(previous, right, subtree) Lfbt          // previous->right = subtree;
Lfbt      Set_Null_T(temp)                    exit          // temp = 0;
```

```
Lnotfound Copy_Var_T(previous, tree)        Lnf2         // previous = tree;
Lnf2      Less_Data_T(tree, element)        Lnf4  // if (compare(tree->data, element) < 0)
Lnf2      Greater_Data_T(tree, element)     Lnf3         // else
Lnf3      Get_Sel_T(tree, tree, left)       Lwhile       // tree = tree->left;
Lnf4      Get_Sel_T(tree, tree, right)      Lwhile       // tree = tree->right;


exit      Set_Null_T(treeLeft)              exit0        // treeLeft = 0;
exit0     Set_Null_T(treeRight)             exitr        // treeRight = 0;

exitr     Set_Null_T(tree)                  exit1        // tree = 0
exit1     Set_Null_T(previous)              exit2        // previous = 0;
exit2     Set_Null_T(previous2)             exit3        // previous2 = 0;
exit3     Set_Null_T(subtree)               exit4        // subtree = 0;
exit4     Set_Null_T(temp)                  exit5        // temp = 0;
exit5     Set_Null_T(following)             exit6        // following = 0;

exit6     Is_Element(element, set)          error
exit6     Is_Not_Sorted_Data_T()            error
exit6     Assert_Permutation_T(set, element)  error

%% Lentry1, exit6, error
```

Membership Test

```
%s PVar {set, tree, element}
%s TRVar {element}
%s TSel {left, right}

#include "pred_sort.tvp"
%%
#include "actions_sort.tvp"
%%

/*
int isElement(Set* set, void* element) //returns 0 if element is not contained,
                                                    //otherwise depth starting at 1
{
    Tree* tree = set->tree;
    int depth = 0;

    while (tree != 0)
      {
        depth++;
               printf("%i\n",*(int*)tree->data);
```

```
        int compresult = compare(tree->data, element);
        if (compresult == 0)
          return 1; //depth;
        else if (compresult < 0)
          tree = tree->left;
        else
          tree = tree->right;
      }

    return 0;
}
*/

L0   Copy_Reach_T(set)                    L1
L1   Get_Sel_T(tree, set, left)           L2     // Tree* tree = set->tree;
//left denotes tree for sets... a new selection predicate would make things complicated

L2   Is_Not_Null_Var(tree)                L3     // while (tree != 0)
L2   Is_Null_Var(tree)                    exitnotfound
L3   Equal_Data_T(tree, element)          exitfound //if (compresult = 0) return 1;
L3   Greater_Data_T(tree, element)        L4     // else if (compresult < 0)
L3   Less_Data_T(tree, element)           L5     // else if (compresult > 0)
L4   Get_Sel_T(tree, tree, left)          L2     // tree = tree->left;
L5   Get_Sel_T(tree, tree, right)         L2     // tree = tree->right;

exitfound    Is_Not_Element(element, set) exitfounderror
exitnotfound Is_Element(element, set)     exitnotfounderror

%% L1, exitfound, exitnotfound, exitfounderror, exitnotfounderror
```