Saarland University
Faculty of Natural Sciences and Technology I
Department of Computer Science
Master's Program in Computer Science

Master's Thesis

Towards Shape Analysis of B-Trees

submitted by

Jörg Herter

on January 10, 2008

Supervisor
Professor Dr. R. Wilhelm

Advisor
Jan Reineke, M.Sc.

Reviewers
Professor Dr. R. Wilhelm
Professor B. Finkbeiner, Ph. D.

Statement

Hereby I confirm that this thesis is my own work and that I have documented all sources used.

Saarbrücken, January 10, 2008

Jörg Herter

Declaration of Consent

Herewith I agree that my thesis will be made available through the library of the Computer Science Department.

Saarbrücken, January 10, 2008

Jörg Herter

Acknowledgements

I am indebted to my advisor Jan Reineke for his guidance and support during the whole course of this thesis. Our weekly discussions were a great source of helpful ideas, new insights, and perspectives and always provided much motivation. Parts of my work on analyzing 2-3-4 tree implementations are based on Jan's work on binary trees and I always felt fortunate to be able to discuss his work with him face-to-face.

I am also very grateful to Professor Reinhard Wilhelm for offering me the opportunity to work on this thesis, his continuous support, and the kind integration into his group. I especially want to thank Professor Finkbeiner for agreeing on reviewing this thesis on such short notice.

For numerous enlightening discussions I would like to thank Basileios Anastasatos, Björn Wachter, Tal Lev-Ami, and Roman Manevich.

Abstract

Shape analysis is concerned with statically analyzing programs that perform destructive updating on dynamically allocated heap storage. The analysis aims at proving structural properties of the allocated heap space. In recent years, a parametric shape analysis framework proposed by Sagiv, Reps, and Wilhelm has been instantiated to create shape analyses which were able to prove partial correctness of many interesting heap-manipulating programs.

We[1] give an overview on this framework and develop instantiations of it that are capable of proving partial correctness of operations on 2-3-4 tree structures. 2-3-4 trees are the simplest form of B-trees. B-trees themselves are a ubiquitous data structure used, for example, in file systems, database systems, and - more recently - peer to peer systems. Their widespread use makes implementations of this data structure a prime objective for an automatical verification. However, their complex structure (compared to other trees) has prevented such a verification so far. We give an extensive introduction to B-trees and their variations as well as an overview on their range of applications. Based on our shape analyses on 2-3-4 tree operations, we identify the remaining challenges in doing a shape analysis of (general) B-trees. We present promising approaches to overcome those challenges.

Our analyses and theoretical considerations are based on object-oriented Java implementations of B- and 2-3-4 trees.

[1] The use of the plural pronoun is customary even in solely authored research papers and theses. Therefore, the author of this thesis is also referred to by *we*.

Contents

1 Introduction

Shape analysis is concerned with statically analyzing programs that perform destructive updating on dynamically allocated (heap-) storage in order to determine heap invariants describing the heap-allocated data structures arising during program execution. The results of such an analysis can be used to understand and to verify the analyzed program. Additionally, they also yield valuable information for debugging, compile-time garbage collection, instruction scheduling, and parallelization [WSR00].
Shape analysis is a generic term denoting different techniques attempting to infer properties of the heap and the linked structures built within. Each technique relies on a sophisticated heap abstraction that abstracts the in general infinite set of possible heap shapes to a finite one but still preserves *enough* information to show that invariants do hold [HR05]. Heap abstractions characterizing the *entire* heap include reachability matrices [HHN94, GH96, GH99], shape graphs [KLR02, SRW98], and, most recently, three-valued logical structures [SRW02, RSW04]. Contrary to those, abstractions characterizing single heap cells are investigated by Rugina et al. [HR05, CR06a, CR06b, Rug04].
Another popular technique utilizes separation logic [Rey02] to abstract structure segments, i.e. segments of the structures arising in the heap during program execution [DOY06, GBC06].
Another interesting approach utilizes extended tree automata to abstract program configurations [BHRV06b, BHRV06a]. Amir Pnueli proposed a *shape analysis by predicate abstractions* [BPZ05]. Wies and Podelski combined key ideas of three-valued logic and predicate abstraction into a new symbolic approach to shape analysis using so called boolean heaps [Wie04, PW05]. The usage of decidable monadic second order logic has led to the shape analysis tool PALE (Pointer Assertion Logic Engine) in which the program to analyze is encoded in logic with explicit loop and function call invariants [JJKS97, EMS00, MS01].

In this thesis we rely on a parametric framework for shape analysis via three-valued logic, presented in detail in [SRW02], which can be instantiated in different ways to create shape analysis algorithms that provide different degrees of precision. The theoretical background behind this framework as well as further developments and tools implementing the framework are summarized and reviewed by M. Jensen in his Master's Thesis [Jen05].
The before mentioned framework for shape analysis was firstly implemented in the tool TVLA (Three-Valued-Logic Analyzer) [LA00, LAS00, LAMS04]. While TVLA and PALE have similar goals, their underlying techniques are radically different. However, they seem to be similarly precise and efficient [MS01]. Still, due to the fact that PALE cannot determine loop invariants by itself and needs those given explicitly, we consider the analysis framework implemented in TVLA to be more powerful.

In the last couple of years, shape analysis by three-valued logic - or more precisely, TVLA - was used to validate, among others, bubble- and insertion-sort algorithms operating on linked lists [LARSW00], implementations of the abstract data type set (internally using a singly-linked list or an unbalanced binary tree) [Rei05] as well as to prove that an implementation of the insertion operation for AVL trees preserves the tree's balancing [Par05]. A generalization of the shape analysis framework was implemented as an extension to TVLA - called 3VMC[1] - and used to verify safety properties of concurrent Java programs [Yah01].

We will use TVLA in its version 3.0-alpha. Besides of version 3.0 being the latest version of TVLA, two features made this version particularly appealing. Firstly, it supports interprocedural shape analysis for cutpoint free programs. This allows for shape analyses on realistic B-tree implementations encapsulating single operations on the tree into multiple methods. Secondly, compared to the version used in the related work given above, revamping of the software resulted in a significant runtime improvement of up to a factor of 50. This speedup was obtained by employing techniques from the database community - such as query optimization and semi-naïve evaluation - in order to reduce the cost of extracting information from shape descriptors and performing abstract interpretation of program statements and conditions [BLARS07].

1.1 Overview

This thesis is organized and structured as follows. In Chapter 2, we introduce general terms and definitions that are used in several chapters throughout the thesis. Chapter 3 gives a self-contained overview on the shape analysis framework proposed by Sagiv et al. we later base our analyses on. A brief overview on the range of applications for B-trees and their history as well as formal definitions for this data structure and its most important variations are given in Chapter 4. In Chapter 5, we present shape analyses for various operations on 2-3-4 trees, identify remaining problems with doing similar analyses on general B-trees, and present solutions to these problems. We conclude in Chapter 6.

Most of our proofs can be found in Appendix A while the source code of the tree implementations, the input files for TVLA and J2TVLA, and other files needed to reproduce our results can be found in Appendix B.

[1] 3VMC is integrated into TVLA since version 0.91.

2 Terms and Definitions

In order to avoid unnecessary confusion and/or misunderstandings we start with defining some terms and notions that we will use in the subsequent chapters.

2.1 Directed Trees

Definition 1 *A directed graph is a tupel (V, E), where V is a set of nodes, $E \subseteq V \times V$ the set of directed edges.*

The edge $e = (v_1, v_2) \in E$ is said to initiate from v_1 and terminate at v_2.

In this thesis, we define a *directed tree* as follows.

Definition 2 *A directed tree is a directed graph $T = (V, E)$, where V is its finite node set, E its edge set, with the following additional properties:*

1. *T is the empty tree or there exists exactly one node at which no edge terminates. Or more formally:*
   $V = \emptyset \vee (\exists! r \in V : \nexists e \in E : e \textit{ terminates in } r)$
   We call r the root of T.

2. *When ignoring all directions of edges any two nodes of T are connected by a unique path.*

We call $L = \{n \in V | \ \nexists e \in E : e \text{ initiates from } n\}$ the set of leaves of T. A single node $n \in L$ is called a leaf.

This is pretty much a standard definition, however, we wanted to clarify that we consider a directed tree to have only a finite number of nodes (and hence a finite number of edges) and that there exists a unique root node if the tree is not empty.

2.2 Natural Numbers

To avoid any confusion whether we want the set of natural numbers, denoted by $\mathbb{N}$, to contain 0 or not, we define 1 to be the smallest element of $\mathbb{N}$. We further refer to the set of natural numbers including 0 as $\mathbb{N}_0$. Hence, $\mathbb{N}_0 = \mathbb{N} \cup \{0\}$.
We use $\mathbb{N}^{>k}$ as an abbreviation for the set $\{n \in \mathbb{N} | n > k\}$.

3 Shape Analysis Foundations

Shape analysis concerns the problem of determining heap - or shape - invariants for programs that perform destructive updating on dynamically allocated memory. We will develop a shape analysis fitting into a parametric framework for shape analysis proposed in [SRW02]. A key innovation of this framework is to represent the heap allocated store that can arise during program execution using 3-valued logical structures.
We will give an introduction to this framework by summarizing the main techniques, observations, and theorems. We will also define key concepts and vocabulary that we will later use in our analysis. For a more detailed treatment of the framework, we refer to [SRW02]. Other, similar overviews and summaries of this framework can also be found in [Jen05] and [Rei05].

3.1 Parametric Shape Analysis via 3-Valued Logic - A Short Overview

The overall approach to shape analysis the framework suggests can be summarized as follows.

1. Concrete stores can be represented using 2-valued logical structures. Therein, the interpretation of unary predicates encodes the contents of program variables. Interpretations of binary predicates encode contents of pointer-valued structure fields (or reference attributes of objects).

2. 2-valued first order logic with transitive closure can be used to describe properties of stores like reachability, (a-)cyclicity, and so on.

3. Kleene's 3-valued logic can be used to relate concrete stores to abstract stores. Kleene logic's third truth value is interpreted as *unknown* which makes it very useful in the presence of summary nodes about which we only have partial information.

4. In the 3-valued world we can utilize summary nodes to keep a finite number of logical structures.

5. The Embedding Theorem ensures that information obtained from the 3-valued world is compatible to that of the 2-valued world.

With those key ideas in mind, we will now introduce the utilized concepts in more detail.

3.2 Representing Stores

Stores are represented by *logical structures*. Concrete stores are encoded by 2-valued logical structures, abstract stores by 3-valued logical structures. Formally, a logical structure S over a *vocabulary* of predicate symbols P is a tuple $\langle U^S, \iota^S \rangle$. Where U^S is a universe of *individuals* and ι^S a function mapping each predicate symbol p of arity k and possible k-tuple of individuals from U^S to a truth value. In a 2-valued logical structure ι^S maps to $\{0,1\}$ (0 representing *false* and 1 representing *true*) while in a 3-valued logical structure the mapping is to $\{0,1,1/2\}$ where 0 stands for *false*, 1 for *true*, and 1/2 for *unknown*.
We encode concrete stores in 2-valued logical structures as follows: individuals represent heap cells, pointer variables to the heap are represented by unary predicates which are true for the heap cell the respective variable points to. Reference fields of objects (or structures) are represented by binary predicates.

2-valued logical structures can be intuitively graphically represented as depicted in Figure 3.1 which lists structures representing linked lists of length ≤ 4.

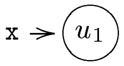
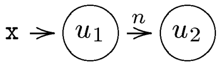
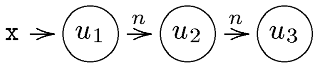
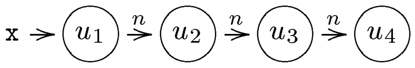
<table>
<tr><th>Name</th><th>Logical Structure</th><th>Graphical Representation</th></tr>
<tr><td>$S_0^\natural$</td><td>unary preds.<table><tr><th>indiv.</th><th>x</th><th>y</th><th>t</th><th>e</th></tr></table>binary preds.<table><tr><th>n</th></tr></table></td><td></td></tr>
<tr><td>$S_1^\natural$</td><td>unary preds.<table><tr><th>indiv.</th><th>x</th><th>y</th><th>t</th><th>e</th></tr><tr><td>u_1</td><td>1</td><td>0</td><td>0</td><td>0</td></tr></table>binary preds.<table><tr><th>n</th><th>u_1</th></tr><tr><td>u_1</td><td>0</td></tr></table></td><td> </td></tr>
<tr><td>$S_2^\natural$</td><td>unary preds.<table><tr><th>indiv.</th><th>x</th><th>y</th><th>t</th><th>e</th></tr><tr><td>u_1</td><td>1</td><td>0</td><td>0</td><td>0</td></tr><tr><td>u_2</td><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>binary preds.<table><tr><th>n</th><th>u_1</th><th>u_2</th></tr><tr><td>u_1</td><td>0</td><td>1</td></tr><tr><td>u_2</td><td>0</td><td>0</td></tr></table></td><td> </td></tr>
<tr><td>$S_3^\natural$</td><td>unary preds.<table><tr><th>indiv.</th><th>x</th><th>y</th><th>t</th><th>e</th></tr><tr><td>u_1</td><td>1</td><td>0</td><td>0</td><td>0</td></tr><tr><td>u_2</td><td>0</td><td>0</td><td>0</td><td>0</td></tr><tr><td>u_3</td><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>binary preds.<table><tr><th>n</th><th>u_1</th><th>u_2</th><th>u_3</th></tr><tr><td>u_1</td><td>0</td><td>1</td><td>0</td></tr><tr><td>u_2</td><td>0</td><td>0</td><td>1</td></tr><tr><td>u_3</td><td>0</td><td>0</td><td>0</td></tr></table></td><td> </td></tr>
<tr><td>$S_4^\natural$</td><td>unary preds.<table><tr><th>indiv.</th><th>x</th><th>y</th><th>t</th><th>e</th></tr><tr><td>u_1</td><td>1</td><td>0</td><td>0</td><td>0</td></tr><tr><td>u_2</td><td>0</td><td>0</td><td>0</td><td>0</td></tr><tr><td>u_3</td><td>0</td><td>0</td><td>0</td><td>0</td></tr><tr><td>u_4</td><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>binary preds.<table><tr><th>n</th><th>u_1</th><th>u_2</th><th>u_3</th><th>u_4</th></tr><tr><td>u_1</td><td>0</td><td>1</td><td>0</td><td>0</td></tr><tr><td>u_2</td><td>0</td><td>0</td><td>1</td><td>0</td></tr><tr><td>u_3</td><td>0</td><td>0</td><td>0</td><td>1</td></tr><tr><td>u_4</td><td>0</td><td>0</td><td>0</td><td>0</td></tr></table></td><td> </td></tr>
</table>

Figure 3.1: 2-valued logical structures representing singly-linked lists of length ≤ 4, taken from [SRW02]

By encoding stores as logical structures, questions about properties of stores can be answered by evaluating formulæ expressing those properties. The property holds or does not hold, depending on whether the respective formula evaluates to 1 or 0, respectively, in the logical structure. This observation is known as the *Property-Extraction Principle*.

3.3 Semantics of Program Statements

Similar to the extraction of store properties, also the expression of the semantics of the program statements is based on evaluating formulæ. The concrete semantics of a program statement *st* are captured by a set of *predicate-update formulæ*. Let σ be the store before statement *st* and σ' the store that arises after *st* is evaluated on σ. Let further σ be encoded in S. A collection of predicate-update formulæ consisting of one formula for each predicate in the vocabulary of S allows one to obtain the logical structure S' that encodes σ'. When evaluated in S, the predicate-update formula for a predicate p determines the value of p in S'.

For an abstract interpretation of program statements we need an abstract semantics that must be able to represent every possible runtime situation and does not yield too many *unknown* values. However, combining our observation that the semantics of program statements can be expressed by logical formulæ together with the fact that evaluating a formula in a 3-valued structure S is safe with respect to an evaluation in any 2-valued structure represented by S, yields an abstract semantics. We can just evaluate the predicate-update formulæ of the concrete semantics on 3-valued structures.
Sagiv et al. formulated this within the so called *Reinterpretation Principle*:
Evaluation of predicate-update formulæ for a statement st in 2-valued logic captures the transfer function for st of the concrete semantics. Evaluation of the same formulæ in 3-valued logic captures the transfer function for st of the abstract semantics.

3.4 Expressing Concrete Semantics

We start by defining the syntax of formulæ for a first-order logic with transitive closure. Let $\mathcal{P} = \{p_1, \ldots, p_n\}$ be a finite set of predicate symbols.

Definition 3 (Formula) *A* formula *over the vocabulary $\mathcal{P}$ is defined inductively by:*

- *The logical literals* 0 *and* 1 *are atomic formulæ with no free variables.*
- *For every $p \in \mathcal{P}$ of arity k, $p(v_1, \ldots, v_k)$ is an atomic formulæ with the set of free variables $\{v_1, \ldots, v_k\}$.*
- *$v_1 = v_2$ is an atomic formula with free variables v_1 and v_2.*

- *Let φ_1 and φ_2 be formulæ whose sets of free variables are V_1 and V_2, respectively, then $\varphi_1 \wedge \varphi_2$, $\varphi_1 \vee \varphi_2$ and $\neg\varphi_1$ are formulæ with sets of free variables $V_1 \cup V_2$, $V_1 \cup V_2$, and V_1, respectively.*

- *Let φ be a formula whose set of free variables is $\{v_1, v_2, \ldots, v_k\}$. $\exists v_1.\varphi$ and $\forall v_1.\varphi$ are both formulæ whose respective set of free variables is $\{v_2, \ldots, v_k\}$.*

- *(Transitive Closure) If φ is a formula with the set of free variables V such that $v_3, v_4 \notin V$ then $(TC\ v_1, v_2.\varphi)(v_3, v_4)$ is a formula with free variables $(V \backslash \{v_1, v_2\}) \cup \{v_3, v_4\}$.*

We also use several shorthand notations. For example, $\varphi_1 \Rightarrow \varphi_2$ (*implication*, shorthand for $\neg\varphi_1 \vee \varphi_2$) and $\varphi_1 \Leftrightarrow \varphi_2$ (*equivalence*, shorthand for $(\varphi_1 \wedge \varphi_2) \vee (\neg\varphi_1 \wedge \neg\varphi_2)$).

Definition 4 (2-Valued Interpretation) *A* 2-valued interpretation *of the language of formulæ over $\mathcal{P}$ is a 2-valued logical structure $S = \langle U^S, \iota^S \rangle$ as defined earlier so that ι^S maps each predicate symbol p of arity k to a truth-valued function:*

$$\iota^S(p) : \left(U^S\right)^k \mapsto \{0, 1\}$$

Definition 5 (Assignment) *An assignment is a function that maps free variables to individuals.*

$$\{v_1, v_2, \ldots\} \mapsto U^S$$

For the remainder of this thesis, we assume all assignments arising in the discussion of some formula φ to be defined on all free variables of φ. Such assignments are also called complete *for φ.*

Definition 6 (Meaning) *The 2-valued meaning of a formula φ under an assignment α, denoted by $[\![\varphi]\!]_2^S(\alpha)$, yields a truth value in $\{0, 1\}$. Again, we define the meaning of a formula inductively:*

- *For an atomic formula consisting of a logical literal $l \in \{0, 1\}$, we define the meaning of l as*
$$[\![l]\!]_2^S(\alpha) = l$$

- *For an atomic formula consisting of a predicate $p(v_1, \ldots, v_k)$, we define*
$$[\![p(v_1, \ldots, v_k)]\!]_2^S(\alpha) = \iota^S(p)\left(\alpha\left(v_1\right), \ldots, \alpha\left(v_k\right)\right)$$

- *For an atomic formula $v_1 = v_2$, we have*
$$[\![v_1 = v_2]\!]_2^S(\alpha) = \begin{cases} 1 & \alpha(v_1) = \alpha(v_2) \\ 0 & \alpha(v_1) \neq \alpha(v_2) \end{cases}$$

- *For a formula built from subformulæ φ_1, φ_2, and the logical connectives $\wedge$, $\vee$, and $\neg$, we define*

$$\begin{aligned}
[\![\varphi_1 \wedge \varphi_2]\!]_2^S(\alpha) &= \mathit{min}\left\{[\![\varphi_1]\!]_2^S(\alpha), [\![\varphi_2]\!]_2^S(\alpha)\right\}\\
[\![\varphi_1 \vee \varphi_2]\!]_2^S(\alpha) &= \mathit{max}\left\{[\![\varphi_1]\!]_2^S(\alpha), [\![\varphi_2]\!]_2^S(\alpha)\right\}\\
[\![\neg\varphi_1]\!]_2^S(\alpha) &= 1 - [\![\varphi_1]\!]_2^S(\alpha)
\end{aligned}$$

- *For a formula having a quantifier as its outermost operator, we define*

$$\begin{aligned}
[\![\forall v.\varphi]\!]_2^S(\alpha) &= \mathit{min}_{u \in U^S} [\![\varphi]\!]_2^S(\alpha[v \mapsto u])\\
[\![\exists v.\varphi]\!]_2^S(\alpha) &= \mathit{max}_{u \in U^S} [\![\varphi]\!]_2^S(\alpha[v \mapsto u])
\end{aligned}$$

- *If φ is a formula of the form $(TC\ v_1, v_2.\varphi)(v_3, v_4)$ then*

$$[\![(TC\ v_1, v_2.\varphi)(v_3, v_4)]\!]_2^S(\alpha) =$$

$$\mathit{max}_{\substack{n \geq 1, u_1, \ldots, u_{n+1} \in U^S,\\ \alpha(v_3) = u_1, \alpha(v_4) = u_{n+1}}} \ \mathit{min}_{i \in \{1, \ldots, n\}} [\![\varphi]\!]_2^S(\alpha[v_1 \mapsto u_i, v_2 \mapsto u_{i+1}])$$

A logical structure S and an assignment α are said to *satisfy* φ if $[\![\varphi]\!]_2^S(\alpha) = 1$. We denote that a structure S and an assignment α satisfy a formula φ by $S, \alpha \models \varphi$ and write $S \models \varphi$ if $S, \alpha \models \varphi\ \forall\alpha$.
By 2-STRUCT[$\mathcal{P}$] we denote the set of 2-valued structures.

3.5 Meaning of Program Statements

The new values of every predicate p after evaluation of a statement st are defined via predicate-update formulæ φ_p^{st}.

Definition 7 ($\mathcal{P}$ Transformer) *Let st be a program statement, and for every k-ary predicate p in vocabulary $\mathcal{P}$, let φ_p^{st} be the predicate-update formula over free variables $v_1, \ldots, v_k$ defining the new value of p after evaluation of st. Then the $\mathcal{P}$* transformer associated with *st, denoted by $[\![st]\!]$: 2-STRUCT[$\mathcal{P}$] $\to$ 2-STRUCT[$\mathcal{P}$], is defined as follows.*

$$[\![st]\!](S) = \langle U^S, \lambda p.\lambda u_1, \ldots, u_k.[\![\varphi_p^{st}]\!]_2^S([v_1 \mapsto u_1, \ldots, v_k \mapsto u_k])\rangle.$$

3.6 Collecting Semantics

We will now define a shape analysis on concrete structures using the previous concepts and notations. Assume G to be the control-flow graph of the program we wish to analyze. For each vertex v of G we want to compute ConcStructSet[v], the potentially infinite set of *structures that may arise on the entry to v for some potential input structure. We* compute *$\mathit{ConcStructSet}[v]$ as the least fixed point in terms of set inclusion of the following system of equations:*

$$
\begin{aligned}
&\mathit{ConcStructSet}[v] = \\
&\begin{cases}
\{\langle\emptyset,\emptyset\rangle\} & \mathit{if}\ v = \mathit{start}\\
\left.\begin{array}{lll}
 & \displaystyle\bigcup_{w\to v\in E(G),\ w\in As(G)} \{[\![st(w)]\!](S) \mid S \in \mathit{ConcStructSet}[w]\} & (1)\\
\cup & \displaystyle\bigcup_{w\to v\in E(G),\ w\in Id(G)} \{S \mid S \in \mathit{ConcStructSet}[w]\} & (2)\\
\cup & \displaystyle\bigcup_{w\to v\in Tb(G)} \{S \mid S \in \mathit{ConcStructSet}[w]\ \mathit{and}\ S \models \mathit{cond}(w)\} & (3)\\
\cup & \displaystyle\bigcup_{w\to v\in Fb(G)} \{S \mid S \in \mathit{ConcStructSet}[w]\ \mathit{and}\ S \models \neg\mathit{cond}(w)\} & (4)
\end{array}\right\} & \mathit{otherwise.}
\end{cases}
\end{aligned}
$$

In this system of equations, $As(G)$ denotes the set of assignment statements. Hence, (1) captures the effects of assignments. $Id(G)$ denotes the set of uninterpreted vertices, (2) captures their effects. The edge sets $Tb(G)$ and $Fb(G)$ represent the true and false branches, respectively, from branch points. $cond(w)$ denotes the formula for the program condition at w. Hence, (3) and (4) handle conditional branches by transferring only such structures arising at w that satisfy the condition associated with the respective edge.

In general, a least fixed point is not always computable due to the possibly infinite number of structures arising at a program point. However, the abstract semantics based on this concrete semantics will overcome this problem.

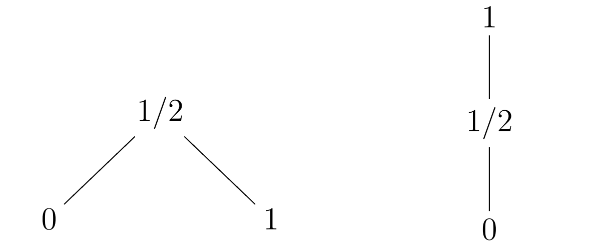
3.7 Representing Sets of Stores Using 3-Valued Logic

We advance by showing how 3-valued logical structures can be utilized to represent sets of concrete stores. We then relate concrete to abstract structures and finally develop an abstract semantics.

3.7.1 Kleene's 3-Valued Logic

Kleene's 3-valued logic adds to the 2-valued logic with its two definite truth values true and false a third, indefinite value. We denote this third value which can be read as unknown by 1/2. We distinguish the *information order* and the *logical order* of those truth values. The information order captures (un-)certainty or - in other words - indicates which value contains more definite information. The logical order indicates potential truth. Figure 3.2 visualizes these two orders on truth values. Figure 3.3 defines the logical connectives for 3-valued logic.

We define 3-valued logical structures and 3-valued interpretation analogous to their 2-valued counterparts.



(a) Information Order (b) Logical Order

Figure 3.2: 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

l	$\neg l$
0	1
1/2	1/2
1	0

Figure 3.3: Definition of the logical connectives in 3-valued logic.

Definition 8 (3-Valued Interpretation) *A* 3-valued interpretation *of the language of formulæ over $\mathcal{P}$ is a* 3-valued logical structure $S = \langle U^S, \iota^S \rangle$*, where U^S is a set of individuals and ι^S a mapping from predicate symbols to truth-valued functions:*

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

The 3-valued meaning of a formula φ, analogous to the 2-valued world denoted by $[\![\varphi]\!]_3^S(\alpha)$, yields now a truth value in $\{0, 1, 1/2\}$. The inductive definition of the meaning of φ is the same as in Definition 6 with the following modification for the case of φ being of the form $v_1 = v_2$:

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

Where the predicate *sm* formalizes the notion of a summary node, i.e. an individual of the 3-valued world which may represent more than 1 individual from corresponding 2-valued structures.

As with 2-valued structures, 3-valued logical structures can also be graphically represented: predicates with truth value 1/2 are represented by dotted lines, summary nodes are drawn doubly circled.
Furthermore, we say that S and α *potentially satisfy* φ (denoted by $S, \alpha \models_3 \varphi$) if $[\![\varphi]\!]_3^S(\alpha) = 1/2 \vee [\![\varphi]\!]_3^S(\alpha) = 1$. Again, we write $S \models_3 \varphi$ if for every assignment α the formula $S, \alpha \models_3 \varphi$ holds. The set of 3-valued structures is denoted by 3-STRUCT$[\mathcal{P} \cup \{sm\}]$.

3.8 Embedding into 3-Valued Structures

The concept of embedding provides a way to relate 2-valued and 3-valued interpretations. We assume every 2-valued structure has a predicate *sm* which is always false for all individuals. Under this assumption, we can define an embedding as follows.

Definition 9 (Embedding Order, Embedding) *Let $S = \langle U^S, \iota^S \rangle$ and $S' = \langle U^{S'}, \iota^{S'} \rangle$ be two logical structures and $f : U^S \mapsto U^{S'}$ a surjective function. f* embeds *S in S', written as $S \sqsubseteq^f S'$, if (1) for every predicate symbol p 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 $u' \in U^{S'}$

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

S can be embedded in *S' (denoted by $S \sqsubseteq S'$) if there exists a function f such that $S \sqsubseteq^f S'$.*

The concept of *tight embeddings* allows for a minimization of information loss when multiple individuals of S are mapped to the same individual in S'. For a definition a tight embeddings we refer the reader to [SRW02].

Embeddings additionally provide a way to define the potentially infinite set of concrete structures represented by a single 3-valued structure.

Definition 10 (Concretization of 3-Valued Structures) *Let S be a 3-valued structure. The set $\gamma(S)$ of 2-valued structures represented by S is defined as*

$$\gamma(S) = \left\{ S^\natural \in \textit{2-STRUCT}[\mathcal{P}] \mid S^\natural \sqsubseteq S \right\}$$

The *Embedding Theorem* states that if $S \sqsubseteq^f S'$ then every information extracted from S' via a formula φ is a conservative approximation of the information extracted from S via φ. For a proof of this theorem as well as a more formal formulation, we again refer the reader to [SRW02].
The concepts introduced so far combined with the Embedding Theorem provide us with:

- a systematic way to use an abstract 3-valued structure to answer questions about properties of concrete 2-valued structures,
- the assurance that evaluating a formula on a single 3-valued structure instead on all 2-valued structures represented by the 3-valued structure (again, which may be infinitely many) is safe and, in particular, definite values in the 3-valued world mean definite values in all concrete 2-valued structures.

3.9 Bounded Structures

The presented shape analysis framework of Sagiv, Reps, and Wilhelm guarantees termination for programs containing loops by keeping the number of potential structures arising during the analysis finite. The mechanism ensuring this is the concept of *Canonical Abstraction.* Before explaining this concept in detail, we need the following definition.

Definition 11 (Bounded Structure) *A* bounded structure *over vocabulary $\mathcal{P} \cup \{sm\}$ is a structure $S = \langle U^S, \iota^S \rangle$ such that for all $u_1, u_2 \in U^S$ $(u_1 \neq u_2)$ there exists an abstraction predicate symbol $p \in \mathcal{A}$ such that $\iota^S(p)(u_1) \neq \iota^S(p)(u_2)$. We denote the set of such structures by B-STRUCT$[\mathcal{P} \cup \{sm\}]$.*

A direct consequence of this definition is that there is an upper bound on the size of structures $S \in$ B-STRUCT$[\mathcal{P} \cup \{sm\}]$, namely $|U^S| \leq 3^{\mathcal{A}}$. This limitation on the sizes of the structures stems from the fact that every abstraction predicate can take any of the three truth values for every individual.

The concept of canonical abstration provides a mechanism to obtain such bounded structures.

Definition 12 (Canonical Abstraction) *The* canonical abstraction *of a structure S, denoted by $t\_embed_c(S)$, is the tight embedding induced by the following mapping:*

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

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

We conclude this section with a reformulation of the *Abstraction Principle* which gives us a method of collapsing structures that always yields bounded structures. The idea is to partition individuals into equivalence classes according to their sets of unary abstraction-property values. Every structure $S^\natural$ is then conservatively represented by a condensed structure S in which each individual of S represents an equivalence class of individuals from $S^\natural$.

3.10 Abstract Semantics

We can now define a simple abstract semantics based on canonical abstraction and the reinterpretation principle. This semantics *collects* for every vertex v of the control-flow graph a finite set of 3-valued structures $StructSet[v]$ which describes at least all 2-valued structures in $ConcStructSet[v]$.

$$
StructSet[v] = \begin{cases} \{\langle\emptyset,\emptyset\rangle\} & \text{if } v = start \\ \left.\begin{array}{ll} & \bigcup\limits_{w\to v\in E(G),\ w\in As(G)} \{t\_embed_c\left([\![st(w)]\!]_3(S)\right) \mid S \in StructSet[w]\} \\ \cup & \bigcup\limits_{w\to v\in E(G),\ w\in Id(G)} \{S \mid S \in StructSet[w]\} \\ \cup & \bigcup\limits_{w\to v\in Tb(G)} \{S \mid S \in StructSet[w] \text{ and } S \models_3 cond(w)\} \\ \cup & \bigcup\limits_{w\to v\in Fb(G)} \{S \mid S \in StructSet[w] \text{ and } S \models_3 \neg cond(w)\} \end{array}\right\} & \text{otherwise.} \end{cases}
$$

The abstract meaning function for a statement w, denoted by $[\![st(w)]\!]_3$, is identical to $[\![st(w)]\!]$ with the exception of formulæ being evaluated in 3-valued logic.

3.11 Additional Concepts and Vocabulary

In this section, we present some additional concepts which will allow us to formulate a more refined abstract semantics. We also introduce some vocabulary and definitions that will be used in the description and definition of our analyses in Chapter 5.

3.11.1 Core and Instrumentation Predicates

We distinguish two types of predicates, *core* and *instrumentation* predicates. Core predicates are used to state facts about the store and to describe program semantics. The unary predicates introduced to represent pointer variables and the binary predicates modeling reference fields of objects we described earlier are examples of core predicates. Instrumentation predicates record information derived from other predicates and are defined in terms of a formula over core predicates. For the sake of abbreviation and readability, we also allow the defining formula of an intrumentation predicate to contain other instrumentation predicates, as long as formulæ do not become mutually recursive. Table 3.1 gives some examples of instrumentation predicates used in shape analyses of linked-list structures.

Formally, our set of predicates $\mathcal{P}$ is disjointly partitioned into the set of core predicates $\mathcal{C}$ and the set of instrumentation predicates $\mathcal{I}$. The *Instrumentation Principle* states that when S is a 3-valued structure representing the 2-valued structure $S^\natural$ by explicitly storing the values a formula φ has in $S^\natural$ in S, it is sometimes possible to extract more precise information from S than by just evaluating φ in S. This clearly motivates the use of instrumentation predicates. The increase of precision comes from several sources. Evaluating the defining formula of an instrumentation predicate may result in an indefinite value, while the predicate itself yields a definite value. We are allowed to use instrumentation predicates as abstraction predicates and thereby use them to partition the heap by keeping definite truth values and thus more precise information about the respective partitions. Another obvious argument is that a 3-valued structure in which some instrumentation predicates

Predicate	**Defining Formula**	**Intended Meaning**
$is[next](v)$	$\exists v_1, v_2.(v_1 \neq v_2 \wedge next(v_1, v) \wedge next(v_2, v))$	v is shared, that is two or more objects have a next-reference pointing to v.
$c[next](v)$	$\exists v_1.(n(v_1, v) \wedge next^*(v_1, v_2))$	v resides on a cycle.
$r[next, x](v)$ for each $x \in Var$	$\exists v_1.(x(v_1) \wedge next^*(v_1, v))$	v is reachable from program variable x via $next$-references.

Table 3.1: Examples of Instrumentation Predicates.

yield definite values may represent less structures than a structure with no instrumentation predicates.

3.11.2 Focus

Still, analyses might yield too many indefinite values. The *focus* operation is used to further increase precision. The key idea behind focus is to make those parts of the heap concrete on which the currently considered program statement operates. More formally, the focus operation takes a set of formulæ F and generates a set of structures on which every formula in F yields a definite value for all assigments. In general, the set of structures returned by focus may be infinite. However, [SRW02] gives an algorithm implementing a focus-operation for a certain class of formulæ needed for shape analysis that always returns a finite set of structures.

For example, consider a statement $x_1 = x_2$ where x_1 and x_2 are pointer/reference variables. The statement lets x_1 point to the heap cell x_2 currently points to. Hence, we might choose $F = \{x_2(v)\}$ so that for all heap cells we have a definite value whether or not x_2 currently points to this cell.

3.11.3 Coerce and Consistency Constraints

The focus operation may create structures that are inconsistent or contradictory and do not represent any concrete structures. The operation also may yield structures that are not as precise as they could be regarding to the information available in instrumentation predicates. The *coerce* operation is intended to sharpen such structures and discard structures that do not represent concrete structures. Behind these insights lies the *Sharpening Principle.*

Definition 13 (Sharpening Principle)
In any structure S, the value stored for $\iota^S(p)(u_1, \dots, u_k)$ should be at least as precise as the value of p's defining formula φ_p evaluated at $u_1, \dots, u_k$.

Furthermore, if $\iota^S(p)(u_1, \ldots, u_k)$ has a definite value and φ_p evaluates to an incomparable definite value then S is a 3-valued structure that does not represent any concrete structures at all.

In order to sharpen or discard structures coerce utilizes *compatibility constraints*, also referred to as *consistency constraints*.

Definition 14 (Compatibility Constraint) *A* compatibility constraint *is a term of the form $\varphi_1 \rhd \varphi_2$, 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 α are said to* satisfy *$\varphi_1 \rhd \varphi_2$, denoted by $S, \alpha \models \varphi_1 \rhd \varphi_2$, if whenever α is an assignment such that $[\![\varphi_1]\!]_3^S(\alpha) = 1$, we also have $[\![\varphi_2]\!]_3^S(\alpha) = 1$.*
We say that S satisfies $\varphi_1 \rhd \varphi_2$, denoted by $S \models \varphi_1 \rhd \varphi_2$, if for every α it holds that $S, \alpha \models \varphi_1 \rhd \varphi_2$.

Hence, if φ_1 evaluates to 1 but φ_2 evaluates to 0 the structure is discarded (eliminating structures not representing concrete structures). If φ_1 evaluates to 1 and φ_2 to 1/2 the interpretation of predicates is changed such that φ_2 also evaluates to 1 (sharpening of structures). In all other cases where φ_1 evaluates to 0 or 1/2, the coerce operation commits no changes on the structures.

3.12 The Shape Analysis Algorithm

We can now define a more refined abstract semantics which utilizes the before-mentioned focus and coerce operations.

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

4 B-Trees

In this chapter, we will briefly review the history of B-trees and their variations as well as the range of applications of such tree structures. We will give a formal definition of a B-tree that will be used throughout the rest of this thesis. Formal definitions are also given for the most important types and variations of B-trees: the 2-3-4 trees, the B+ trees and the B*-trees. We also discuss the basic algorithms for working with B-trees: contains, insert and delete.

4.1 Introduction

B-trees are balanced search trees. They can be viewed as a generalization of binary search trees. In a binary search tree, each node stores a key k as well as a left and a right pointer. The keys stored in the subtree pointed to by the left pointer are smaller than k, those in the subtree rooted at the node pointed to by the right pointer are greater. A node of a B-tree may store many keys, from a handful to several thousands, depending on the field of application. As with binary trees, each stored key is a dividing point separating the range of keys stored in the subtrees pointed to by adjacent child pointers. Hence, the branching factor of a B-tree may be very high. Figure 4.1 shows a simple B-tree. For now, we will stick to this intuitive definition of a B-tree. However, in Section 4.2, we will define B-trees formally.

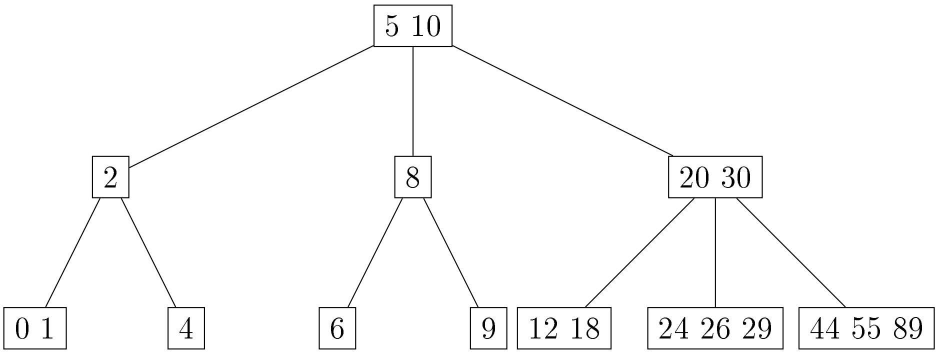


Figure 4.1: Graphical representation of a B-tree

B-trees were introduced by Bayer and McCreight in 1971 to organize and maintain an index for a dynamic random access file [BM72]. By the end of the seventies, B-trees and their variations have become a, de facto, standard for file organization. Comer gives an elaborate review on B-trees, their success in the seventies, and their variations, especially the B+ tree which we will discuss in Section 4.5, in [Com79].

Today, B-trees and their variants are still ubiquitous. In 2006, Hudzia et al. proposed a tree based peer-to-peer network architecture utilizing a slightly modified B+ tree structure for discovery and load-balancing operations [HKO06]. Two years earlier, Crainiceanu et al. proposed the P-tree, a new distributed fault-tolerant index structure for peer-to-peer networks, the key idea behind which is to maintain parts of semi-independent B+ trees at each peer [CLGS04].

Besides those new applications in peer-to-peer systems, B-trees are still used in virtually all file system implementations. The Reiser file system uses B+ trees to organize file system structures such as file state data or directories ([Rei, BCSZ03, BFH02]). HFS (Hierarchical File System) organizes its complete cataloging structure in a B-tree [Bru90]. The NTFS file system uses a B-tree structure for all folders to minimize the number of disk accesses that a hard drive must perform to find a file [Mic]. Even distributed file systems employ B-trees as shown with the Google File System [GGL03]. Further examples for file systems that utilize B-trees are XFS which uses B-trees for tracking free extents in the file system, to index directory entries, to manage file extent maps that overflow the number of direct pointers kept in the inodes, and to keep track of dynamically allocated inodes scattered throughout the file system [SDH$^+$96, BFH02] which enables XFS to support larger directories and efficiently manage free space [TV05]. As well as JFS [IBM], the Cedar File System (CFS), FSD - the reimplementation of CFS - [Hag87], and yFS where the latter makes use of B*-trees for representing large files and directories to keep directory traversals efficient [ZG03]. Recapitulatory, newer file systems often use B-trees to map logical file offsets to physical disk addresses, to manage disk space, and to perform efficient lookup for files in the name space [Isa03].

Relational databases (RDBs) also commonly rely on B-tree structures to organize indices in order to retrieve a small number of desired records without having to scan the entire database (or an entire table) [Kim02, IMM$^+$04]. The usage of B-trees to manage indices is effectively a standard in RDBs. Hence, pointing to literature documenting the usage of such trees in some concrete RDB would virtually end up in a listing of all existing relational database systems. However, we representative point out to MySQL [MyS07], Oracle Database [Dix01], and POSTGRES database management system [SK91] as examples for RDBs employing B-tree structures to store indices.

As a closing example of where B-tree structures are employed we point out that optimized index structures for Resource Description Framework (RDF) database systems also rely on B-trees [HD05]. Storing and querying RDF data is one of the basic tasks within any Semantic Web application. It seems that in this area a new field of application for B-trees and their ability to efficiently handle indices is evolving.

4.2 Formal Definition

4.2.1 B-tree

Definition 15 *Let $h \in \mathbb{N}_0$ and $t \in \mathbb{N}^{>1}$. A directed tree T constitutes a B-tree if T is either empty, i.e. $h = 0$, or has the following properties:*

1. *(P1) Each path from the root to any leaf has the same length h.*

2. *(P2) Each node with the exception of the root and the leaves has at least t children. The root is either a leaf or has at least 2 children.*

3. *(P3) Each node has at most* $2t$ *children.*

4. *(P4) Each node holds between* $t-1$ *and* $2t-1$ *index elements, except the root which may hold between 1 and* $2t-1$ *index elements.*

5. *(P5) For each node* v *holds: if* k *is the number of index elements stored with* v *and* v *is not a leaf then* v *has exactly* $k+1$ *children.*

6. *(P6) Within each node, index elements and child pointers are logically*[1] *stored as a sequence* $c_0 i_0 c_1 i_1 \dots i_{k-1} c_k$*. The index elements are sequential in increasing order, i.e.* $i_l < i_{l+1}$, $0 \leq l \leq k-2$*. In leaves , the child pointers (*c_l*,* $0 \leq l \leq k$*) are undefined. (See Figure 4.2 for a graphical representation of this node structure.)*

7. *(P7) Let* $\mathcal{I}(c_l)$ *be the set of all index elements stored in the subtree the (child) pointer* c_l *points to. Then for all nodes the following statements always hold:*
   $\forall i \in \mathcal{I}(c_0) : i < i_0$
   $\forall i \in \mathcal{I}(c_j) : i_j < i < i_{j+1}, \quad j = 1, 2, ..., k-2$
   $\forall i \in \mathcal{I}(c_k) : i_{k-1} < i$

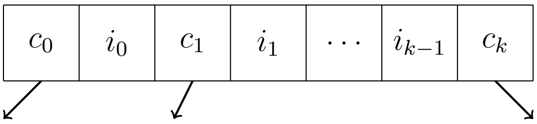


Figure 4.2: A node of a B-tree.

We call h the *height* of T and t the *degree* of T.

We note that the properties 1) to 3) enforce the shape of B-trees, while properties 4) to 7) enforce the organization and sortation of the index elements stored in the tree.
The given properties slightly differ from the original properties for B-trees given in [BM72]. The most significant difference is that we say the number of children for inner nodes is in $[t, 2t]$ and hence the number of stored index elements in $[t-1, 2t-1]$. Originally, nodes were allowed to have between $t+1$ and $2t+1$ children and to store between t and $2t$ index elements. However, our choice of these bounds aligns with that used in more recent literature, see for example [CLRS01]. We will see later that nodes storing between $t-1$ and $2t-1$ index elements can be split independent of index element insertions. The original definition can only split nodes if a new index element is to be inserted into an already full node. This also implies that with our definition implementations of the insertion-operation can be done more efficiently. Another difference is that we explicitly disallow B-trees of degree 1.
The height of such a B-tree is in $\mathcal{O}(\log n)$, where n is the number of index elements stored in the tree. The proof for this is fairly simple and given in [CLRS01]. One should keep in mind that the

[1] Implementations will probably use two separated data structures; one for the index elements and one for the pointers.

base of the logarithm can be very large and hence B-trees are commonly very flat compared to other trees.

We call a node of a B-tree *full* if there are currently $2t - 1$ index elements stored at this node.

4.2.2 Index Element

An index element is a tuple (κ, α), where $\kappa \in \mathcal{K}$ is a key from some totally ordered key universe $\mathcal{K}$ and α some data associated with this key.
We do not make too specific demands about the keys, although in practice simple integers should be an appropriate choice. However, we need that the keys are unique and that there is a strict total order on the keys. Or more formally:

$$\forall \kappa_1, \kappa_2 \in \mathcal{K} : \kappa_1 < \kappa_2 \oplus \kappa_2 < \kappa_1 \tag{4.1}$$

$$\forall (\kappa_1, \alpha_1), (\kappa_2, \alpha_2) : \kappa_1 \neq \kappa_2 \tag{4.2}$$

This strict order on the keys is used to imprint a strict order on the index elements by the following equivalence

$$(\kappa_1, \alpha_1) < (\kappa_2, \alpha_2) \Leftrightarrow \kappa_1 < \kappa_2 \tag{4.3}$$

4.3 Algorithms

B-trees are commonly needed to support element insertions, element deletions, and to provide a membership test to check whether a given element is stored in the tree. In this section, we will briefly discuss how those three operations contains, insert and delete must perform in order to be efficient and to preserve all B-tree invariants. The presented algorithms are based on the description given by Cormen et al. [CLRS01], rather than on the original algorithms given by Bayer and McCreight [BM72]. This is done for two reasons. Firstly, the *newer* algorithms exploit the modified properties of B-trees explained in Section 4.2 to improve their efficiency. Secondly, a survey on actual B-tree implementations and descriptions confirmed that the *newer* ones are commonly used.
This section, however, only gives a theoretical description of the B-tree operations. Actual implementations of all these operations using Java as the implementation language can be found in Appendix B.2.

4.3.1 The contains-Operation

The contains-operation performs a search in the tree in order to determine if a given index element (κ, α) is present. The operation starts its search at the root node. Each node, including the root, storing k index elements keeps a sequence $c_0 i_0 c_1 i_1 \ldots i_{k-1} c_k$ as defined in (P6). We choose i_l from this sequence such that $l = max\{m | i_m \leq (\kappa, \alpha)\}$. We now have to consider three cases:

Case 1 $i_l = (\kappa, \alpha)$. In this case, the algorithm has found the given index element and can terminate with an appropriate return value.

Case 2 We did not find such an i_l, i.e. $\{m|i_m \leq (\kappa, \alpha)\} = \emptyset$. In this case, we either continue our search by choosing a new i_l from the sequence stored at the node pointed to by c_0 and again considering these three cases. Or, if c_0 is not defined, the operation terminates with a return value indicating that (κ, α) is not stored in the tree.

Case 3 i_l is defined but not equal to the element we are looking for. In this case, we either continue our search by choosing a new i_l from the sequence stored at the node pointed to by c_{l+1} and again considering these three cases. Or, if c_{l+1} is not defined, the operation terminates with a return value indicating that (κ, α) is not stored in the tree.

Obviously, for finding i_l we can either use a linear-search on $c_0 i_0 c_1 i_1 \ldots i_{k-1} c_k$ or perform a binary search on this sequence, resulting in a running time in $\mathcal{O}(t \cdot h)$ and $\mathcal{O}(\log t \cdot h)$, respectively. However, in most applications the critical factor is the number of node accesses, i.e. the height h. Hence, we may just say, both approaches need $\mathcal{O}(h)$ node accesses and therefore there is no harm in implementing the easier approach which uses a linear-search.

4.3.2 The insert-Operation

When inserting new elements, we want to exploit that our B-trees can split full nodes without the need to first insert an element into the node we want to split. With the original definition, full nodes could only be split when inserting a new element into. We start by introducing a split-operation that we will afterwards use within our insert-operation.

The split-Operation

Let x be a full node and $p(x)$ the parent node of x. We assume $p(x)$ to be non-full. Hence, x stores exactly $2t - 1$ elements and $p(x)$ at most $2t - 2$. We can split x into two nodes without violating any B-tree invariant. Let $c_0 i_0 c_1 i_1 \ldots c_{t-1} i_{t-1} c_t \cdots i_{2t-2} c_{2t-1}$ be the sequence of index elements and child pointers of node x. We create a new node x' which stores the sequence $c_t \cdots i_{2t-2} c_{2t-1}$ and replace the sequence stored at x by $c_0 i_0 c_1 i_1 \ldots c_{t-1}$. The median element i_{t-1} is moved to $p(x)$ as a dividing point for x and x'. Figure 4.3 and Figure 4.4 illustrate this splitting operation. We can convince ourselves that all properties of B-trees are still intact:

1. (P1) still holds because x' has the same parent node and thus the same height as x. All child nodes of x have the same height as before regardless whether they moved to x' or stayed in x. We therefore (a) do not change the height of any existing node and (b) x' will be inserted at the same height as x and be a leaf node if x was a leaf node.

2. (P2), (P3), (P4), and (P5) still hold; both x and x' have exactly t child nodes and $t - 1$ elements, $p(x)$ has at most $2t - 2 + 1 = 2t - 1$ elements stored and $2t - 1 + 1 = 2t$ child nodes.

3. (P6) and (P7) still hold due to the fact that we set the sequences stored at x and x' to one consecutive part of a valid sequence starting with a child pointer.

Obviously, splitting a node needs $\mathcal{O}(1)$ node accesses.

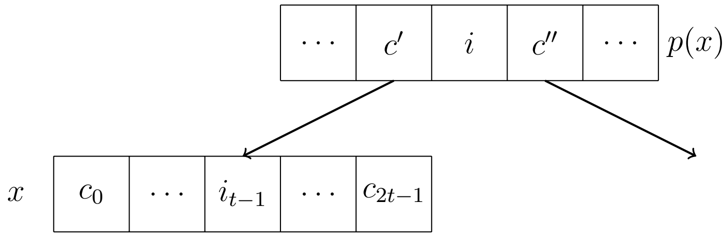


Figure 4.3: Situation before splitting.

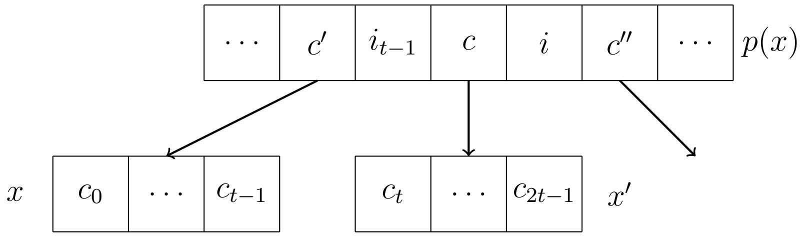


Figure 4.4: Situation after splitting.

The insertNonFull-Operation

Let us assume we want to insert an index element i into the tree rooted at node x and we know that x is not full. We define an operation insertNonFull that operates as follows.
First, the algorithm checks whether x is a leaf node.

Case 1 x is a leaf node. We then insert i into the sequence $c_0 i_0 c_1 i_1 \dots i_{k-1} c_k$ stored at x in such a manner that property (P6) is not violated.

Case 2 x is an inner node. In this case, we make a branching decision, i.e. we determine into which child node c_l of x the element i has to be inserted in order to preserve (P7). If c_l is already full, we split c_l into $c_l^<$ and $c_l^>$ and recursively call insertNonFull with arguments i and $c_l^<$ or $c_l^>$, depending on whether i is greater or less than the median element of c_l. In the remaining case that c_l is not already full, we recursively call insertNonFull with arguments i and c_l.

The Final insert-Operation

The last thing we have to do is get rid of the assumption that we start the insertion with a non-full node. The final insert-operation expects as parameters an index element i and the root r of the B-tree i is to be inserted into. The operation first checks whether r is already full which results in two possible cases.

Case 1 r is full. In this case we have to create a new root-node x with no elements stored and the former root r as its leftmost child, i.e. c_0 of x points to r. We then split r and call insertNonFull with x and i as arguments.

Case 2 r is not full, hence we can directly proceed with our insertNonFull-operation.

We observe that our insert-operation does not violate any B-tree invariants and is able to insert an index element into a B-tree in a single pass down the tree. Furthermore, insertions always take place at leaf nodes. The original insertion algorithm would - in the worst case - need two passes. One down the tree, through already full nodes that could not be splitted to an already full leaf which is splitted after the new element was inserted. The median element that has to be passed to the parent node would cause the parent to be splitted, too. This splitting would propagate upwards the whole tree, up to the root. Resulting in a second pass up the tree.
The number of node accesses performed by the insert operation lies in $\mathcal{O}(h)$.

4.3.3 The delete-Operation

Similar to the insert-operation, we want to have the delete-operation perform a single pass down the tree. For this reason we must ensure that only such nodes are entered by the algorithm that store more than the minimum number of index elements. That is - except for the root node - at least t elements.

The move- and join-Operations

We first consider what to do when we need to enter a subtree rooted at a node with the minimal number of elements.
Let x be a node, $p(x)$ the parent node of x and $s = c_0 i_0 c_1 i_1 \dots c_{l-1} i_{l-1} c_l i_l c_{l+1} \dots i_{k-1} c_k$ the sequence stored at $p(x)$ where c_l is the child pointer pointing to x. Let further $s' = c'_0 i'_0 c'_1 i'_1 \dots i'_{t-2} c'_{t-1}$ be the sequence stored at x. Our tree properties guarantee that at least one sibling node exists. Two cases may arise.

Case 1 One of the nodes pointed to by c_{l-1} or c_{l+1} stores more than $t-1$ elements. Our restructuring of the tree will then move the greatest or the least element of such a sibling of x to $p(x)$, replacing i_l which moves to x as the least or greatest element there. Figure 4.5 and 4.6 illustrate this operation for the case that we move an element from the right sibling pointed to by c_{l+1}. The procedure works symmetrically for taking an element from the left sibling. Note that the element removed from the sibling node was a dividing point for two adjacent child nodes one of whose child pointer has to be moved to x.

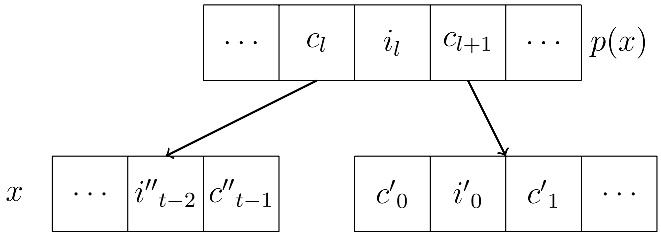


Figure 4.5: Situation before moving an element to x.

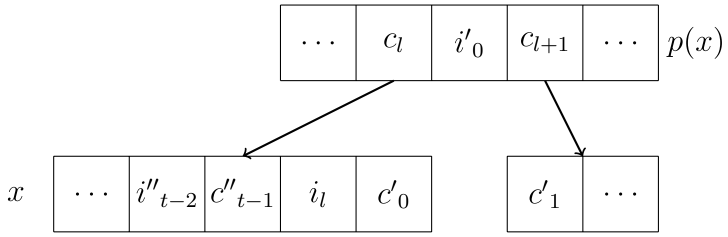


Figure 4.6: Situation after moving an element to x.

Case 2 No sibling stores more than $t-1$ elements. In this case, we join x with one of its siblings, resulting in a node storing $2t-1$ elements. Joining two neighboring nodes is in some sense the exact reversal of the split-operation. Let x and y be the nodes we intend to join, c_x and c_y, respectively, the pointers to x and y, and i their dividing element in their common parent node. Let s_x and s_y be the sequences of child pointers and index elements stored at x and y, respectively. The join-operation removes i and c_y from the sequence stored at the parent node and then sets the sequence stored at node x to $s_x i s_y$. Figures 4.7 and 4.8 illustrate the aforementioned.

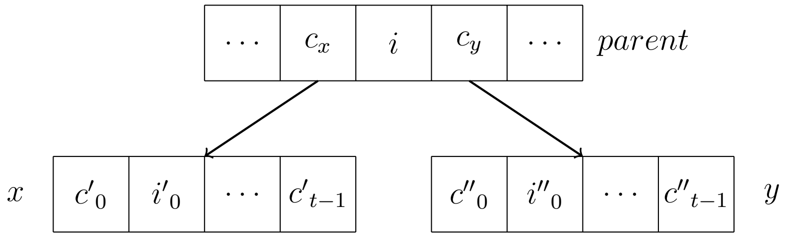


Figure 4.7: Situation before joining x and y.

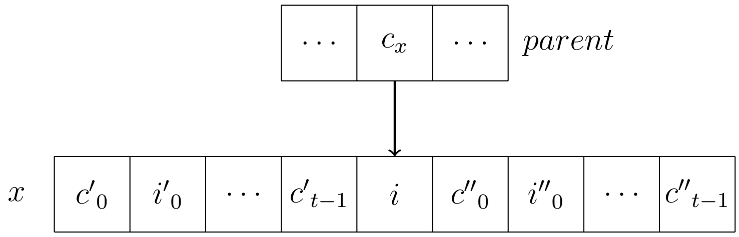


Figure 4.8: Situation after joining x and y.

In both cases node accesses are in $\mathcal{O}(1)$.

The Final delete-Operation

The delete-operation works similar to the (final) insert-operation. It starts at the root (which holds always enough elements to allow for a deletion), makes a branching decision, and ensures that the next node to enter holds more than the minimum number of elements. To ensure this the algorithm uses the move- and join-operations as described in the previous subsection. Once the element to be removed, call it i_r, is found, the algorithm again has to distinguish between two possible cases.

Case 1 The current node is a leaf. In this case, the algorithm can just delete i_r from the sequence stored at the current node.

Case 2 The current node is an inner node. There exist again two possibilities. i_r can be replaced by its predecessor or successor if the nodes c_r and c_{r+1}, respectively, hold more than the minimum number of elements. By predecessor of some index element $i_\S$ we denote the index element preceding $i_\S$ in the total order imprinted on them by their keys. Successor of $i_\S$ denotes, analogously, the element succeding $i_\S$. I.e. successor and predecessor of index elements do not denote index elements logically stored next to those elements. The predecessor or successor of course has to be deleted at its old position. This replacing is allowed because the predecessor of i_r is the greatest index element in the subtree rooted at c_r and its successor the least index element contained in the subtree rooted at c_{r+1}. Hence, property (P7) will not be violated. If c_r and c_{r+1} store only $t-1$ elements the algorithm joins those two nodes. Afterwards, i_r can be removed from the newly created node (which has $2t-1$ elements).

The last problem we have to deal with regarding the delete-operation is that it is still possible to remove all elements from a non-leaf root node. This happens if the root node holds just 1 element and its two child nodes are joined. In this case, we will delete the current root node and set its only child, pointed to by c_0 after the join, to be the new root.

Since most of the elements stored in a B-tree reside in leaves, in practice, delete operations most of the time act in one pass down the tree. When deleting an element in an inner node, the procedure might have to back up after unsuccessfully trying to locate the predecessor in a node holding more than $t-1$ elements. Overall, the number of node accesses needed to delete an index element from a B-tree is in $\mathcal{O}(h)$.

4.4 2-3-4 Tree

We call a B-tree of degree $t = 2$ a 2-3-4 tree. Obviously, with $t = 2$ each node of such a B-tree can only have 2 $(= t)$, 3 or 4 $(= 2t)$ child nodes, which explains the name 2-3-4 tree.
This is also the simplest form of a B-tree as we do not allow B-trees of degree $t = 1$.

4.5 B+ Tree

A B+ tree is a B-tree that stores only keys of index elements at its inner nodes and complete index elements, i.e. keys and data, at its leaves.

Definition 16 *Let $h \in \mathbb{N}_0$ and $t \in \mathbb{N}^{>1}$. A directed tree T constitutes a B+ tree if T is either empty, i.e. $h = 0$, or has the following properties:*

1. *(P1) Each path from the root to any leaf has the same length h.*

2. *(P2) Each node with the exception of the root and the leaves has at least t children. The root is either a leaf or has at least 2 children.*

3. *(P3) Each node has at most $2t$ children.*

4. *(P4$^+$) Each node holds between $t-1$ and $2t-1$ keys from the key universe $\mathcal{K}$ of the index elements, except the root which may hold between 1 and $2t-1$ keys.*

5. *(P5$^+$) For each inner node v_i holds: if k is the number of keys stored with v_i then v_i has exactly $k+1$ children. For each leaf node v_l holds: if k is the number of keys stored with v_l then v_l has exactly k pointers to data objects associated with the stored keys.*

6. *(P6$^+$) Within each non-leaf node keys and child pointers are logically stored as a sequence $c_0\kappa_0c_1\kappa_1\ldots\kappa_{k-1}c_k$. The keys are sequential in increasing order, i.e. $s_l < s_{l+1}$, $0 \leq l \leq k-1$. In leaf nodes the sequence consists of keys and pointers to their associated data, i.e. leaves store a sequence $\kappa_0\alpha_0'\kappa_1\alpha_1'\ldots\kappa_{k-1}\alpha_{k-1}'$. We denote by α_l' a pointer to the data object α_l.*

7. *(P7$^+$) Let $\mathcal{S}(c_l)$ be the set of all keys stored in the subtree the (child) pointer c_l points to. Then for all nodes the following statements always hold:*
   $\forall\kappa \in \mathcal{S}(c_0) : \kappa \leq \kappa_0$
   $\forall\kappa \in \mathcal{S}(c_j) : \kappa_j \leq \kappa < \kappa_{j+1}, \quad j = 1, 2, ..., k-2$
   $\forall\kappa \in \mathcal{S}(c_k) : \kappa_{k-1} \leq \kappa$

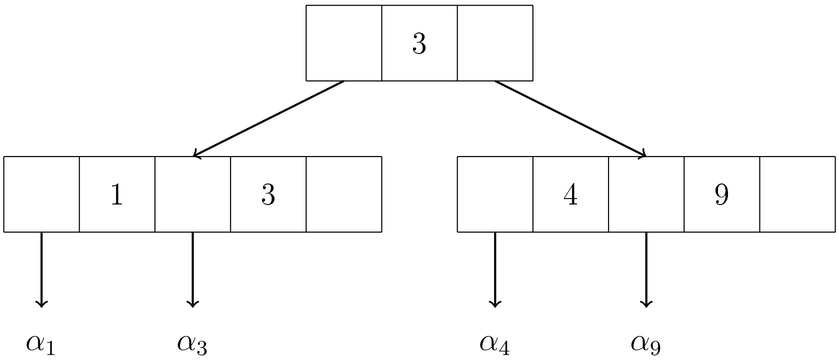


Figure 4.9: A B+ tree.

B+ trees have some advantages over general B-trees. With B+ trees element deletions always occur at leaves which allows for a simpler delete-operation. Also, not storing data in inner nodes leaves more storing capacity for keys and thus may lead to a higher branching factor which decreases the height of the tree and thus reduces the number of node accesses performed by the operations, resulting in better running times.

4.6 B*-tree

We define a B*-tree as a B-tree for which the stronger condition holds that each node except its root is at least 2/3 full. Hence,

Definition 17 *A B*-tree is a B-tree for which property (P4) is replaced by*
(P4) Each node holds between* $2/3 \cdot (2t-1)$ *and* $2t-1$ *index elements except the root which may hold between 1 and* $2t-1$ *index elements.*

5 Shape Analysis of B-Trees and Variations of B-Trees

5.1 Introduction

In this chapter, we present our shape analysis of 2-3-4 trees. We also establish - although just theoretically - how a shape analysis on general B-trees can be implemented.
As 2-3-4 trees are a specialization of B-trees, namely their simplest form, a shape analysis of this structure can be generalized to a shape analysis of general B-trees. We could also consider binary trees to be a special form of 2-3-4 trees, by allowing exactly 2 child nodes per node and discarding some balancing properties. By doing so it seems an obvious approach to generalize predicates used in the shape analysis of binary trees and structures internally using binary trees in order to cope with 2-3-4 trees. Additionally, some new predicates have to be introduced to keep track of balancing properties. By keeping all predicates general enough to cope with binary and 2-3-4 trees, shape analysis of binary trees can be redone with the same set of predicates used to verify our 2-3-4 tree implementation.

The remainder of this chapter is organized as follows. After a short section on how we generated the control-flow graphs needed for shape analysis from Java implementations, we present in detail our shape analysis of 2-3-4 trees. The last section of this chapter describes how our approach to 2-3-4 trees can be generalized and adapted to handle general B-trees.

5.1.1 Transforming Java into TVP

The implementations we are going to perform our shape analysis on are done in Java. TVLA, however, requires the control-flow graph of the program to analyze supplied in a special format called TVP (**T**hree **V**alued **P**rogram) [MS04]. Transforming Java source code to TVP by hand is a tedious and error-prone task. And even if we would translate a Java program by hand to its control-flow graph, we would have to show that this control-flow graph is really corresponding to the control-flow graph of the compiled Java program. To alleviate this procedure we could try to translate Java byte code to TVP. But this work - done by hand - is still error-prone and tiresome, especially considering the complexity of some of the operations on B-trees.
To overcome those problems, we utilize J2TVLA, a framework for building translation programs that transform Java byte code (.class files) to TVP format (.tvp files) [MSSY02].

Statement of the Shape Analysis The shape analysis is performed not directly on our Java implementations but on TVP code obtained from the Java classes. Therefore, we feel compelled to discuss *what* such an analysis does state about the underlying Java implementation.

Figure 5.1 shows how Java source code is transformed to the TVP code used in the analysis.

$$\begin{tikzcd}
P_{Java} \arrow[r, "\text{javacc}", "\text{program synthesis}"'] & P_{Class} \arrow[r, "\text{J2TVLA/SOOT}", "\substack{\text{program synthesis/} \\ \text{program migration}}"'] & P_{SootIR} \arrow[loop above, "\substack{\text{J2TVLA/SOOT} \\ \text{program transformation}}"] \arrow[r, "\text{J2TVLA/SOOT}", "\text{program migration}"'] & P_{TVP}
\end{tikzcd}$$

Figure 5.1: Sketch: Transformation steps from Java source code to input code for the analysis framework. We denote by P_{Java} the Java source code of a program, by P_{Class} its compiled byte code, by P_{SootIR} its intermediate representation within the Soot framework and J2TVLA, and by P_{TVP} the program in TVP code.

The first transformation is done by the Java compiler (javacc): compilation of the Java sources. J2TVLA then relies on the Soot framework to obtain an intermediate representation of the program from the Java byte code [VRHS$^+$99, Raj98, MSSY02]. After a dead variables elimination, J2TVLA generates a TVP program from this intermediate representation.
As the figure shows, we distinguish three different kinds of transformations. *Program syntheses* transform programs from one level of abstraction to another. Java source code is on a higher abstraction level than Java byte code, hence the compilation done by javacc is a program synthesis. J2TVLA generates its internal representation from a 3-address code representation of byte code (Jimple) that again is on a higher abstraction level than byte code itself. Our dead variables elimination transforms the program to an observably equivalent program in the same representation language. We call such a transformation just *program transformation*. Transformations from one representation language to another are called *program migrations* if both representations are on an equal level of abstraction.

Before putting the TVP program and the Java source code into relation, we start by defining some vocabulary and concepts that allow us to formulate our concluding claim.

Definition 18 (observable behavior) *Let P be some program. We call the sequence of outputs $s_{out} := P(s_{in})$ that P generates for a given sequence of inputs s_{in} the* observable behavior *of P.*

Definition 19 (semantics-preserving) *A program transformation T that transforms a given program P into a program $T(P)$ is* semantics-preserving *if the observable behavior of the program $T(P)$ is equal to that of P.*
Formally speaking, let P be a program and $T(P)$ the program resulting from applying transformation T to P then T is semantics-preserving if and only if:

$$\forall P \ . \ \forall \text{ input sequence } s_{in} \ . \ P(s_{in}) = (T(P))(s_{in})$$

These are more or less standard definitions in work dealing with program analysis and program transformation. However, they are built up on a very simple and limited concept of what a (computer) program is. Nonetheless, these definitions are suitable for our scenario.

Definition 20 (making a property visible) *We say, we* make a program property F visible *if we add F to the observable behavior of the program.*

We note that making a property F of program P visible results in a new program P'. Furthermore, $P(s_{in})$ differs from $P'(s_{in})$ such that $P'(s_{in})$ additionally contains output dependent from or describing F.

Definition 21 (preserved property) *Let T be a semantics-preserving transformation. A program property F of program P is called a* preserved property under T of P *if after making F visible, resulting in program P', it holds that*

$$T(P')(s_{in}) = (T(P)')(s_{in})$$

for all input sequences s_{in}. Where $T(P)'$ denotes the program obtained from program $T(P)$ by making F visible.

Example: Let P be a program with program variables u, v, x where u and v are pointers to two heap cells joined in a linked list such that u has a pointer to v. Assume x to be a dead integer variable, currently storing the value 3. Let T be a transformation that sets dead variables to 0. T is a well-known semantics-preserving transformation. The property *u and v are linked* is preserved under T as P' and $T(P)'$ would both show them linked in their output. However, statements about the value of x are not preserved under T as P' and $T(P)'$ would generate different outputs due to the different values of x.

We can now formulate the following claim.

Claim 1 *The structure of trees built in the heap is a preserved property of the analyzed tree operations under all transformations used to convert Java source code to TVP code.*

Unfortunately, we are not able to prove this claim formally. However, we assume that (a) the Java compiler works correctly and produces byte code that builds heap structures according to the Java source code; and (b) the Soot framework correctly parses this byte code and builds an appropriate intermediate representation. We can now argue that our dead variables elimination preserves the structure of trees in the heap. Only program variables can be dead variables. Heap structures, however, are built by references between objects. We then have to assume that J2TVLA transformes Soot's intermediate representation of the program into TVP code such that all heap manipulating operations are preserved.
TVP code describes a control-flow graph where edges are labeled with actions - or better action macros - which represent program statements. Hence, actions define the semantics of the TVP program and we have to make sure that those actions preserve the effects of the heap manipulating operations of the previous program representation. Therefore, we have to finally argue about the correctness of the action definition. Actions are defined in terms of update formulæ or preconditions. We will later, when presenting our analyses, describe all formulæ used to capture the effects of actions on the heap states and argue their correctness.

Usage of J2TVLA J2TVLA provides a default engine that generates a control-flow graph containing only skip-statements. Creating a meaningful translation depends on the predicates used in the shape analysis and is thus left to the user of J2TVLA.
The default engine does categorize common language constructs (assignments, conditional branches, etc.) and translates those using output text specified via a properties files which contains per default only skips. Hence, most of the time adjusting the default properties file to one's predicates suffices to create appropriate translations of class files with J2TVLA.
However, if more complex translations are needed J2TVLA's default translator class[1] can be extended and the default treatment of language constructs can be overwritten to meet one's requirements.
As we had to change the treatment of some language structures to cope with our B-tree implementations, we implemented our own translator class - BTreeTranslator - which changed some of J2TVLA's default behavior. The source code of this class as well as the properties file we used to translate our class files to the TVP format can be found in the Appendix B.

Modifications on J2TVLA During the work on this thesis, we revised J2TVLA's live variable analysis and its corresponding transformation on the control-flow graph. Our intention was to set all variables to null as soon as they become dead in order to speed up the shape analysis using TVLA. A speed-up can be achieved as program variables correspond to pointer predicates which in turn are used as abstraction predicates. Hence, true pointer predicates can lead to finer abstractions in the form of more complex logical structures. In the case of dead variables, more complex structures due to predicates corresponding to such variables are unwanted.
In our revised version, J2TVLA adds for every dead variable a set-null statement to the produced TVP code. By assigning null to dead variables, we allow for smaller structures during the shape analysis as dead variables - after the transformation - cannot point to any heap cells and thus prevent abstracting these anymore. Our analysis seems to become non-conservative by setting pointer predicates corresponding to program variables to 0 instead of setting them to 1/2. However, dead variables correspond to dead predicates which we are allowed to set to 0. Setting dead predicates to 0 was already proposed in earlier work [Rei05, Man03].
During the implementation of our translator class we also did some refactoring on J2TVLA's default translator class. This work consisted mainly of extracting methods following the well-known Extract Method technique [FBB$^+$99]. This simple refactorization allowed for a more selective extention of the default translator by the BTreeTranslator class.
Our refactorizations and improvements will probably become part of the J2TVLA distribution.

5.2 Validation of a 2-3-4 Tree Implementation

In our Java implementation, 2-3-4 trees consist of Node234 objects. Every such node maintains pointers to its up to four child nodes and to up to three IndexElement objects stored at the node. Index elements store an integer value and maintain a pointer to some data object. During the shape analysis phase, logical variables, i.e. individuals, may represent either a node or an index element. Therefore - contrary to previous work [LARSW00, Rei05] - we have to distingiush two

[1] j2tvla.Translator

```
public class Node234 {
    public Node234 left;
    public Node234 cleft;
    public Node234 cright;
    public Node234 right;

    public IndexElement ie1;
    public IndexElement ie2;
    public IndexElement ie3;
}
```

(a) Node class

```
public class IndexElement {
    public int key;
    public Object content = null;
}
```

(b) Index element class

Figure 5.2: Java classes representing nodes and index elements, respectively.

types of logical variables which results in some additional and more complex predicates. Our set of selectors Sel_G is simply the union of the sets of selectors of each class. In our case, the union of $Sel_{Node234} = \{left, cleft, cright, right, ie1, ie2, ie3\}$, the set of all references maintained at an Node234 object, and $Sel_{IndexElement} = \{content\}$, the set of references of an IndexElement object. Integer and boolean attributes of objects are directly associated with the corresponding logical variables.

Our level of abstraction during the shape analysis additionally allows us to ignore the *content* reference of index elements, so that for the remainder of this work, we set the set of selectors, denoted by Sel, to $Sel_G \backslash \{content\}$. By $Vars$ we denote the set of program variables.

5.2.1 Contains-Operation

We consider a membership test as described in 4.3.1 and implemented as shown in Figure 5.3. In our shape analysis, we limit ourselves to prove that if the contains operation is performed on a valid 2-3-4 tree structure which

1. stores an index element equal to the search argument then a reference/pointer to the node at which this element is stored is returned by the operation

2. does not store an index element equal to the search argument then a null pointer is returned by the operation

Hence, we do not prove that height properties are preserved. At this point we argue that a membership test just traverses the structure without doing any restructuring and thus, if operating on a valid structure, will preserve this validity. We also do not attempt to verify that our contains operation runs in $\mathcal{O}(h)$ by showing that at most h nodes are visited during the traversal.

Predicate Definitions

As outlined in the introduction to this chapter, we borrowed predicates from existing work on binary trees and generalized them to additionally handle 2-3-4 trees. The original definitions of those predicates stem from [Rei05].

```
public static Node234 contains(
                         Node234 node,
                         IndexElement ie ) {
  Node234 result = null;
  while( node != null ) {
      if(    node.ie3 != null
          && node.ie3.key <= ie.key) {
        if( node.ie3.key == ie.key){
          result = node;
          node = null;
        }
        else
          node = node.right;
      }
      else if(    node.ie2 != null
               && node.ie2.key <= ie.key ) {
        if( node.ie2.key == ie.key){
          result = node;
          node = null;
        }
        else
          node = node.cright;
      }
      else if(    node.ie1 != null
               && node.ie1.key <= ie.key ) {
        if( node.ie1.key == ie.key){
          result = node;
          node = null;
        }
        else
          node = node.cleft;
      }
      else if(    node.ie1 != null
               && node.ie1.key > ie.key )
        node = node.left;
      else
        node = null;
    }
    return result;
}
```

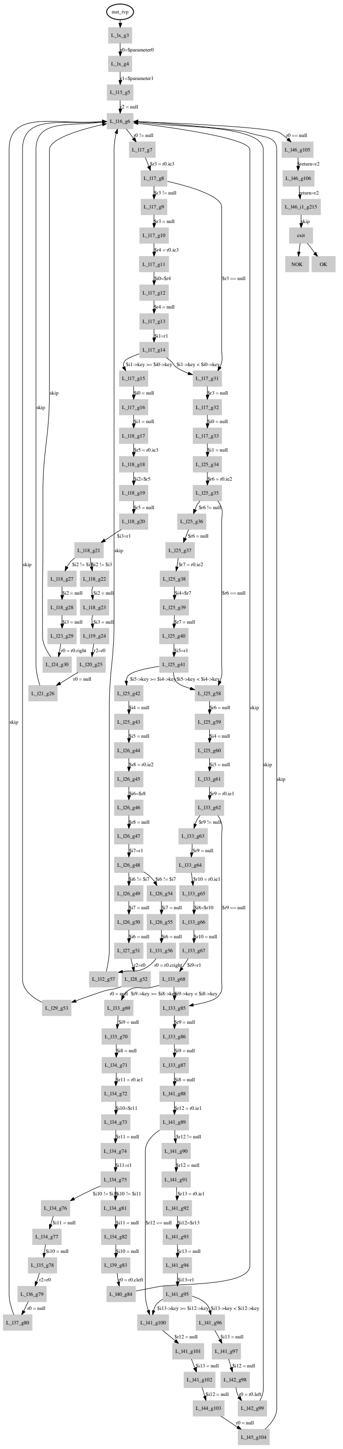


Figure 5.3: Java implementation of contains with its corresponding control-flow graph.

Core Predicates The following core predicates were used to logically represent heap states. In our abstraction, logical variables correspond to objects of the Java program. We also say these variables correspond to heap cells to keep in line with the nomenclature of previous work. In order to do so, we require each object to fit in one (abstract) heap cell.

To express that a program variable z points to some object v, we define a unary predicate $z(v)$. To capture the fact that selector sel of object v_1 points to v_2, we define a binary predicate $sel(v_1, v_2)$. Concrete values of the *key* attribute of index elements are modeled by the *kge*-predicate. Where $kge(v_1, v_2)$ holds if and only if the value of the *key* field of object v_1 is greater than or equal to the value of the *key* field of v_2. We assume *kge* to be reflexive and transitive during the analysis. For variables corresponding to tree nodes which do not maintain a *key* field, *kge* is not defined and may at any time be set to any truth value.

For technical reasons[2], we introduce the unary predicate *heapcell* which holds for all heap cells, i.e. for all logical variables. The predicate *heapcell* is not used as an abstraction predicate.

Compared to the corresponding core predicates of binary trees very little has changed. The set *Sel* became larger and $dle(v_1, v_2)$ was replaced by $kge(v_1, v_2)$ to match J2TVLA's normalization of comparisons.

Table 5.1 gives an overview of the core predicates.

Predicate	Intended Meaning
$z(v)\ \forall z \in Vars$	Pointer variable z points to heap cell v.
$sel(v_1, v_2)\ \forall sel \in Sel$	Selector *sel* of v_1 points to v_2.
$kge(v_1, v_2)$	The key(s)/data stored at heap cell v_1 is greater than or equal to the key(s)/data stored at heap cell v_2.
$heapcell(v)$	Holds for all v.

Table 5.1: Core Predicates

Instrumentation Predicates We used the following instrumentation predicates to gain precision and model properties of 2-3-4 trees not expressible by core predicates alone. The binary predicate $down(v_1, v_2)$ is used to express that v_1 has a reference pointing to v_2. The antisymmetric predicate $downStar(v_1, v_2)$ is defined as the reflexive transitive closure of *down* and can thus be used to record reachability between heap cells or objects. To record reachability from variables to heap cells or objects, we use the unary predicate $r[x](v)$, where $x \in Vars$ is some variable name. Hence, $r[x](v)$ holds if and only if heap cell v can be reached from the heap cell that variable x currently points to. We declare the reachability predicates as non-abstracting predicates, with the exception of the predicate modeling reachability from the program variable `node` which we do use as an abstraction predicate. The predicate $isStore(v)$ states that heap cell v holds an index element, i.e. an object at which a key value is stored. This predicate is not used as an abstraction predicate. With the *isStore*-predicate heap cells representing index elements and tree nodes, respectively, can be told apart during the analysis. In order to exploit this and the fact that

[2] In consistency rules, the free variables of the body and the head must match exactly [MS04]. The predicate *heapcell* is used to artificially add a free variable used in the head to the body.

cells representing index elements cannot have any successor nodes, we define another instrumentation predicate *storeProp*. The unary non-abstracting predicate *storeProp*(v) states that v has the *store property* which means if *isStore*(v) holds it must hold that $\neg\exists\ u\ down(v,u)$. In other words, being an index element implies not having any references pointing to other objects. We note again, that in our abstraction, we do not consider the content attribute of the IndexElement class.
To achieve the needed precision for the analysis we have to introduce predicates capturing general facts about tree structures. The first thing we wanted to express is that some node u lies logically left to some other node v. By *lying logically left* we mean the following. The logical structure of an 2-3-4 tree node can - following from our definitions in Chapter 4 - be thought of as depicted in Figure 5.4 where an ordering is impressed on the reference pointers. Now, our terminology becomes obvious. If we say a heap cell u lies logically left of some cell v, we mean that the reference pointing to u precedes the reference pointing to v in this impressed ordering. We utilize the predicate $leftOf(u,v)$ to express that u is (logically) left-adjacent to v, i.e. u is directly followed by v in the imprinted logical ordering. In the definition of this instrumentation predicate, we have to additionally consider the case of the node being a leaf. In leaf nodes, i.e. in nodes where no left pointer exists, $ie1$ is left of $ie2$ and $ie2$ left of $ie3$.The predicate $leftOfStar(u,v)$ is defined as the non-reflexive transitive closure of the $leftOf$-predicate. We extend this local view on a single node to a global view by introducing a predicate capable of expressing that a heap cell v is situated in a subtree (logically) to the right of u. Technically, we do this by defining $rightSubTree(u,v)$ to hold if there exists some w such that $leftOfStar(u,w) \wedge downStar(w,v)$ holds. Analogous, we introduce the predicates $rightOf(u,v)$, $rightOfStar(u,v)$, and $leftSubTree(u,v)$.
We are now ready to establish what it means for a structure to be a tree. From the many equivalent definitions of treeness[3] that are commonly used we choose to derive the following equivalent definition:

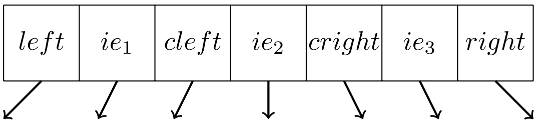


Figure 5.4: Logical representation of a 2-3-4 tree node.

Definition 22 *Let* $T=(V,E)$ *be a connected directed graph.* T *is a tree if and only if* $\forall\ v_1,v_2,v_3 \in V$

$$leftOfStar(v_1,v_2) \Rightarrow \neg(downStar(v_1,v_3) \wedge downStar(v_2,v_3))$$

[3] Let $T=(V,E)$ be an undirected graph then the following statements are equivalent:

1. T is a tree.
2. T is is acyclic and connected.
3. Any two vertices of T are connected by a unique path.
4. T is connected and $T-e$ is not connected for all $e \in E$.
5. T is acyclic and $T+e$ is not for all $e \in \left(\binom{V}{2}\backslash E\right)$.

The equivalence of this definition to a commonly used definition of a directed trees is given in Appendix A.

Besides knowledge about the structural properties of trees we need some knowledge about the ordering of stored values during the analysis. We therefore use the predicate $inOrder()$ to express that if $leftSubTree(u, v)$ holds then the value of a key stored at v is strictly less than the value of a key stored at u. While $rightSubTree(u, v)$ implies that the key stored at u is strictly less than that stored at v. To collect heap cells identified as storing values greater than some given value in one summary node during the analysis, we add the predicate $kge[z, left](v)$ to our set of abstraction predicates. For collecting cells that store values less than a given value, we also add a predicate $kge[z, right](v)$. Where, in both cases, z denotes a program variable and the predicates read as *according to the ordering imprinted by the kge predicate, the heap cell pointed to by z resides in the tree left and right, respectively, of v.*

Our analysis still needs some knowledge about the properties of 2-3-4 trees. We therefore introduce a predicate $p2\_5(v)$ to model properties (P2) and (P5) as defined in Chapter 4. Properties (P3) and (P4) are given implicitly by the selectors and (P1) is not relevant for our analysis of the contains method. The remaining properties (P6) and (P7) that define the ordering of stored values within the tree are already modeled by the $inOrder()$-predicate.

To express that a given key value equal to the value associated with heap cell u is stored at a heap cell v or at a heap cell reachable from v we use the binary predicate $isElement(u, v)$.

We also need to state that the kge-predicate models the $\geq$-relation on numbers. This is done by the $greRelation()$-predicate which states that if $kge(a, b)$ does not hold then $kge(b, a)$ must hold. Table 5.2 lists all instrumentation predicates with their defining formulæ and a brief description of their intended meanings.

Compared to the instrumentation predicates used in the analyses of binary trees, we observe that the $downStar[sel](v)$- ($sel \in \{left, right\}$) predicates were not redefined to cope with 2-3-4 trees. Those predicates were used to record reachability where the first selector is given (sel). While traversing a binary tree the first selector chosen from some node v determines whether we are descending into the left or the right subtree rooted at v. Hence, those predicates were subsumed by our $leftSubTree$- and $rightSubTree$-predicates.

Predicate	**Defining Formula**	**Intended Meaning**
$down(v_1, v_2)$	$\bigvee_{sel \in Sel} sel(v_1, v_2)$	There is a pointer reference from v_1 to v_2
$downStar(v_1, v_2)$	$down^*(v_1, v_2)$	Records reachability; v_2 is reachable from v_1
$r[x](v)\ \forall x \in Vars$	$\exists v_1.\,(x(v_1) \wedge downStar(v_1, v))$	Records reachability from a (pointer) variable x
$isStore(v)$	$\exists v'\,.\,(ie1(v', v) \vee ie2(v', v) \vee ie3(v', v))$	Heap cell v corresponds to an index element
$leftOf(v_1, v_2)$	$\exists w.\left(\bigvee_{(s_1,s_2)} s_1(w, v_1) \wedge s_2(w, v_2)\right)$ where $(s_1, s_2) \in \{(s_1, s_2) \in Sel \times Sel \mid s_1$ is left-adjacent to $s_2)\}$	v_1 is pointed to by a pointer that is *logically left-adjacent* to the pointer pointing to v_2

Predicate	**Defining Formula**	**Intended Meaning**
$leftOfStar(v_1, v_2)$	$leftOf^+(v_1, v_2)$	Non-reflexive transitive closure of $leftOf(u, v)$
$rightSubTrees(v_1, v_2)$	$\exists w.(leftOfStar(v_1, w) \wedge downStar(w, v_2))$	v_2 resides in a subtree rooted at a pointer logically right of v_1
$rightOf(v_1, v_2)$	$\exists w. \left(\bigvee_{(s_1,s_2)} s_1(w, v_1) \wedge s_2(w, v_2) \right)$ where $(s_1, s_2) \in \{(s_1, s_2) \in Sel \times Sel \mid s_1$ is right-adjacent to $s_2)\}$	v_1 is pointed to by a pointer that is *logically right-adjacent* to the pointer pointing to v_2
$rightOfStar(v_1, v_2)$	$leftOf^+(v_1, v_2)$	Non-reflexive transitive closure of $rightOf(u, v)$
$leftSubTree(v_1, v_2)$	$\exists w.(rightOfStar(v_1, w) \wedge downStar(w, v_2))$	v_2 resides in a subtree rooted at a pointer logically left of v_1
$treeness$	$\forall\ v_1, v_2, v_3\ leftOfStar(v_1, v_2) \Rightarrow \neg(downStar(v_1, v_3) \wedge downStar(v_2, v_3))$	If treeness holds a structure represents a tree
$inOrder[kge]$	$\forall v_1, v_2.(rightSubTree(v_1, v_2) \Rightarrow (kge(v_2, v_1) \wedge \neg kge(v_1, v_2))) \wedge \forall v_1, v_2.(leftSubTree(v_1, v_2) \Rightarrow (\neg kge(v_2, v_1) \wedge kge(v_1, v_2)))$	All index elements stored within the tree structures are properly ordered
$kge[x, left](v)\ \forall x \in Vars$	$\forall x \in Var\ \exists v_1. (x(v_1) \wedge kge(v, v_1) \wedge \neg kge(v_1, v))$	The key of v is greater than the key of the object pointed to by x
$kge[x, right](v)\ \forall x \in Vars$	$\forall x \in Var\ \exists v_1. (x(v_1) \wedge kge(v, v_1) \wedge \neg kge(v_1, v))$	The key of v is less than the key of the object pointed to by x
$p2\_5(v)$	$(\exists\ u\ left(v, u) \Leftrightarrow \exists\ u'\ cleft(v, u')) \wedge (\exists\ u\ cright(v, u) \Rightarrow \exists\ u'\ cleft(v, u')) \wedge (\exists\ u\ cright(v, u) \Rightarrow \exists\ u'\ ie2(v, u')) \wedge (\exists\ u\ right(v, u) \Rightarrow \exists\ u'\ cright(v, u')) \wedge (\exists\ u\ right(v, u) \Rightarrow \exists\ u'\ ie3(v, u')) \wedge (\exists\ u\ ie2(v, u) \Rightarrow \exists\ u'\ ie1(v, u')) \wedge (\exists\ u\ ie3(v, u) \Rightarrow \exists\ u'\ ie2(v, u')) \wedge (\exists\ u\ left(v, u) \Rightarrow \exists\ u'\ ie1(v, u')) \wedge (\exists\ u, v\ ie3(v, u) \wedge cright(v, w)) \Rightarrow (\exists\ u'\ right(v, u'))$	(P2) and (P5) hold for heap cell v

Predicate	**Defining Formula**	**Intended Meaning**
$isElement(v_1, v_2)$	$\exists v_{equal}.(downStar(v_2, v_{equal}) \wedge kge(v_{equal}, v_1) \wedge kge(v_1, v_{equal}) \wedge isStore(v_{equal})$	There exists an object with a key equal to the key of v_1 in the (sub-) tree rooted at v_2
$greRelation()$	$\forall\ u, v\ \neg kge(u, v) \Rightarrow kge(v, u)$	For any two numbers a, b holds that $\neg(a \geq b) \Rightarrow b \geq a$

Table 5.2: Instrumentation Predicates

Consistency Rules We use the set of consistency constraints[4] given in Figures 5.5 and 5.6.
All consistency constraints have to be implied by the core and instrumentation predicates. Otherwise we might discard structures that are valid according to our predicate definitions. We give formal proofs that all used consistency constraints are indeed implied by the predicate definitions in Appendix A.
The used predicates file in TVP format can be found in Appendix B on page 102. In this file, we partioned the set $Vars$ of program variables into two disjoint sets $PVarVisible$ and $PVarInvisible$. This was done because we generated the control-flow graph not from Java source code but from compiled Java byte code (see Section 5.1.1) and the Java compiler introduced many additional program variables. While these additional variables need to be tracked in the analysis, we would like to exclude them from being shown in TVLA's output graphs. Hence, we use two sets of variables one of which is visible in the output while the other is not printed.

Action Definitions

We describe the effects of program statements on our logical structures by the following predicate-update formulæ. We only give update formulæ for statements actually occuring in our implementation of the contains-method.

Assignments We need update formulæ for several kinds of assignment statements. We need to model the effects of assigning a reference/pointer to a reference/pointer (AssignRefToRef), assigning a field reference to a reference (AssignFieldRefToRef), and setting a reference to null (SetNull). Our analysis uses the predicate-update formulæ listed in Table 5.3.

Conditionals The control-flow graph generated from our Java implementation contains 8 different types of conditional statements. Table 5.4 lists those 8 statements. Instead of a predicate-update formula we give a precondition formula which is evaluated in order to check whether this action should be performed. The action is performed, i.e. the respective edge in the control-flow graph is traversed, if the formula is closed and evaluated to 1 or 1/2. In the case that the formula contains free variables the action is applied for each assignment to those free variables that potentially satisfies the formula. All action-definitions for conditionals involving key comparisons, i.e.

[4] We omit the $heapcell(v)$-predicate in the formulæ.

$$
\begin{array}{rcl}
\forall\ u, v\ (\mathit{isStore}(v) \land \mathit{storeProp}(v) & \triangleright & \nexists\ v'\ \mathit{down}(v, v')) \\
\forall\ u, v\ (\neg \mathit{downStar}(u, v) & \triangleright & \neg \mathit{down}(u, v)) \\
\forall\ u', v\ (\mathit{treeness}() \land (\exists\ u\ \mathit{down}(u, v) \land u \neq u') & \triangleright & \neg \mathit{down}(u', v)) \\
\forall\ \mathit{sel} \in \mathit{Sel}\ \forall\ u, v\ (\neg \mathit{down}(u, v) & \triangleright & \neg \mathit{sel}(u, v)) \\
\forall\ s_1 \in \mathit{Sel}\ \forall\ s_2 \in \mathit{Sel} \backslash \{s_1\}\ \forall\ u, v\ (\mathit{treeness}() \land s_1(u, v) & \triangleright & \neg s_2(u, v)) \\
\forall\ v, w\ (\exists\ u\ \mathit{treeness}() \land \mathit{rightSubTree}(u, v) \land \mathit{leftSubTree}(u, w) & \triangleright & \neg \mathit{downStar}(v, w)) \\
\forall\ v, w\ (\exists\ u\ \mathit{treeness}() \land \mathit{rightSubTree}(u, v) \land \mathit{leftSubTree}(u, w) & \triangleright & \neg \mathit{downStar}(w, v)) \\
\forall\ v, w\ ((\exists\ u\ (\mathit{leftSubTree}(u, v) \land \mathit{ie1}(w, u)) & \triangleright & \neg \mathit{cleft}(w, v)) \\
\forall\ v, w\ ((\exists\ u\ (\mathit{leftSubTree}(u, v) \land \mathit{ie1}(w, u)) & \triangleright & \neg \mathit{cright}(w, v)) \\
\forall\ v, w\ ((\exists\ u\ (\mathit{leftSubTree}(u, v) \land \mathit{ie1}(w, u)) & \triangleright & \neg \mathit{right}(w, v)) \\
\forall\ v, w\ ((\exists\ u\ (\mathit{leftSubTree}(u, v) \land \mathit{ie1}(w, u)) & \triangleright & \neg \mathit{ie2}(w, v)) \\
\forall\ v, w\ ((\exists\ u\ (\mathit{leftSubTree}(u, v) \land \mathit{ie1}(w, u)) & \triangleright & \neg \mathit{ie3}(w, v)) \\
\forall\ v, w\ ((\exists\ u\ (\mathit{leftSubTree}(u, v) \land \mathit{ie2}(w, u)) & \triangleright & \neg \mathit{cright}(w, v)) \\
\forall\ v, w\ ((\exists\ u\ (\mathit{leftSubTree}(u, v) \land \mathit{ie2}(w, u)) & \triangleright & \neg \mathit{right}(w, v)) \\
\forall\ v, w\ ((\exists\ u\ (\mathit{leftSubTree}(u, v) \land \mathit{ie2}(w, u)) & \triangleright & \neg \mathit{ie3}(w, v)) \\
\forall\ v, w\ ((\exists\ u\ (\mathit{leftSubTree}(u, v) \land \mathit{ie3}(w, u)) & \triangleright & \neg \mathit{right}(w, v)) \\
\forall\ v, w\ ((\exists\ u\ (\mathit{rightSubTree}(u, v) \land \mathit{ie1}(w, u)) & \triangleright & \neg \mathit{left}(w, v)) \\
\forall\ v, w\ ((\exists\ u\ (\mathit{rightSubTree}(u, v) \land \mathit{ie2}(w, u)) & \triangleright & \neg \mathit{left}(w, v)) \\
\forall\ v, w\ ((\exists\ u\ (\mathit{rightSubTree}(u, v) \land \mathit{ie2}(w, u)) & \triangleright & \neg \mathit{cleft}(w, v)) \\
\forall\ v, w\ ((\exists\ u\ (\mathit{rightSubTree}(u, v) \land \mathit{ie2}(w, u)) & \triangleright & \neg \mathit{ie1}(w, v)) \\
\forall\ v, w\ ((\exists\ u\ (\mathit{rightSubTree}(u, v) \land \mathit{ie3}(w, u)) & \triangleright & \neg \mathit{left}(w, v)) \\
\forall\ v, w\ ((\exists\ u\ (\mathit{rightSubTree}(u, v) \land \mathit{ie3}(w, u)) & \triangleright & \neg \mathit{cleft}(w, v)) \\
\forall\ v, w\ ((\exists\ u\ (\mathit{rightSubTree}(u, v) \land \mathit{ie3}(w, u)) & \triangleright & \neg \mathit{cright}(w, v)) \\
\forall\ v, w\ ((\exists\ u\ (\mathit{rightSubTree}(u, v) \land \mathit{ie3}(w, u)) & \triangleright & \neg \mathit{ie1}(w, v)) \\
\forall\ v, w\ ((\exists\ u\ (\mathit{rightSubTree}(u, v) \land \mathit{ie3}(w, u)) & \triangleright & \neg \mathit{ie2}(w, v))
\end{array}
$$

Figure 5.5: Consistency constraints.

$$
\begin{array}{rcl}
\forall\ v_1, v_2\ (p2\_5(v_1) \wedge \neg(\exists\ v\ left(v1, v)) & \triangleright & \neg cleft(v1, v2)) \\
\forall\ v_1, v_2\ (p2\_5(v_1) \wedge \neg(\exists\ v\ cleft(v1, v)) & \triangleright & \neg left(v1, v2)) \\
\forall\ v_1, v_2\ (p2\_5(v_1) \wedge \neg(\exists\ v\ ie1(v1, v)) & \triangleright & \neg ie2(v1, v2)) \\
\forall\ v_1, v_2\ (p2\_5(v_1) \wedge \neg(\exists\ v\ ie2(v1, v)) & \triangleright & \neg ie3(v1, v2)) \\
\forall\ v_1, v_2\ (p2\_5(v_1) \wedge \neg(\exists\ v\ ie1(v1, v)) & \triangleright & \neg left(v1, v2)) \\
\forall\ v_1, v_2\ (p2\_5(v_1) \wedge \neg(\exists\ v\ ie2(v1, v)) & \triangleright & \neg cright(v1, v2)) \\
\forall\ v_1, v_2\ (p2\_5(v_1) \wedge \neg(\exists\ v\ ie3(v1, v)) & \triangleright & \neg right(v1, v2)) \\
\forall\ v_1, v_2\ (p2\_5(v_1) \wedge \neg(\exists\ v\ cleft(v1, v)) & \triangleright & \neg cright(v1, v2)) \\
\forall\ v_1, v_2\ (p2\_5(v_1) \wedge \neg(\exists\ v\ cright(v1, v)) & \triangleright & \neg right(v1, v2)) \\
\forall\ v_1, v_2\ (p2\_5(v_1) \wedge \neg(\exists\ u\ right(v1, u)) \wedge (E(v) cright(v1, v)) & \triangleright & \neg ie3(v1, v2)) \\
\forall\ u, v\ (inOrder() \wedge rightSubTree(u, v) & \triangleright & kge(v, u)) \\
\forall\ u, v\ (inOrder() \wedge rightSubTree(u, v) & \triangleright & \neg kge(u, v)) \\
\forall\ u, v\ (inOrder() \wedge leftSubTree(u, v) & \triangleright & kge(u, v)) \\
\forall\ u, v\ (inOrder() \wedge leftSubTree(u, v) & \triangleright & \neg kge(v, u)) \\
\forall\ a, c\ (inOrder() \wedge \exists\ b\ (kge(a, b) \wedge kge(b, c) \wedge & & \\
a \neq b \wedge a \neq c \wedge b \neq c) & \triangleright & \neg kge(c, a)) \\
\forall\ u, v\ (greRelation() \wedge \neg kge(u, v) & \triangleright & kge(v, u)) \\
\forall\ u, w\ (\exists\ p\ (\neg rightSubTree(u, w) \wedge downStar(p, w) \wedge & & \\
down(p, u) \wedge \neg downStar(u, w) & & \\
\wedge w \neq u \wedge w \neq p \wedge u \neq p) & \triangleright & leftSubTree(u, w)) \\
\forall\ u, w\ (\exists\ p\ (\neg leftSubTree(u, w) \wedge downStar(p, w) \wedge & & \\
down(p, u) \wedge \neg downStar(u, w) & & \\
\wedge w \neq u \wedge w \neq p \wedge u \neq p) & \triangleright & rightSubTree(u, w))
\end{array}
$$

Figure 5.6: Consistency constraints.

Statement	**Corr. Java Statement**	**Update Formulæ**
AssignRefToRef	$x_1 = x_2;$	$x_1(v) = x_2(v)$ and $r[x_1](v) = r[x_2](v)$
AssignFieldRefToRef	$x_1 = x_2.x_3;$	$x_1(v) = \exists\ v_1\ x_2(v_1) \wedge x_3(v_1, v)$ and $r[x1](v) = \exists\ v_1, v_2\ x_2(v_1) \wedge x_3(v_1, v_2) \wedge downStar(v_2, v)$
SetNull	$x =$ null;	$x1(v) = 0$ and $r[x_1](v) = 0$

Table 5.3: Predicate-update formulæ for assignments.

actions corresponding to Java expressions of the form $x_1.key \,\square\, x_2.key$ ($\square \in \{==, >=, <=, !=\}$), contain a check (via TVLA's message mechanism) to ensure that neither x_1 nor x_2 points to null which would cause a null-pointer exception during program execution.

Statement	**Corr. Java Statement**	**Precondition Formula**
IsNullVar	$x ==$ null	$\neg(\exists\ v\ x(v))$
IsNotNullVar	$x\ !=$ null	$\exists\ v\ x(v)$
IsEqualRef	$x_1 == x_2$	$\forall\ v\ x_1(v) \Leftrightarrow x_2(v)$
IsNotEqualRef	$x_1\ != x_2$	$\neg(\forall\ v\ x_1(v) \Leftrightarrow x_2(v))$
GreaterEqualKey	$x_1.key >= x_2.key$	$\exists\, v_1, v_2\ x_1(v_1) \wedge x_2(v_2) \wedge kge(v_1, v_2)$
NotGreaterEqualKey	$x_1.key < x_2.key$	$\exists\ v_1, v_2\ x_1(v_1) \wedge x_2(v_2) \wedge \neg kge(v_1, v_2)$
EqualKey	$x_1.key == x_2.key$	$\exists\ v_1, v_2\ x_1(v_1) \wedge x_2(v_2) \wedge kge(v_1, v_2) \wedge kge(v_2, v_1)$
NotEqualKey	$x_1.key\ != x_2.key$	$\neg(\exists\ v_1, v_2\ x_1(v_1) \wedge x_2(v_2) \wedge kge(v_1, v_2) \wedge kge(v_2, v_1))$

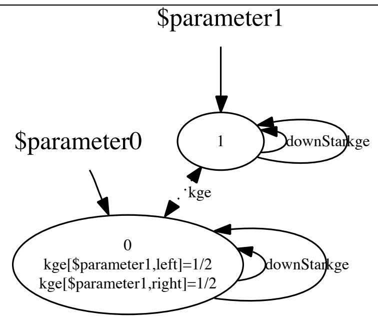
Table 5.4: Precondition formulæ for conditionals.

Input Structures

Our contains method takes two arguments: a reference to a 2-3-4 tree structure and a reference to an index element. The input structures for the analysis should describe all concrete logical structures such that

- the first argument is a reference to a valid 2-3-4 tree structure and
- the second argument is a reference to a valid index element.

Index elements are on our level of abstraction just single heap cells which makes the second argument trivial to model. To describe all possible 2-3-4 structures we need two abstract logical



```
%n = {r,ie}
%p = {
    heapcell = {r, ie}
    $parameter0     = {r}
    $parameter1     = {ie}

    downStar        = {r->r, ie->ie}
    r[$parameter0]  = {r}
    r[$parameter1]  = {ie}

    kge  =  { r->ie:1/2, ie->r:1/2,
                ie->ie, r->r:1/2}

    storeProp       = {r, ie}

    p2_5 = {r,ie}

    kge[$parameter1, left]  = {r:1/2, ie:1/2}
    kge[$parameter1, right] = {r:1/2, ie:1/2}

    greRelation = 1
    treeness    = 1
    inOrder     = 1
}
```

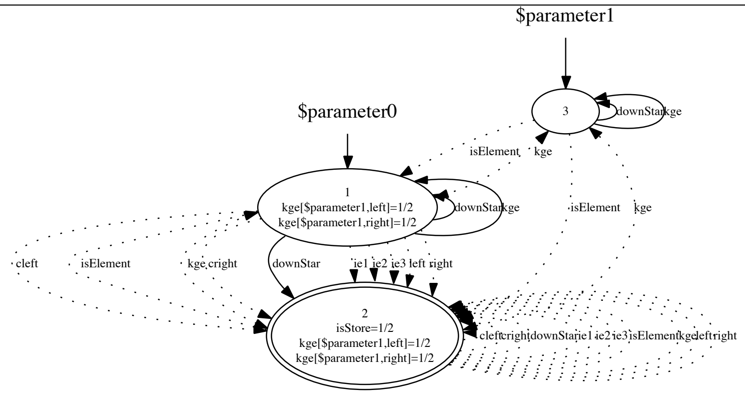
Figure 5.7: Input structure with empty tree used in the analysis of contains.

structures. The first represents empty trees which again are just single heap cells. The second represents all non-empty trees and consists of two nodes. One representing the root node and a summary node representing all other nodes and index elements linked in the tree. Figures 5.7 and 5.7 give a detailed specification for those two structures.

Analysis

We can use our predicates to formulate what property we want to show in our analysis, namely that at the exit node of the control-flow graph of contains the following formula is satisfied

$$\begin{aligned} F \;=\; & \mathit{treeness}() \wedge \mathit{inOrder}() \wedge (\forall v\ p2\_5(v)) \wedge \\ & ((\exists \mathit{ret}, k_1, k_2, r\ \$\mathit{parameter}0(r) \wedge \$\mathit{parameter}1(k_1) \wedge \mathit{isElement}(k_1, r) \wedge \mathit{return}(r) \\ & \quad \wedge \mathit{down}(\mathit{ret}, k_2) \wedge \mathit{kge}(k_2, k_1) \wedge \mathit{kge}(k_1, k_2)) \\ & \vee(\exists k_1, k_2, r\ \$\mathit{parameter}0(r) \wedge \$\mathit{parameter}1(k_1) \wedge \neg \mathit{isElement}(k_1, r) \\ & \quad \wedge \neg(\exists \mathit{ret}\ \mathit{return}(\mathit{ret})))) \end{aligned}$$


$parameter1
3
downStarkge
$parameter0
isElement
kge
1
kge[$parameter1,left]=1/2
kge[$parameter1,right]=1/2
downStarkge
isElement
kge
cleft
isElement
kge cright
downStar
ie1 ie2 ie3 left right
2
isStore=1/2
kge[$parameter1,left]=1/2
kge[$parameter1,right]=1/2
cleftcrightdownStarie1 ie2 ie3isElementkgeleftright

```
%n = {r, ts, ie}
%p = {

    heapcell        = {r, ts, ie}

    sm              = {ts:1/2}

    $parameter0     = {r}
    $parameter1     = {ie}

    down            = {r->ts:1/2, ts->ts:1/2}
    downStar        = {r->r, ie->ie, ts->ts:1/2, r->ts}

    r[$parameter0]  = {r, ts}
    r[$parameter1]  = {ie}

    ie1             = {r->ts:1/2, ts->ts:1/2}
    ie2             = {r->ts:1/2, ts->ts:1/2}
    ie3             = {r->ts:1/2, ts->ts:1/2}

    left             = {r->ts:1/2, ts->ts:1/2}
    cleft            = {r->ts:1/2, ts->ts:1/2}
    cright           = {r->ts:1/2, ts->ts:1/2}
    right            = {r->ts:1/2, ts->ts:1/2}

    kge              = { r->r:1/2, ie->ie, ts->ts:1/2,
                         r->ie:1/2, ie->r:1/2,
                         r->ts:1/2, ts->r:1/2,
                         ie->ts:1/2, ts->ie:1/2 }

    isStore          = {ts:1/2}
    storeProp        = {r, ts, ie}

    leftOf                     = {ts->ts:1/2, r->ts:1/2, ts->r:1/2, r->r:1/2}
    leftOfStar                 = { ts->ts:1/2, r->ts:1/2,
                                   ts->r:1/2, r->r:1/2, ie->ie:1/2}
    rightSubTree               = {ts->ts:1/2, r->ts:1/2, ts->r:1/2, r->r:1/2}

    rightOf                    = {ts->ts:1/2, r->ts:1/2, ts->r:1/2, r->r:1/2}
    rightOfStar                = { ts->ts:1/2, r->ts:1/2,
                                   ts->r:1/2, r->r:1/2, ie->ie:1/2}
    leftSubTree                = {ts->ts:1/2, r->ts:1/2, ts->r:1/2, r->r:1/2}

    p2_5                       = {r,ts,ie}

    kge[$parameter1, left]     = {r:1/2, ts:1/2, ie:1/2}
    kge[$parameter1, right]    = {r:1/2, ts:1/2, ie:1/2}

    isElement   = {ie->r:1/2, ie->ts:1/2, ts->r:1/2, ts->ts:1/2}

    greRelation = 1
    inOrder     = 1
    treeness    = 1
}
```

Figure 5.8: Input structure with non-empty trees used in the analysis of contains.

This formula captures exactly the properties we stated at the beginning of this chapter as the intented proof obtained from our shape analysis.
We append two conditional statements to the exit node of the control-flow graph of the program to collect valid output structures at one program point and invalid structures at another. This is done in order to simplify - and to rule out human errors within - the evaluation of the output structures. We consider an output structure valid if it satisfies a control formula C_f, invalid otherwise. We choose C_f to be F which encodes what we aimed at proving with our shape analysis. Hence, we add two conditional statements as given in Table 5.5 to our set of action definitions.

Statement	Precondition Formula
StructuresOK	$treeness() \wedge inOrder() \wedge (\forall v \; p2\_5(v)) \wedge (\exists r, k_1, k_2, root \; \$parameter0(root) \wedge \$parameter1(k_1) \wedge isElement(k_1, root) \wedge return(r) \wedge down(r, k_2) \wedge kge(k_2, k_1) \wedge kge(k_1, k_2)) \vee (\exists k_1, k_2, root \; \$parameter0(root) \wedge \$parameter1(k_1) \wedge \neg isElement(k_1, root) \wedge \neg(\exists r \; return(r)))$
StructuresNOK	$(treeness() \wedge inOrder() \wedge (\forall v \; p2\_5(v)) \wedge (\exists r, k_1, k_2, root \; \$parameter0(root) \wedge \$parameter1(k_1) \wedge isElement(k_1, root) \wedge return(r) \wedge down(r, k_2) \wedge kge(k_2, k_1) \wedge kge(k_1, k_2)) \vee (\exists k_1, k_2, root \; \$parameter0(root) \wedge \$parameter1(k_1) \wedge \neg isElement(k_1, root) \wedge \neg(\exists r \; return(r))))$

Table 5.5: Preconditions of additional conditional statements.

The key idea is that the analysis is able to find the invariant that all nodes potentially storing a key value equal to the one the method scans for are rooted at program variable *node*. Hence, we introduce two abstraction predicates for heap cells identified as storing only keys less than or greater than the one scanned for. If the analysis is able to find the invariant, all heap cells not directly pointed to by some program variable can be summarized in one of three nodes. They are either added to (1) a node summarizing all heap cells reachable from program variable *node* which - during the analysis - corresponds to cells reachable from program variable $r0$. (2) A node summarizing all cells identified as storing keys greater and (3) less, respectively, than the key scanned for. The same idea was successfully used with binary trees. However, in our case we have to remember that the *key greater or equal* predicate is not defined for heap cells representing node objects. Again, such objects store a set of keys and are not associated with a single key value. Therefore, the analysis has to consider a fourth summary node collecting cells corresponding to node objects visited during the analysis. In order to force the analysis not to summarize such cells with those reachable from $r0$ we used reachability from $r0$ as an abstraction predicate. This yields the desired effect. Every heap cell not directly pointed to by a program variable and visited during the analyzed algorithm is collected in exactly one of (at most) four summary nodes. The analysis can also establish that every cell not reachable from $r0$ cannot store a key equal to the one of the second argument of the method.

In total, the analysis computed 17 possible output structures. For all of which our control formula C_f was satisfied. Figure 5.9 shows some examples of those output structures.

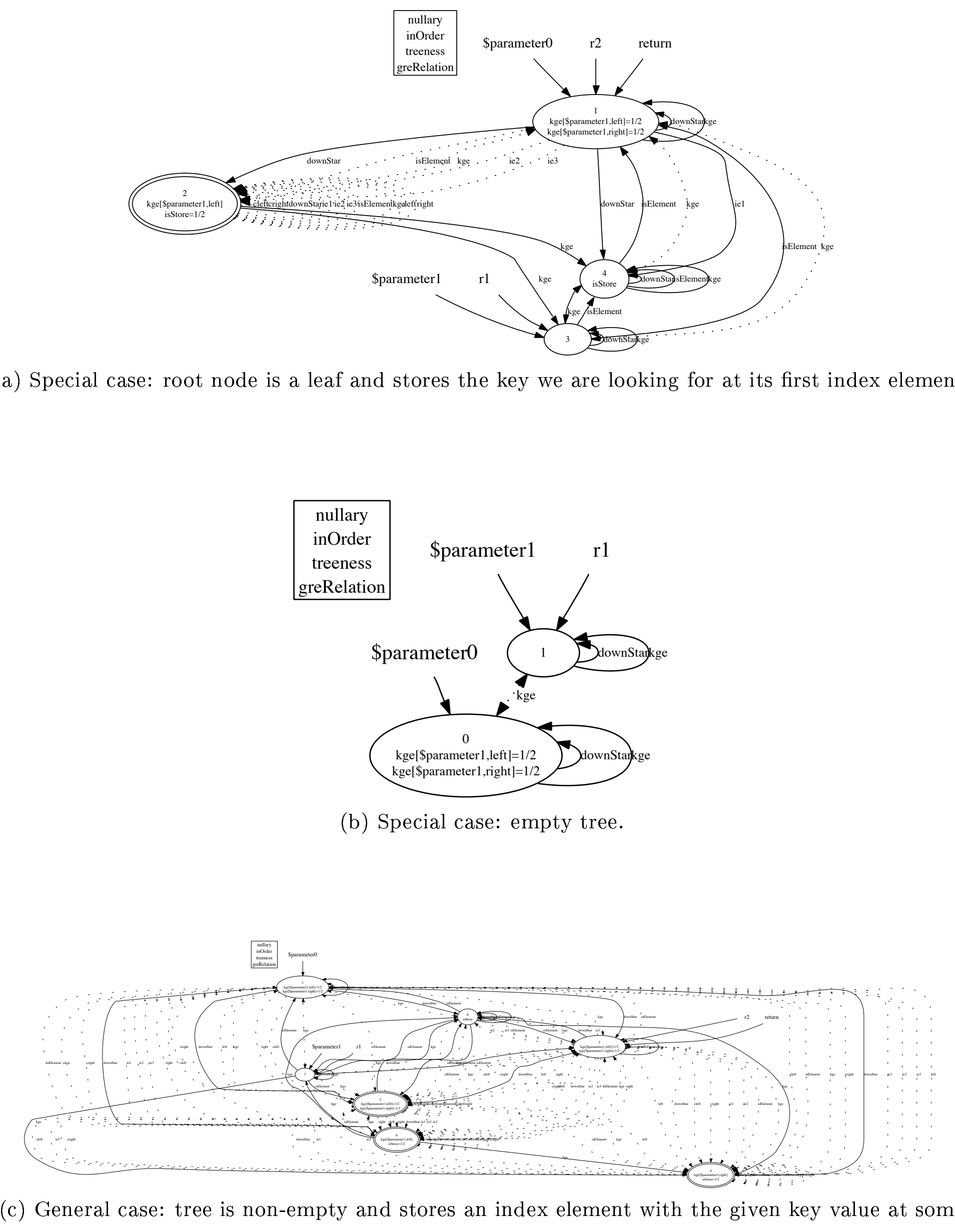
To conclude the analysis of the contains method, we like to note that although the analyzed implementation was a direct adaption of the algorithm presented in Chapter 4.3.1 a programer might choose a different approach. The presented algorithm works in a backward fashion, considering the keys stored at a node in a reversed order. A forward approach in which the algorithm considers the keys in ascending order seems more intuitive. Figure 5.10 shows such an arguable more intuitive implementation. We did a shape analysis on this method - which we called contains2 - utilizing exactly the same set of predicates, update rules, and input structures. The analysis produced the same set of output structures at the exit point of the program as in the analysis of the first implementation. This suggests that our predicates are sufficiently general to be able to prove partial correctness of various implementations of membership tests on 2-3-4 trees. This claim is made under the assumption that all such implementations manage one program variable pointing to the subtree in which an index element with a key equal to the one scanned for has to be located. However, this seems to be a fairly reasonable assumption.

5.2.2 Insert-Operation

We do our shape analysis on an insert algorithm implemented slightly different than introduced in Chapter 4.3.2. The way we presented the algorithm it starts with the method insert() which calls the function insertNonFull(). The latter recursively calls itself until a leaf node is reached at which the new key is eventually inserted. A closer look shows that this algorithm is tail recursive. It can therefore easily be turned into an interation. The algorithm then just traverses the tree from its root to the leaf in which the new key is to be inserted. On its way down the tree the split operation is applied whenever an already full node would be traversed. We therefore decided to implement the algorithm to work iterative rather than recursive.
Our iterative implementation had just two calls of the split() function so we decided to inline the function for the analysis. During the inlining we observed that the first call of the split-function has fixed arguments. We therefore just inlined the code of split() actually reachable with this arguments. Figure 5.11 shows the Java source code of the insert implementation we are going to analyze.

Excursus: Insert Implementations and Integer Arithmetic

The careful reader should have noticed that our implementation of insert uses pointer comparisons to determine the number of index elements stored at a node. References are also used to determine where to store the index element propagated to the parent node after a split operation. For 2-3-4 trees there is no reason not to use references for those tasks. However, real-life implementations often store an integer value indicating the number of stored index elements with a node object. Such implementations can access this value directly when they need to obtain the number of index elements stored at a node. Implementations of general B-trees (almost) always use integer values to indicate where keys are inserted after split operations. This is not surprising as it seems natural to store index elements and child pointers in arrays and use indices to address them.



(a) Special case: root node is a leaf and stores the key we are looking for at its first index element.

(b) Special case: empty tree.

(c) General case: tree is non-empty and stores an index element with the given key value at some inner node.

Figure 5.9: Sample output structures of the analysis of contains.

```
Node234 result = null;
if( node.ie1 == null )
  node = null;
  while( node != null ) {
    if( ie.key < node.ie1.key )
      node = node.left;
    else if ( ie.key == node.ie1.key ) {
      result = node;
      node   = null;
    }
    else if (
           node.ie2 == null
        || ie.key < node.ie2.key )
      node = node.cleft;
    else if (
               node.ie2 != null
         && ie.key == node.ie2.key ) {
      result = node;
      node   = null;
    }
    else if (
            node.ie3 == null
         || ie.key < node.ie3.key )
      node = node.cright;
    else if (
               node.ie3 != null
         && ie.key == node.ie3.key ) {
      result = node;
      node   = null;
    }
    else
      node = node.right;
  }
  return result;
```

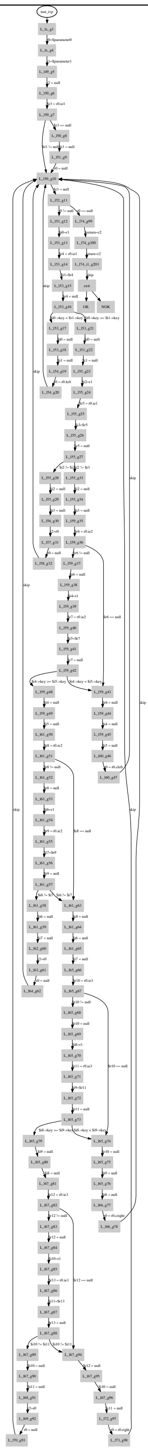


Figure 5.10: Another Java implementation of contains with its corresponding control-flow graph.

```
static public Node234 insert( Node234 root, IndexElement ie ) {
  if( root.ie3 != null ) {
    // save reference to old root
    Node234 child = root;
    // create new root with old root as leftmost child
    root = new Node234();
    Node234 z = new Node234();

    // collect
    IndexElement iel1, iel2, iel3;
    Node234 left1, left2, cleft1, cleft2;
    iel1 = child.ie1;
    iel2 = child.ie2;
    iel3 = child.ie3;
    left1 = child.left;
    cleft1 = child.cleft;
    left2 = child.cright;
    cleft2 = child.right;
    // spill
    child.ie3 = null;
    child.ie2 = null;
    child.ie1 = null;
    child.left = null;
    child.cleft = null;
    child.cright = null;
    child.right = null;
    root.ie1 = iel2;
    child.ie1 = iel1;
    z.ie1 = iel3;

    root.left = child;
    root.cleft = z;

    child.left = left1;
    child.cleft = cleft1;
    z.left = left2;
    z.cleft = cleft2;
  }
  Node234 node = root;
  Node234 child = null;
  while( node.left != null ) {
    // find correct child to insert the index element into
    if( node.ie3 != null && node.ie3.key < ie.key)
      child = node.right;
    else if( node.ie2 != null && node.ie2.key < ie.key )
      child = node.cright;
    else if( node.ie1 != null && node.ie1.key < ie.key )
      child= node.cleft;
    else
      child = node.left;
    // if child is already full, split it first
    if( child.ie3 != null ) {
      Node234 parent = node;
      Node234 z = new Node234();

      z.ie1 = child.ie3;
      if( child.left != null ) {
        z.left  = child.cright;
        z.cleft  = child.right;
        child.cright = null;
        child.right  = null;
      }
      // add new child and new index element to parent
      if( child == parent.left ) { // node at left was split
        parent.right = parent.cright;
        parent.cright = null;
        parent.cright = parent.cleft;
        parent.cleft = null;
        parent.cleft = z;
        parent.ie3 = parent.ie2;
        parent.ie2 = null;
        parent.ie2 = parent.ie1;
        parent.ie1 = null;
        parent.ie1 = child.ie2;
      }
       // node at cleft was split
      else if( child == parent.cleft ) {
        parent.right = parent.cright;
        parent.cright = null;
        parent.cright = z;
        parent.ie3 = parent.ie2;
        parent.ie2 = null;
        parent.ie2 = child.ie2;
      }
      // node at cright was split
      else if( child == parent.cleft ) {
        parent.right = z;
        parent.ie3 = child.ie2;
      }

      child.ie2 = null;
      child.ie3 = null;

      // refind correct child to insert into
      if( node.ie3 != null && node.ie3.key < ie.key)
        child = node.right;
      else if( node.ie2 != null && node.ie2.key < ie.key )
        child = node.cright;
      else if( node.ie1 != null && node.ie1.key < ie.key )
        child= node.cleft;
      else
        child = node.left;
    }
    //
    node = child;
  }
  // insert into non-full leaf
  IndexElement a,b,c;
  a = node.ie1; b = node.ie2; c = node.ie3;
  // collect
  if( node.ie2 != null ) {
    if( node.ie2.key < ie.key  ) {
      c = ie;
      ie = null;
    }
    else
      c = node.ie2;
  }
  if( node.ie1 != null && ie != null ) {
    if( node.ie1.key < ie.key  ) {
      b = ie;
      ie = null;
    }
    else
      b = node.ie1;
  }
  if( ie != null ) {
    a = ie;
  }
  // spill
  node.ie3 = null;
  node.ie2 = null;
  node.ie1 = null;
  node.ie1 = a;
  node.ie2 = b;
  node.ie3 = c;

  return root;
}
```

Figure 5.11: An iterative Java implementation of the insert-operation for 2-3-4 trees.

In this excursus, we describe how shape analysis can cope with integers used to determine the number of index elements stored at a node. We also give a shape analysis of a simple Java method which uses integer arithmetic as a proof-of-concept. The key idea is that we want the analysis to *know* about the values of integers.

Integer Arithmetic In order to be able to analyze insert implementations for 2-3-4 trees the analysis must be able to handle integer variables with values in $\{0, 1, 2, 3, 4\}$ as well as increment and decrement operations on such variables.
The idea to solve this problem is to build a logical structure as depicted in 5.12. We add such a structure to the heap of the programs we want to analyze. Having each number represented by a heap cell and modeled the successor function of those numbers by a binary predicate $succ(v_1, v_2)$ we can easily model update formulæ for increment and decrement operations.

More formally, we allow our analysis to include integer variables as follows. Let $values \subset \mathbb{N}$

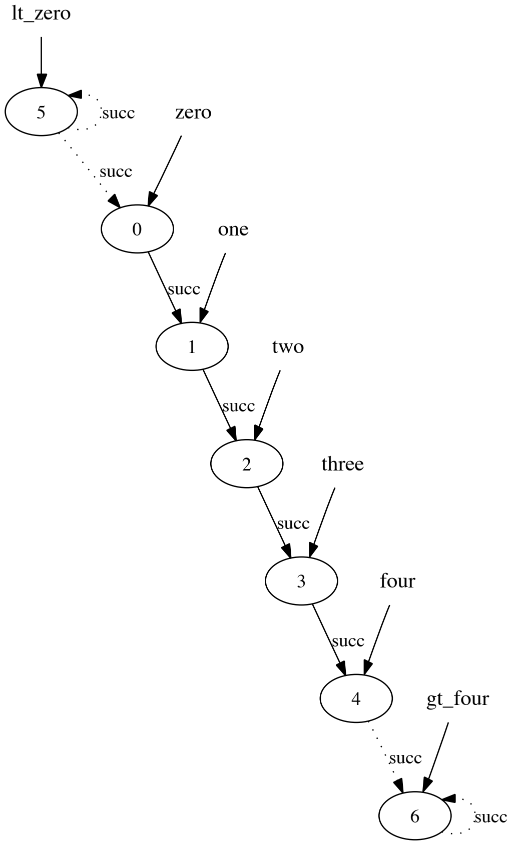


Figure 5.12: Logically representing a finite number of integer values.

be a finite intervall (on natural numbers) containing all values of integer variables that we are interested in. For each $number \in values$ we create a unary predicate $number(v)$ as well as a logical variable v_{number} such that $number(v) = 1 \Leftrightarrow v = v_{number}$. We also create two summary nodes representing integer values less than the smallest element of *values* and values greater than the greatest element of *values*, respectively. Hence, let $\bot \in values$ be the smallest element of our set of integer values and $\top \in values$ the greatest element. Then we add two predicates $lt\_\bot(v)$ and $gt\_\top(v)$. The binary predicate $succ(v_1, v_2)$ is used to model the successor function on numbers. To keep this *list of numbers* connected, we need a concept of reachability via the successor function. This is implemented by the predicate $r[succ](v_1, v_2)$ which is defined as

$$r[succ](v_1, v_2) = succ^+(v_1, v_2)$$

I.e. the non-reflexive transitive closure of $succ(v_1, v_2)$.
Update formulæ for the arithmetical operations are given in Table 5.6. We observe that no conditional statements need to be modeled. Equality checks between two integer variables is reduced to an already modeled check whether both variables point to the same heap cell. The same holds for equality checks between an integer variable and an integer constant c. This is because the latter is translated to the predicate name $pred_c$ if this number is modeled, or to $gt\_\top$ or $lt\_\bot$, respectively, otherwise. Unequality works analogous.

Statement	**Corr. Java Statement**	**Update Formula**
inc(x)	x = x + 1	$x(v) = \exists\ u\ x(u) \wedge succ(u, v)$.
dec(x)	x = x - 1	$x(v) = \exists\ u\ x(u) \wedge succ(v, u)$.

Table 5.6: Update formulæ for integer arithmetic.

Example We consider a class `Node234` and a method `poc( Node234 n )` as shown in Figure 5.13.

```
public class Node234 {
    public int iesStored;

    public Node234 left;
    public Node234 cleft;
    public Node234 cright;
    public Node234 right;

    public IndexElement ie1;
    public IndexElement ie2;
    public IndexElement ie3;
}
```

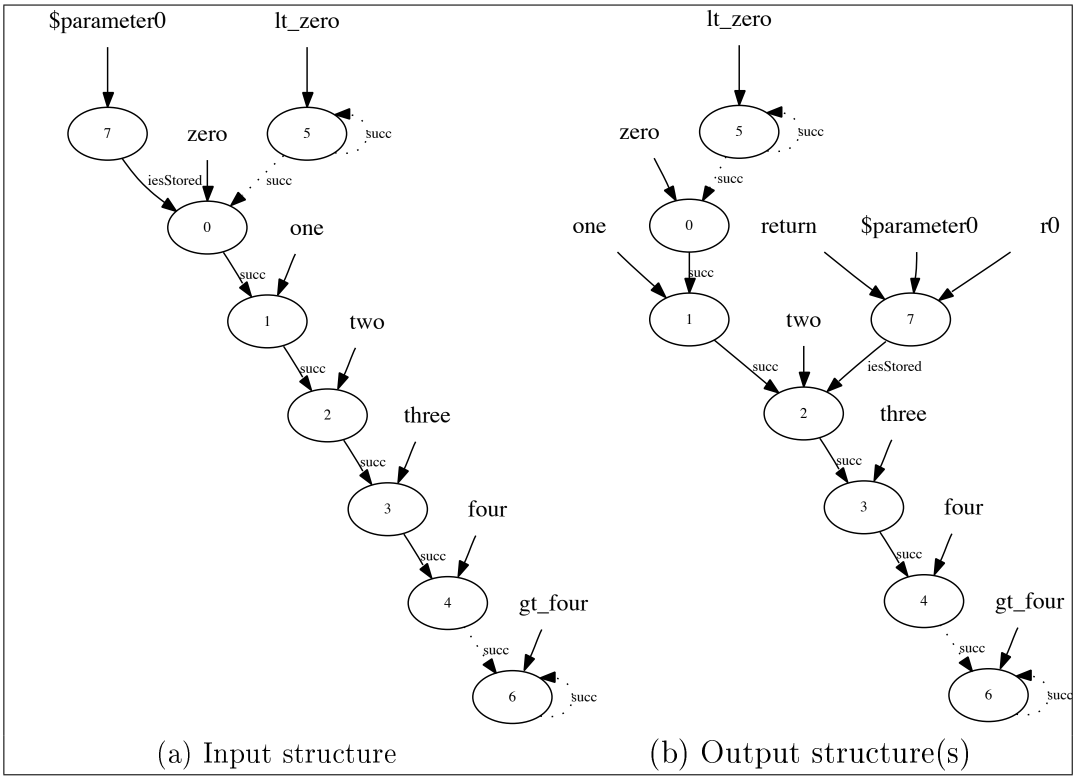
(a) Java class modeling nodes of 2-3-4 trees

```
public Node234 poc( Node234 n ) {
  n.iesStored = 1;
  n.iesStored++;
  if( n.iesStored == 2 )
    return n;
  return null;
}
```

(b) A method using integer arithmetic

Figure 5.13: A node class and a method using integers.

To show that our approach does work we modified our BTreeTranslator class to translate operations on integer variables into predicates of our integer domain described in the previous paragraph. We also let our BTreeTranslator class translate accesses to the integer attribute *iesStored* of the node class in the same way as accesses to a reference attribute of that name. We then merged our predicates and update formulæ from the analysis of the contains method with those developed in this excursus. We exemplarily did a shape analysis on this method using an empty Node234 object as input for the analysis. Figure 5.14 shows the results. As desired, the analysis was able to show that (a) the iesStored attribute of the node passed to the method is set to 2 and (3) a reference to the argument is returned at the end of the method invocation.



(a) Input structure (b) Output structure(s)

Figure 5.14: Input and outputstructures for the proof-of-concept example.

Analysis

The implementation of insert we want to analyze has some static code optimizations. We inlined the split()-function calls and even removed unreachable code from one of those inlined functions. We also set reference fields that are to be reassigned explicitly to null in the Java code. As in previous work, we formulate update formulæ for statements of the form $x.y = z$ under the assumption that $x.y$ equals null. Of course it is not necessary to explicitly set reference fields to null before assigning to. Normally, we would just change the translation of such an assignment to action macros from

```
%LBEGIN AssignRefToInstanceFieldRefStmt(%LHS_BASE, %LHS_FIELD, %RHS) %LEND
```

to

```
%LBEGIN AssignNullToInstanceFieldRefStmt(%LHS_BASE, %LHS_FIELD);
AssignRefToInstanceFieldRefStmt(%LHS_BASE, %LHS_FIELD, %RHS) %LEND
```

in the used J2TVLA properties file.

The shape analysis must be able to verify property (P1), i.e. that all leaf nodes are on the same level. We therefore introduce a new binary core predicate $level(u, v)$. The intended meaning of $level(u, v)$ is that the nodes the logical variables u and v correspond to are on the same level within the tree structure. Hence, $level(u, v)$ must be modeled reflexive, transitive, and symmetric. Although $level(u, v)$ is a core predicate, we think of it as *defined* be the following formula

$$level(u, v) \Leftrightarrow u = v \vee \exists\ a, b\ level(a, b) \wedge down(a, u) \wedge down(b, v)$$

We have to ensure that update formulæ capture effects of heap-manipulating statements in such a way that $level(u, v)$ always fits the above definitions.
We can now define an additional instrumentation predicate reflecting (P1) and enabling the analysis to verify that all leaf nodes reside on the same level within the tree.

$$p1() = \forall\ u, v\ level(u, v) \wedge \neg(\exists\ a\ left(u, a)) \Rightarrow \neg(\exists\ a\ left(v, a))$$

Intuitively, the consistency constraint

$$treeness() \wedge downStar(u, v) \wedge u \neq v \triangleright \neg level(u, v)$$

enables coerce to sharpen structures with respect to the *level*-predicate.

We want our shape analysis to prove that calling insert(tree, ie) where *tree* is a reference to a valid tree structure and *ie* a valid index element reference returns a valid tree structure which stores *ie*. In terms of our predicates, we might formulate the situation at the return-site via the following formula:

$$treeness() \wedge inOrder() \wedge p1() \wedge (\forall\ v\ p2\_5(v)) \wedge (\exists\ r, e\ tree(r) \wedge ie(e) \wedge isElement(e, r))$$

The implementation of the insert-operation contains assignment statements of the form $a.x = b;$, where a, b are references[5] and $x \in \{left, cleft, cright, right, ie1, ie2, ie3\}$. The implementation additionally introduces statements of the form $Node\ n = new\ Node();$. To handle such statements, we define the actions given in Table 5.7.

We introduced different actions for assignments to the various reference fields of Node-objects. This is done because we have to manually update the *leftOf*-, *leftOfStar*-, *leftSubTree*-, *rightOf*-, *rightOfStar*-, and *rightSubTree*-predicates. To do so, we need a different update formula for each field. By introducing one action per field, we can attach a specialized update formula with each such action. If the null-reference is assigned to a field, we do not need to distinguish the fields and can hence use a single, parameterized action for such assignments. As only Node-objects are created, it suffices to define one new-action.

[5] b may be a null-reference.

Statement	Corresponding Java Statement
new	$Nodex_1 = newNode()$
AssignRefToleftStmt	$x_1.left = x_2$
AssignRefTocleftStmt	$x_1.cleft = x_2$
AssignRefTocrightStmt	$x_1.cright = x_2$
AssignRefTorightStmt	$x_1.right = x_2$
AssignRefToie1Stmt	$x_1.ie1 = x_2$
AssignRefToie2Stmt	$x_1.ie2 = x_2$
AssignRefToie3Stmt	$x_1.ie3 = x_2$
AssignNullToInstanceFieldRefStmt	$x_1.x_2 = null, x_2 \in \{left, cleft, cright, right, ie1, ie2, ie3\}$

Table 5.7: Additional actions needed for the implementation of the insert-operation.

Table 5.7 does not give the update-formulæ of the actions due to their complexness. The new-action contains update-formulæ for all predicates, most of which are trivial. The assignment-actions update predicates capturing tree properties (treeness, right- and left subtrees, and so on) which mostly have complex, hard to read update formula. So we limit ourselves to presenting the most interessting formula only. The complete set of update formulæ can be found in Appendix B. Assigning a Node-object to a left-, cleft-, cright or right-field determines the level in the tree structure at which this Node-object is located. We define the following update formula to capture the effects of assignments of the form $x_1.x_2 = x_3$ on the *level*-predicate:

$$\begin{aligned}
&level(u, v) = \\
&(x_3(u) \lor x_3(v)) \quad ? \\
&\qquad (u = v \lor \exists\ a, b\ level(a, b) \land \\
&\qquad\ (x_1(a) \lor x_1(b)) \land (down(a, u) \lor down(a, v) \lor down(b, u) \lor down(b, v))) \\
&\quad : \quad level(u, v)
\end{aligned}$$

Unfortuately, we are not able to actually do this analysis - as described - because the static size of the insert program is too large for our computer hardware to handle.

Static Size of a Program By *Static Size of a Program P* we denote the *size* of the control-flow graph of P. Let $|\cdot|_{\mathcal{U}} : \mathcal{U} \mapsto \mathbb{N}$ be a function mapping an object of some universe $\mathcal{U}$ to its size, represented by a dimensionless positive integer value. Furthermore, let $G(P)$ be the control-flow graph of P, $E(G)$ the edge set of $G(P)$. The static size of P can then be computed as

$$|G(P)|_{\mathcal{U}} = \left(\sum_{e \in E(G)} |am(e)|_{\mathcal{U}} \right) + |V(G)|$$

where $am(e)$ denotes the action macro the edge e is labeled with and $|V(G)|$ the cardinal number of the vertex set of $G(P)$.

The size of an action is determined by the focus formulæ, precondition or update formulæ, and

coerce constraints of the action. Update formulæ not given by the user for the respective action are obtained from differencing and can become extremely large. Also, within all these formulæ instrumentation predicates might be expanded, that is they are (possibly recursively) substituted by their definitions.
Complex - or simply many - instrumentation predicates, complex action updates, and many program statements thus tend to lead to an exploding static size of the program.
To make things even worse, we must take the implementation of the shape analysis framework into account. TVLA, due to its nice object-oriented implementation style, represents each formula by an object. This includes representing also the atomic subformulæ these formulæ consist of - down to literals and variables - by objects. Therefore, at the start of the analysis of a program P a graph structure of $|G(P)|_{\mathcal{U}}$ objects has to be built up in memory.

However, switching to an interprocedural implementation of insert overcomes - in this case - problems with the static program size.

5.2.3 Interprocedural Shape Analyses

The analysis framework we use does already support interprocedural shape analyses [RSY05, RS01]. The analysis proposed by Rinetzky, Sagiv, and Yahav in [RSY05] computes procedure summaries as transformers from inputs to outputs. Parts of the heap not relevant to the procedure are ignored in this computation. This leads to an analysis modular in the heap which allows reusing the effect of a procedure at different call-sites and even between different contexts at the same call-site [RSY05].

An interprocedural implementation of the insert-operation - as given in Appendix B - would not cause any problems due to an overly large static size of the program. Empirical data also suggests that the cost of analyzing an interprocedural program is smaller than the cost of analyzing the same program with procedures inlined [RSY05]. Hence, switching to an interprocedural implementation might allow us to actually do a shape analysis on insert. However, we have to make sure that all procedure invocations are cutpoint-free as the interprocedural analysis can as yet only handle cutpoint-free programs. We call an object separating the local heap accessible by an instance of a procedure from the rest of the program heap a cutpoint. An invocation in which no such objects exist is called a cutpoint-free invocation.

Time constraints prevented us from following this approach further and actually doing a shape analysis of an interprocedural implementation of insert. The interprocedural analysis implemented in TVLA uses a new, although similar, file format: PTS. To generate PTS files from our Java byte code we have to implement a new Translator class, similar to the BTreeTranslator class we programmed to adjust J2TVLA's default translation behavior to our needs. We do not anticipate any problems with programming a new translator, however, we decided due to the time need to implement the interprocedural analysis outside the scope of the thesis.

We stated that an interprocedural implementation of insert (as given in Appendix B) does not generate functions with overly large static sizes. We base this claim on the results we obtained from doing shape analyses on our intraprocedural implementation with special input structures

(empty tree, tree with only 1 not yet full node and so on) for which we could identify and remove unreachable parts of the program in order to reduce the static size and make the program analyzable. By doing so, we could test some of the parts of the program that would constitute single functions in an interprocedural implementation. Besides reducing the number of program points we could try to eliminate superfluous predicates or deactivate differencing for update-formulæ by explicitly giving suitable update-formulæ in order to reduce the static size. However, we could not identify any predicate as superfluous[6] and experiments with update-formulæ did not result in any significant decrease in memory consumption.
The following pages present our results for specialized analyses.

Inserting Into Non-Full Leaf Node

We already observed that insert always inserts new elements into leaf nodes that store strictly less than the maximum number of elements. This part of our intraprocedural implementation can be easily tested by limiting input structures to 2-3-4 trees consisting of only one node storing at most 2 index elements. We marked the first program points not reachable for such input structures in the control-flow graph with $full$ and $noLeaf$, respectively. By verifying that no structures arise at those points, we can be sure that we have removed only unreachable parts of the control-flow graph.
At program exit, we would anticipate situations as depicted in Figure 5.15. The first case reflects an insertion into an empty node. The next two cases reflect the situation that the newly inserted index element has a key smaller than all elements already stored at the node. As the node was not already full, there could previously 1 or 2 elements been stored. Cases 4 and 5 show the results when an element with a key greater than the keys of already stored elements was inserted. Case 6 arises if a smaller and a greater element than the newly inserted one already existed.

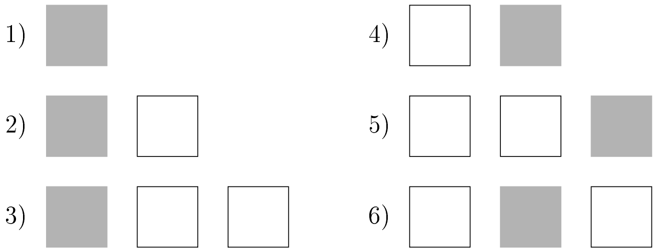
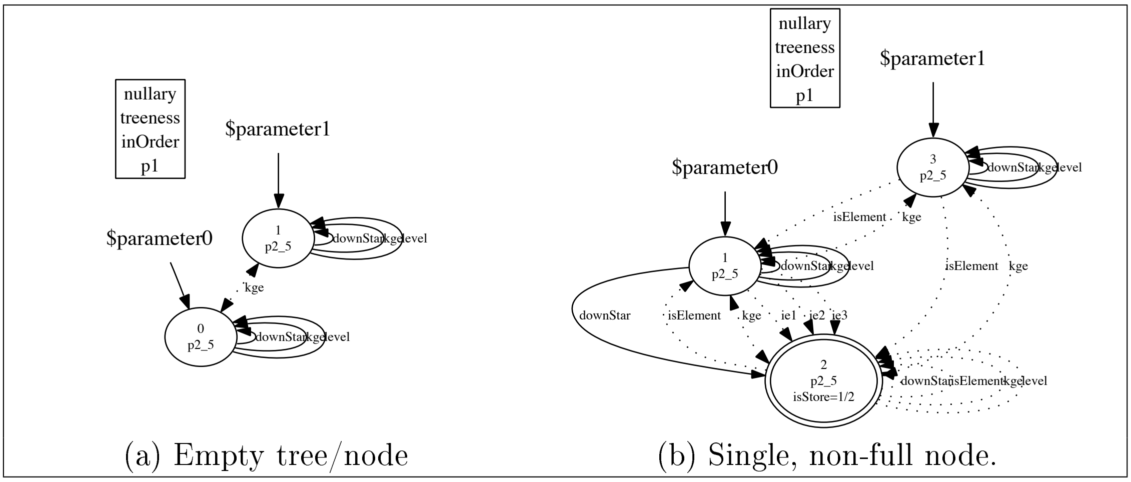


Figure 5.15: Possible situations arising after the insertion of an index element into a non-full leaf node. The grey boxes mark the locations of the newly inserted element.

Running a shape analysis with the input structures given in Figure 5.16 shows that no structures arise at program points $noLeaf$ and $full$ and generates the output structures presented in Figure 5.17. We observe that the abstract output structure (a) represents case 1, (b) embeds cases 4 and 5, (c) represents case 6, and cases 2 and 3 are represented by (d). We further observe that all

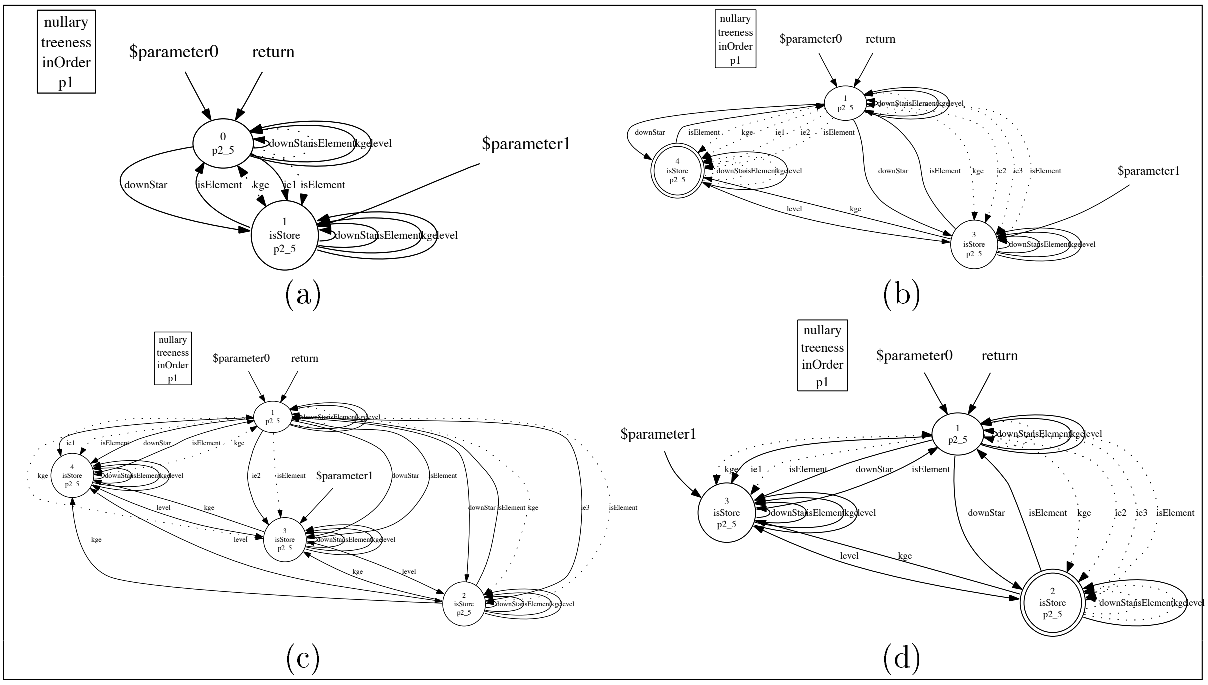
[6] One might consider the $leftSubTree$-predicate superfluous given the $rightSubTree$-predicate or vice-versa. However, defining one in terms of the other leads to even larger formulæ when these intrumentation predicates are expanded.

invariants (such treeness and properties 1 to 7) hold at the exit point. Please note that not all predicates are shown in the figures in order to increase readability. For example, $p2\_5(v)$ is true for all heap cells v, although it is not indicated in the graphical representations.



(a) Empty tree/node (b) Single, non-full node.

Figure 5.16: Input structures for analyzing insertion into non-full nodes.



(a) (b) (c) (d)

Figure 5.17: Output structures computed for the input structures from Figure 5.16.

Inserting Into a Possibly Full Node

If we allow our input structure to also represent trees with a full root node, we can include the split-operation into our analysis. Adjusting the input structures, restoring the now reachable parts of the control-flow, and running the analysis yields the anticipated results. At program point *exit* we observe the same 4 structures as previously given in Figure 5.17. At program point *noLeaf* a structure as depicted in Figure 5.18 arises which shows that the root node was correctly splitted.

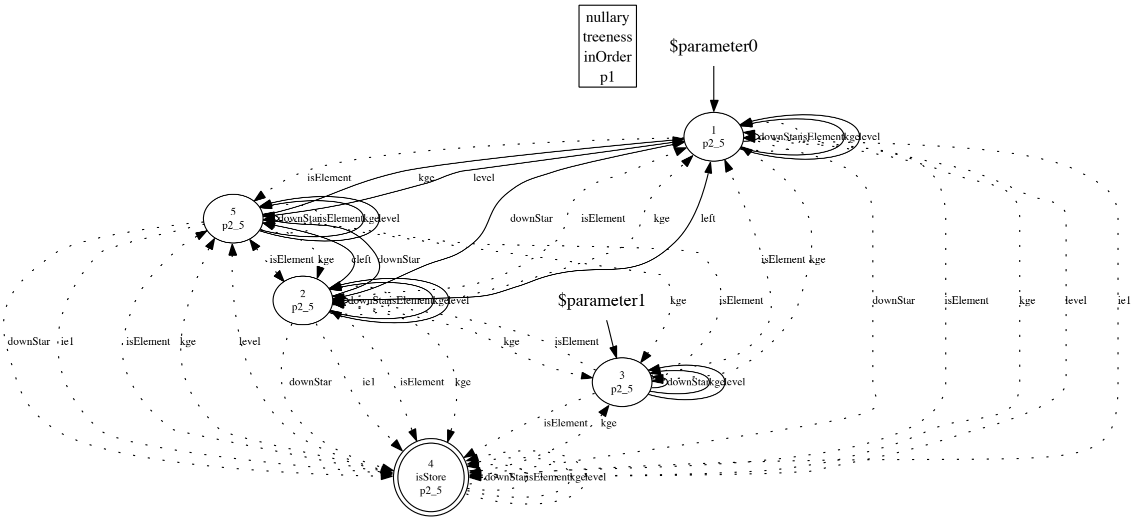


Figure 5.18: Structure representing the effects splitting a full root node.

5.2.4 Delete-Operation

The delete-operation has an even larger static size than the insert-operation. We therefore decided to refrain from any further attemps to analyse this operation and wait until an interprocedural analysis is available. As stated above, interprocedural analyses for our 2-3-4 tree implemenation will become available soon.

Our Java implementation of a delete-method for 2-3-4 trees - already implemented interprocedurally - can be found in Appendix B.

5.2.5 Empirical Results

Table 5.8 presents some empirical data obtained from our analyses. The comparison of the data for the two implementations of contains shows the significant influence of the number of structures on the analysis time. This may suggest that it is in general a sound strategy to *stay as abstract as long as possible* when it comes to designing analyses. Even if this *staying abstract* comes at the price of introducing additional predicates instead of generating more (and more conrete) structures. Our observations by designing the analyses presented in this work do reinforce this conjecture.

Analysis	**#locations in CFG**	**#unary predicates**	**#binary predicates**	**#structures**	**av #structs per location**	**max #structs per location**	**time sec (Mac, PC)**
contains (1)	108	76	18	5336	18	35	168.6, 79.3
contains (2)	102	76	18	4560	16	42	109.7, 48.2
poc (integer arithmetics)	17	98	21	15	0	1	0.18, 0.07
insert (partial)	106	166	19	278	2	6	4.13, 1.84

Table 5.8: Empirical Results

The worst-case complexity of the analysis, however, is in $\mathcal{O}(n^3)$, where n is the number of abstraction predicates.
All analyses were performed on two computers in order to show the influence of main memory and CPU power on the analysis time. We used an iBook G4 equipped with a 1.42 GHz PowerPC G4 processor and 1 GB RAM running under Mac OS X 10.4.11 and a PC equipped with a Pentium 4 2.6 GHz processor and 2 GB RAM running a Linux kernel in version 2.6.21.1.

5.3 Validation of (general) B-Tree Implementations

In this section, we will have a closer look on which problems arise with the analysis of general B-trees.

Figure 5.19 shows two classes representing nodes of general B-trees and index elements, respectively. While the index element class is identical to the one used with 2-3-4 trees, the node class has become more complex. Nodes now have attributes storing the number of index elements currently stored at the node (iesStored), the order of the tree to which the node belongs (t), and whether or not the node is a leaf. The latter is indicated by the boolean attribute isLeaf. Pointers to child nodes and the index elements the node stores are now organized in two arrays. With the latter two attributes being of an object type it makes sense to add a constructor to initialize the two arrays.
Incorporating constructors into shape analyses is a solved problem. However, it means that general B-trees are to be analyzed using an interprocedural shape analysis. In this case, the constructor is handled like a procedure that is called at the creation of a new node object. Interprocedural shape analyses were already described in Section 5.2.3.

The boolean attribute isLeaf of node objects is fairly simple to model. We just add an additional unary instrumentation predicate $isLeaf(v)$ which is true if v represents a node object which has

```
public class Node {
  public int t;
  public boolean isLeaf;

  public int iesStored = 0;

  public Node[] childs;
  public IndexElement[] elements;

  public Node( int t ) {
    this.t = t;
    childs = new Node[ 2*t ];
    elements = new IndexElement[2*t - 1];
}
```

(a) Node class

```
public class IndexElement {
    public int key;
    public Object content = null;
}
```

(b) Index element class

Figure 5.19: Java classes representing nodes and index elements, respectively, of general B-trees.

no child pointers, and false otherwise. The update formulæ for assignments to boolean variables just adjusts the value of the predicate accordingly. I.e. the effects on the heap of the statement

$$n.isLeaf = b, \; b \in \{\text{true}, \; \text{false}\}$$

can be captured by the following update formula

$$isLeaf(v) = (v = n)?b : isLeaf(v)$$

In summary, due to the structures/objects representing general B-trees only two new problems arise: handling arrays and the values they store as well as handling the order attribute t. As the constructor only initializes empty arrays we may postpone the first problem of considering contents of arrays until such content becomes crucial to the implementation and hence to the analysis.

The second problem might be postponed until the analysis of the insert-operation. At that point, the analysis has to consider integer arithmetics involving the order of the B-tree object. This value is bounded - which is a prerequisite for our handling of integer values - but not known a-priori. How to cope with a bounded but unknown value, especially when also dependencies to arrays exist, remains an open problem at this time.

5.3.1 Contains-Operation

As we already settled for an interprocedural analysis to cope with general B-trees, we choose to implement the contains operation as a recursive function. Figure 5.20 shows a recursive contains-method implemented in Java.

The implementation of the contains-operation on general B-trees is - measured by the lines of code - smaller than the respective operation on 2-3-4 trees. We may therefore conclude that the static size will not become a problem in the analysis. However, we now reached a point where the analysis needs to consider arrays. The following subsection describes how a shape analysis can cope with arrays in general. We also instantiate this general approach to our contains-method.

```
  public static Node contains( Node n, IndexElement ie ) {
    int i = 0;
    while( i < n.iesStored && ie.key > n.elements[i].key )
      i++;
    if( i < n.iesStored && ie.key == n.elements[i].key )
      return n;
    if( n.isLeaf )
      return null;
    else
      return contains( n.childs[i], ie );
}
```

Figure 5.20: Recursive Java implementation of the contains operation.

5.3.2 Handling Arrays

Reasoning about values stored in arrays is a problem that has already been addressed. In 1993, Masdupuy used numeric domains to relate values and index positions of elements stored in statically initialized arrays [Mas93]. In 2002, Blanchet et al. proposed two approaches to handle values stored in arrays: array expansion and array smashing [BCC+02]. In the same year, Flanagan and Quadeer used a predicate abstraction capable of handling arrays of unbounded sizes to infer loop invariants [FQ02]. Černý proposed a parametric predicate abstraction to reason about properties of array elements in 2003 [Č03].

In [GDD+04] a systematic approach to designing summarizing abstract numeric domains from existing numeric domains was described. Summarizing domains represent a potentially unbounded collection of numeric objects. In 2005, [GRS05] presented a static-analysis framework to cope with arrays by combining canonical abstraction (as described in Chapter 3) and summarizing numeric domains. This framework for numeric analyses of array operations was implemented into TVLA and successfully used to proof partial correctness of several smaller functions working on arrays, including an insertion-sort procedure.
We give a short introduction to the techniques proposed in [GRS05] and then investigate how these techniques can be applied to help in the analysis of B-trees.

Introduction to the Framework for Numeric Analysis of Array Operations

This section briefly summarizes the techniques presented in [GRS05], for a more elaborated introduction, we refer the reader to the original publication.

We denote the sets of scalar and array variables by

$$Scalar = \{v_1, \ldots, v_n\} \text{ and } Array = \{A_1, \ldots, A_n\},$$

respectively. A^S denotes the set of elements of array A in state S. Let further $\mathbb{V}$ denote the set of possible numeric values. Concrete states are encoded using the following functions:

- $Value^S : Scalar \mapsto \mathbb{V}$ which maps each scalar variable to its value,
- $Size^S : Array \mapsto \mathbb{N}$ which maps each array variable to its size, i.e. number of cells allocated for this array,

- $Value_A^S : A^S \mapsto \mathbb{V}$ which maps an element of array A to its value,
- and $Index_A^S : A^S \mapsto \mathbb{N}$ which maps an element of array A to its index position within the array.

Considering an array-manipulating language as given in Figure 5.21 we can define concrete collecting semantics by attaching a set of concrete states, D, to each program point. The set transformers given in Figure 5.22 are used to propagate the sets of concrete states through the program. The exact sets of concrete states is in general not computable. We therefore use the framework of abstract interpretation [CC77] to compute at each program point an overapproximation of the set of states that may arise at this point.

$$
\begin{array}{rl@{\qquad}rl}
\mathit{expr} ::= & c & \mathit{stmt} ::= & v \leftarrow \mathit{expr} \\
\mid & v & \mid & a[v] \leftarrow \mathit{expr} \\
\mid & a[v] & \mid & \mathbf{if}(\mathit{cond})\ \mathit{stmt}\ \mathbf{else}\ \mathit{stmt} \\
\mid & \mathit{expr} \odot \mathit{expr} & \mid & \mathbf{while}(\mathit{cond})\ \mathit{stmt} \\
\mathit{cond} ::= & \mathit{expr} \bowtie \mathit{expr} & \mid & \mathit{stmt};\ \mathit{stmt}
\end{array}
$$

$$
\begin{gathered}
c \in \mathbb{V},\quad v \in \mathit{Scalar},\quad a \in \mathit{Array} \\
\odot \in \{+, -, \times\},\quad \bowtie \in \{<, \leq, =, \geq, >\}
\end{gathered}
$$

Figure 5.21: Array-manipulating language used by Gopan, Reps, and Sagiv.

We move on to define a family of abstract domains whose elements are sets of abstract memory configurations. An abstract memory configuration $S^\sharp$ is a triple $\langle P^\sharp, \Omega^\sharp, \Delta^\sharp \rangle$, in which $P^\sharp$ specifies the array partitioning, $\Omega^\sharp$ the corresponding abstract numeric state, and $\Delta^\sharp$ stores the values of auxiliary predicates. The set of all possible abstract partitions is denoted by $\Sigma^\sharp$.

We use array partitioning for several purposes. First, we want to isolate array elements used in the currently considered statement to be able to perform strong updates when assigning to such elements. Secondly, we try to minimize the precision loss due to summarization by grouping elements with similar properties together. We achieve this by partition an array such that each element whose index is equal to the value of any scalar variable is placed in a group by itself and represented by a non-summary abstract array element. Consecutive array elements in between the indexed elements are grouped together and represented by summary abstract array elements. We formally define array partitions by using a fixed set of partitioning functions, Π. For an array A and a scalar v, a function $\pi_{A,v}$ in a concrete state S is of the form

$$\pi_{A,v} : A^S \mapsto \{-1, 0, 1\}$$

and is evaluated as

$$\pi_{A,v} = \begin{cases} -1 & \text{if } Index_A^S < Value^S(v) \\ 0 & \text{if } Index_A^S = Value^S(v) \\ 1 & \text{if } Index_A^S > Value^S(v) \end{cases}$$

Notation:
$c \in \mathbb{V},\ v \in Scalar,\ A \in Array,\quad S \in \Sigma,\ D \subseteq \Sigma$
$\odot \in \{+, -, \times\},\ \bowtie \in \{<, \leq, =, \geq, >\}$
$elem(S, A, v) = \big\{u \in A : Index^S_A(u) = Value^S(v)\big\}$

Expressions:
$[\![c]\!](S) = c,\quad [\![v]\!](S) = Value^S(v)$
$$[\![A[v]]\!](S) = \begin{cases} Value^S_A(u) & \text{if } \exists u \in elem(S, A, v) \\ \bot & \text{otherwise} \end{cases}$$
$[\![expr_1 \odot expr_2]\!](S) = [\![expr_1]\!](S) \odot [\![expr_2]\!](S)$

Conditions:
$[\![expr_1 \bowtie expr_2]\!](S) = [\![expr_1]\!](S) \bowtie [\![expr_2]\!](S)$

Assignments:
$[\![v \leftarrow expr]\!](S) = S\,[v \mapsto [\![expr]\!](S)]$
$$[\![a[v] \leftarrow expr]\!](S) = \begin{cases} S\,[u \mapsto [\![expr]\!](S)] & \text{if } \exists u \in elem(S, A, v) \\ \bot & \text{otherwise} \end{cases}$$

Errors:
$[\![.]\!](\bot) = \bot$

Set transformers:
$[\![v \leftarrow expr]\!](D) = \{[\![v \leftarrow expr]\!](S) : S \in D\}$ $(Assign_s)$
$[\![a[v] \leftarrow expr]\!](D) = \{[\![a[v] \leftarrow expr]\!](S) : S \in D\}$ $(Assign_a)$
$[\![cond]\!](D) = \{S : S \in D \text{ and } [\![cond]\!](S) = true\}$ $(Cond)$
$D_1 \sqcup D_2 = D_1 \cup D_2$ $(Join)$

Figure 5.22: Concrete collecting semantics for the array-manipulating language as given by Gopan, Reps, and Sagiv.

We call the set of partitioning functions parameterized with A Π_A. In a concrete state, we partition each array A by grouping together elements of A for which all partitioning functions in Π_A evaluate to the same value. Each group is represented by an abstract array element. Hence, $P^\sharp$ maps each array to a corresponding set of abstract array elements. These sets are always finite, although they might be combinatorially large.

The abstract numeric state, $\Omega^\sharp$, attaches to each partition an element of a summarizing numeric domain. Quantities of abstract objects are modeled by a dimension in the domain. In array analysis, non-summary dimensions represent the values of scalar variables, array sizes, and the values and index positions of non-summarized abstract array elements. Summary dimensions are used to model values and index positions of summary abstract array elements. Detailed information regarding the summarizing numeric domains we use here can be found in [GDD$^+$04].

The summarizing numeric domains allow us to reason about numeric properties of summarized array elements. The set of additional predicates Δ allows us to capture properties beyond the capabilities of summarizing numeric domains. In a concrete state S, a predicate in $\delta_A \in \Delta$ maps each element of array A to a truth value in $\{0,1\}$ indicating whether or not the predicate holds for this element. Formally, we define:

$$\delta_A : A^S \mapsto \{0,1\}$$

In abstract memory configurations, we simply use abstract predicates, denoted by $\delta^\sharp_A$, corresponding to the concrete predicates δ_A and map to a truth value in $\{0,1,1/2\}$:

$$\delta^\sharp_A : P^\sharp(A) \mapsto \{0,1,1/2\}$$

In abstract domains, $\Delta^\sharp$ stores the interpretation of auxiliary predicates and is defined as:

$$\Delta^\sharp\left(\delta_A, u\right) = \delta^\sharp_A(u)$$

We conclude this introduction with the definition of abstract states. Let

$$S^\sharp_1 = \langle P^\sharp_1, \Omega^\sharp_1, \Delta^\sharp_1\rangle \text{ and } S^\sharp_2 = \langle P^\sharp_2, \Omega^\sharp_2, \Delta^\sharp_2\rangle$$

denote two abstract memory locations. We can define a partial-order relation for abstract memory configurations as follows:

$$S^\sharp_1 \sqsubseteq S^\sharp_2 \Leftrightarrow P^\sharp_1 = P^\sharp_2 \wedge \Omega^\sharp_1 \sqsubseteq \Omega^\sharp_2 \wedge \Delta^\sharp_1 \sqsubseteq \Delta^\sharp_2$$

and a join-operation under the premise that $P^\sharp_1 = P^\sharp_2 = P^\sharp$ as:

$$S^\sharp_1 \sqcup S^\sharp_2 = \langle P^\sharp, \Omega^\sharp_1 \sqcup \Omega^\sharp_2, \Delta^\sharp_1 \sqcup \Delta^\sharp_2\rangle$$

An abstract state $D^\sharp$ is a set of abstract memory configurations with distinct array partitions. Given two abstract states $D^\sharp_1, D^\sharp_2 \in \Sigma^\sharp$, we define a partial-order relation on abstract states as:

$$D^\sharp_1 \sqsubseteq D^\sharp_2 \Leftrightarrow \forall S^\sharp_1 \in D^\sharp_1\ \exists S^\sharp_2 \in D^\sharp_2\ .\ S^\sharp_1 \sqsubseteq S^\sharp_2$$

The join-operator for abstract states computes the union of the corresponding sets of abstract memory configurations. Configurations with similar array partitions are joined together.

Applying the Framework to the Analysis of B-Trees

Unfortunately, the framework operates on a fixed, finite set of arrays (and scalar variables). Hence, it seems not directly applicable to our analysis in which B-trees of unboundedly many nodes are considered. Such trees imply also unboundedly many arrays - two at each node. However, an analysis modular in single procedures might only have to consider a fixed number of nodes per procedure and hence also a fixed number of arrays. We observe that the split-operation needs to consider three nodes, insert at most two, and insertNonFull - while considering its unboundedly many child nodes - has to consider only two arrays.

We might instantiate the framework to allow a shape analysis of the contains-operation (as depiected in Figure 5.20) as follows. We set

$$Array = \{elements, childs\},$$

$$Scalar = \{i, ie.key, n.elements.length, n.childs.length, n.iesStored\},$$

and $\Pi = \{\pi_{n.elements,i}, \pi_{n.childs,i}\}$. To increase precision, we employ auxiliary predicates

$$\Delta = \{sortedAndUnique(v), keyLessEqual(v)\}$$

with the obvious intended meaning. At the entry point of the contains-method, *sorted*(v) is true for all array elements and *keyLessEqual*(v) is set to 1/2. We now consider the while-loop. In each iteration, one array element is compared to *ie.key*. The iteration stops when *ie.key* is less than or equal to the array element. Due to our partitioning we have the following situation[7] after the while-loop.

Both arrays are partitioned into 3 abstract array elements, each. The analysis can determine that all elements in the leftmost partition are smaller than *ie.key* because *sortedAndUnqiue*(v) was true for all these elements and each element v was compared to *ie.key* such that $ie.key > v$ was satisfied. The middle partition contains a non-summary abstract array element representing the array value which failed the loop-condition, i.e. which is less or equal to *ie.key*. Due to the sortedness of the array, every element in the last, rightmost partition has to be strictly greater than *ie.key*. The analysis might set the *keyLessEqual*-predicate to false for this partition. Figure 5.23 shows the situation graphically.

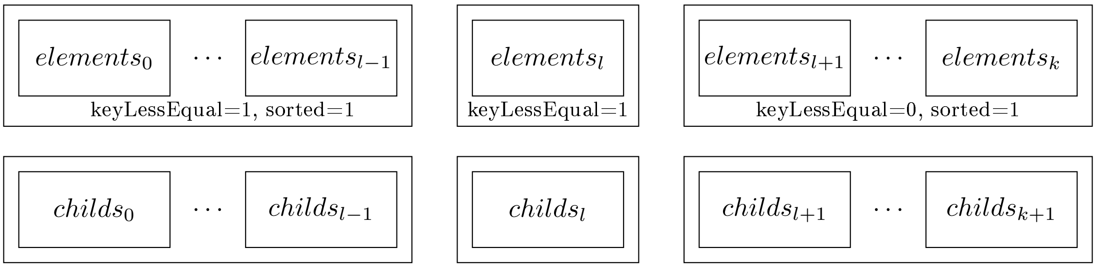


Figure 5.23: Partitioning of arrays during the analysis of contains.

The next two if-statements present no new challenges for an analysis. To handle the else statement we need auxiliary predicates representing the properties (P1) to (P7) of B-trees. If these predicates are true we can conclude that the reference returned by the else statement points to the only child element that can contain an index value with a key equal to *ie.key*.

5.3.3 Insert- and Delete-Operation

Insert and delete seem to add no new challenges. With 2-3-4 trees insert introduced some complexity issues but this is not the case this time. By partitioning the insert and delete operation into several procedures we can keep the programs to analyze - and thus their static sizes - small. Figure 5.24 shows the insert-operation of general B-trees implemented using several functions.

[7] There are also special cases when the loop stops after 0, 1, $n.iesStored - 1$ or *n.iesStored* iterations. However, those special cases are handled analogously to the general case discussed here.

```
public static Node insert( Node tree,
                           IndexElement ie ) {
  if( tree.iesStored == T * 2 - 1 ) {
    Node n = tree;
    Node s = new Node( T );
    tree = s;
    s.isLeaf = false;
    s.iesStored = 0;
    s.childs[0] = n;
    split( s, 1, n );
    insertNonFull( s, ie );
    return tree;
  }
  else {
    insertNonFull( tree, ie );
    return tree;
  }
}
```

(a) Main insert

```
public static void split( Node node, int i,
                          Node ithChild ) {
  Node z = new Node( T );
  z.isLeaf = ithChild.isLeaf;
  z.iesStored = T - 1;
  int j;
  for( j = 0; j < T - 1; j++ )
    z.elements[j] = ithChild.elements[j+T];
  if( !ithChild.isLeaf ) {
    for( j = 0; j < T; j++ )
      z.childs[j] = ithChild.childs[j+T];
  }
  ithChild.iesStored = T - 1;
  for( j = node.iesStored; j >= i; j-- )
    node.childs[j+1] = node.childs[j];
  node.childs[i] = z;
  for( j = node.iesStored - 1;
         j >= i - 1; j-- )
    node.elements[j+1] = node.elements[j];
  node.elements[i-1] =
      ithChild.elements[T-1];
  node.iesStored++;
  for( j = T - 1; j < 2 * T - 1; j++ )
    ithChild.elements[j] = null;
}
```

(b) Splitting full nodes

```
public static void insertNonFull( Node n, IndexElement ie ) {
  int i = n.iesStored - 1;
  if( n.isLeaf ) {
    while( i >= 0 && ie.key < n.elements[i].key ) {
      n.elements[ i + 1 ] = n.elements[ i ];
      i--;
    }
    n.elements[i+1] = ie;
    n.iesStored++;
  }
  else {
    while( i >= 0 && ie.key < n.elements[i].key )
      i--;
    i++;
    if( n.childs[i].iesStored == 2*T-1 ) {
      split( n, i+1, n.childs[i] );
      if( ie.key > n.elements[i].key )
        i++;
    }
    insertNonFull( n.childs[i], ie );
  }
}
```

(c) Insert into non-full node

Figure 5.24: Java implementation of the insert-operation on general B-trees.

The usage of arrays and problems resulting thereof is already addressed in 5.3.2. To represent integer values and model integer arithmetics we rely on the numeric domain introduced with the handling of arrays. However, we are not sure whether additional auxiliary predicates and/or consistency constraints are needed for an analysis. Our experience strongly suggests that such predicates and constraints are needed.

Considering all suboperations (insertNonFull, split, insert) separately, none of them should introduce new challenges. Although, updating our predicates manually may need some additional considerations. The main operations on arrays that are performed are insertions into already sorted arrays. Similar array operations were already analyzed during the shape analysis of an insert-sort procedure [GRS05].

From the theoretical point of view, we are looking forward to practically do such a shape analysis. However, it is totally unclear whether such a shape analysis can be done on current hardware in a reasonable time or if we have to consider a more modular approach to keep the number of structures during the analysis small enough to handle.

6 Conclusion

6.1 Contributions

This work presents instantiations of a parametric framework for shape analysis capable of proving partial correctness of several Java methods implementing operations on 2-3-4 tree. It was also theoretically established how these analyses might be generalized or adapted to work with general B-trees. We beliefe that based on the analyses and approaches presented in this thesis it will not take long until an actual B-tree implementation can be analyzed and its correctness partially proven be a shape analysis.
B-trees are one of the most popular data structures for data accesses due to their scalability and efficiency. Due to the particular properties of flash memory storage management algorithms and data structures that were designed for magnetic discs are not always appropriate for flash discs. See for example [GT05] for an introduction to the problems with flash memory storage systems and algorithms and data structures coping with those. Although B-trees are used in nearly all file systems, Wu et al. showed that B-tree implementations do not have to be modified to cope with flash memory by introducing a second layer over the flash translation layer (FTL) which provides a B-tree index management over flash memory storage systems [WCK03a, WKC07]. A similar layer has been developed for R trees [WCK03b]. Therefore, we do not only feel that this data structure will keep its popularity but also have high hopes that B-tree implementations and algorithms will not change because of flash memory. Hence, the analyses and approaches presented in this work stay valid.

6.2 Future Work

The immediate next step is quite obvious: overcoming the remaining obstacles and actually do a shape analysis of general B-trees and of the remaining operations on 2-3-4 trees. However, we are confident that this step can be taken soon and our theoretical considerations on how to adapt the presented shape analysis of 2-3-4 trees to general B-trees will be put to practice in the near future.

The current framework for shape analysis along with its newer modifications (e.g. the ability to analyze interprocedural cutpoint-free programs) should be efficient and precise enough to prove partial correctness of numerous interesting data structures and algorithms. Future work might address such structures and algorithms.

It also seems that TVLA might need further improvements in order to be able to handle more than just small methods and programs. The static size of a program tends to become the limiting factor very rapidly. One of the reasons for a large static program size is the fact that each action macro is replaced by an unparameterized formula. Alternatively, one could replace action macros by references to parametric formulæ. This way, each action type would generate one formula-object. Currently, each action instance generates one formula-object. Hence, at the moment, the analysis itself is faster but the control-flow graph might be significantly larger. The alternative implementation would trade a smaller control-flow graph for a slower analysis phase. However, this problem is already addressed in current work in the course of a Master's thesis at Tel Aviv University.

The subsequent sections list the work already planned or worked on following this thesis.

6.2.1 Interprocedural Analyses of insert and delete (on 2-3-4 Trees)

In Section 5.2.3 we sketched how an interprocedural analysis might overcome the identified remaining problems with analyzing insert- and delete-operations on 2-3-4 trees. The main reason we did not already implement an interprocedural analysis was that we did not want to protract this thesis by implementing a whole new Translator class able of coping with B-trees to use with J2TVLA (or rather J2PTS). However, work on such an interprocedural analysis has already started and we are optimistic to finish a complete shape analysis on 2-3-4 trees soon.

6.2.2 Generality of Predicates

We said in Chapter 5.1 that we intend to generalize the predicates used in the shape analysis of binary trees described in [Rei05]. By doing so, we allow for a shape analysis of binary trees with our set of predicates as well. This generality of our predicates, i.e. their ability to successfully show partial correctness of binary and 2-3-4 trees implementations, still has to be proven.
The remaining obstacle here is the delete-operation on 2-3-4 trees which is still not successfully analyzed. We cannot exclude the possibility that we need additional instrumentation predicates and/or consistency constraints. However, the instrumentation predicates used in the analysis of the contains and insert methods of 2-3-4 trees are already sufficient to redo the shape analysis on binary trees. We base the latter claim on the fact that all predicates used in anylses of binary trees have a generalized counterpart among the predicates used with 2-3-4 trees.
However, as soon as the delete method of 2-3-4 trees is successfully analyzed, we strongly plan to show the generality of our predicates by using them to perform a shape analysis on both kinds of trees.
It might also be worth investigating whether it is possible to analyze also other tree structures with this generalized set of predicates. Or whether it is possible to generalize the predicates even further to cope with a larger set of different kinds of tree structures.

List of Figures

List of Tables

Bibliography

[BCC$^+$02] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. *The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, chapter Design and Implementation of a Special-Purpose Static Program Analyzer for Safety-Critical Real-Time Embedded Software, pages 85–108. Lecture Notes in Computer Science 2566. Springer-Verlag, 2002.

[BCSZ03] Bouchra Bouqata, Christopher D. Carothers, Boleslaw K. Szymanski, and Mohammed J. Zaki. Understanding Filesystem Performance for Data Mining Applications, March 23, 2003.

[BFH02] Ray Bryant, Ruth Forester, and John Hawkes. Filesystem Performance and Scalability in Linux 2.4.17. In *Proceedings of the FREENIX Track: 2002 USENIX Annual Technical Conference (FREENIX-02)*, pages 259–274, Berkeley, CA, June 10–15, 2002. USENIX Association.

[BHRV06a] Ahmed Bouajjani, Peter Habermehl, Adam Rogalewicz, and Tomás Vojnar. Abstract Regular Tree Model Checking. *Electr. Notes Theor. Comput. Sci*, 149(1):37–48, 2006.

[BHRV06b] Ahmed Bouajjani, Peter Habermehl, Adam Rogalewicz, and Tomás Vojnar. Abstract Regular Tree Model Checking of Complex Dynamic Data Structures. In Kwangkeun Yi, editor, *Static Analysis, 13th International Symposium, SAS 2006, Seoul, Korea, August 29-31, 2006, Proceedings*, volume 4134 of *Lecture Notes in Computer Science*, pages 52–70. Springer, 2006.

[BLARS07] Igor Bogudlov, Tal Lev-Ami, Thomas Reps, and Mooly Sagiv. Revamping TVLA: Making Parametric Shape Analysis Competitive, 2007.

[BM72] Bayer, R. and McCreight. Organization and Maintenance of Large Ordered Indexes. *Acta Informatica*, 1:173–189, 1972.

[BPZ05] Ittai Balaban, Amir Pnueli, and Lenore D. Zuck. Shape Analysis by Predicate Abstraction. In Radhia Cousot, editor, *Verification, Model Checking, and Abstract Interpretation, 6th International Conference, VMCAI 2005, Paris, France, January 17-19, 2005, Proceedings*, volume 3385 of *Lecture Notes in Computer Science*, pages 164–180. Springer, 2005.

[Bru90] Bruffey et al. Hierarchical File System to Provide Cataloging and Retrieval of Data, 1990. United States Patent US 4,945,475.

[CC77] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In

Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[CLGS04] Adina Crainiceanu, Prakash Linga, Johannes Gehrke, and Jayavel Shanmugasundaram. Querying Peer-to-Peer Networks Using P-Trees. Technical report, Cornell University, February 05, 2004.

[CLRS01] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* MIT Press, Cambridge, MA, 2001. Second Edition.

[Com79] Douglas Comer. The Ubiquitous B-Tree. *ACM Computing Surveys*, 11(2):121–137, June 1979.

[CR06a] Sigmund Cherem and Radu Rugina. Compile-Time Deallocation of Individual Objects. In Erez Petrank and J. Eliot B. Moss, editors, *Proceedings of the 5th International Symposium on Memory Management, ISMM 2006, Ottawa, Ontario, Canada, June 10-11, 2006*, pages 138–149. ACM, 2006.

[CR06b] Sigmund Cherem and Radu Rugina. Maintaining Structural Invariants in Shape Analysis with Local Reasoning. Technical report, Cornell University, September 29, 2006.

[Dix01] Paul Dixon. Basics of Oracle Text Retrieval. *IEEE Data Eng. Bull*, 24(4):11–14, 2001.

[DOY06] D. Distefano, P. O'Hearn, and H. Yang. A Local Shape Analysis Based on Separation Logic, 2006.

[EMS00] Jacob Elgaard, Anders Møller, and Michael I. Schwartzbach. Compile-Time Debugging of C Programs Working on Trees. In *Proc. Programming Languages and Systems, 9th European Symposium on Programming, ESOP '00*, volume 1782 of *LNCS*, pages 182–194. Springer-Verlag, March/April 2000.

[FBB$^+$99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code.* Addison Wesley, 1999.

[FQ02] Cormac Flanagan and Shaz Qadeer. Predicate Abstraction for Software Verification. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 191–202, New York, NY, USA, 2002. ACM.

[GBC06] Alexey Gotsman, Josh Berdine, and Byron Cook. Interprocedural Shape Analysis with Separated Heap Abstractions, 2006.

[GDD$^+$04] Denis Gopan, Frank Dimaio, Nurit Dor, Thomas Reps, and Mooly Sagiv. Numeric Domains with Summarized Dimensions, January 09, 2004.

[GGL03] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *SOSP*, pages 29–43, 2003.

[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*, pages 1–15, 1996.

[GH99] Rakesh Ghiya and Laurie J. Hendren. Is it a Tree, a DAG, or a Cyclic Graph?, February 08, 1999.

[GRS05] Denis Gopan, Thomas Reps, and Mooly Sagiv. A Framework for Numeric Analysis of Array Operations. *SPNOTICES: ACM SIGPLAN Notices*, 40, 2005.

[GT05] Eran Gal and Sivan Toledo. Algorithms and Data Structures for Flash Memories. *CSURV: Computing Surveys*, 37, 2005.

[Hag87] Robert B. Hagmann. Reimplementing the Cedar File System Using Logging and Group Commit. In *SOSP*, pages 155–162, 1987.

[HD05] Andreas Harth and Stefan Decker. Optimized Index Structures for Querying RDF from the Web. In *LA-WEB*, pages 71–80. IEEE Computer Society, 2005.

[HHN94] Joseph Hummel, Laurie J. Hendren, and Alexandru Nicolau. A General Data Dependence Test for Dynamic, Pointer-Based Data Structures. In *PLDI*, pages 218–229, 1994.

[HKO06] B. Hudzia, M-T. Kechadi, and A. Ottewill. TreeP: A Tree-Based P2P Network Architectur, August 29, 2006.

[HR05] Brian Hackett and Radu Rugina. Region-Based Shape Analysis with Tracked Locations. *ACM SIGPLAN Notices*, 40(1):310–323, January 2005.

[IBM] IBM Corporation. JFS Website. http://www.ibm.com/developerworks/linux/library/l-jfs.html.

[IMM+04] Bala Iyer, Sharad Mehrotra, Einar Mykletun, Gene Tsudik, and Yonghua Wu. A Framework for Efficient Storage Security in RDBMS. In *Advances in Database Technology — EDBT 2004: 9th International Conference on Extending Database Technology*, volume 2992 of *Lecture Notes in Computer Science*, pages 147–164, Heraklion, Crete, Greece, March 2004. Springer-Verlag, Berlin Germany.

[Isa03] Florin Isaila. *An Overview of File System Architectures*, chapter 13, pages 273–289. Lecture Notes in Computer Science. Springer-Verlag, Dagstuhl, Germany, March 2003.

[Jen05] Martin Mosegaard Jensen. Understanding Parametric Shape Analysis. Master's thesis, University of Aarhus, Denmark, 2005.

[JJKS97] Jacob L. Jensen, Michael E. Joergensen, Nils Klarlund, and Michael I. Schwartzbach. Automatic Verification of Pointer Programs Using Monadic Second-Order Logic. In *PLDI '97*, 1997.

[Kim02] Won Kim. On Three Major Holes in Data Warehousing Today. *Journal of Object Technology*, 1(4):39–47, 2002.

[KLR02] Viktor Kuncak, Patrick Lam, and Martin Rinard. Role Analysis, 2002.

[LA00] Tal Lev-Ami. TVLA: A Framework for Kleene Logic Based Static Analyses. Master's thesis, Tel Aviv University, 2000.

[LAMS04] Tal Lev-Ami, Roman Manevich, and Shmuel Sagiv. TVLA: A System for Generating Abstract Interpreters. In René Jacquart, editor, *IFIP Congress Topical Sessions*, pages 367–376. Kluwer, 2004.

[LARSW00] Tal Lev-Ami, Thomas Reps, Mooly Sagiv, and Reinhard Wilhelm. Putting Static Analysis to Work for Verification: A Case Study. *ACM SIGSOFT Software Engineering Notes*, 25(5):26–38, 2000.

[LAS00] Tal Lev-Ami and Shmuel Sagiv. TVLA: A System for Implementing Static Analyses. In Jens Palsberg, editor, *Static Analysis, 7th International Symposium, SAS 2000, Proceedings*, volume 1824 of *Lecture Notes in Computer Science*, pages 280–301. Springer, 2000.

[Man03] Roman Manevich. Data Structures and Algorithms for Efficient Shape Analysis. Master's thesis, Tel-Aviv University, Israel, March 25, 2003.

[Mas93] F. Masdupuy. *Array Indices Relational Semantic Analysis Using Rational Cosets and Trapezoids.* PhD thesis, École Polytechnique, Palaiseau, France, December 1993.

[Mic] Microsoft Corporation. Local File Systems for Windows. http://www.microsoft.com/whdc/device/storage/default.mspx.

[MS01] Anders Møller and Michael I. Schwartzbach. The Pointer Assertion Logic Engine. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '01*, June 2001. Also in SIGPLAN Notices 36(5) (May 2001).

[MS04] Roman Manevich and Mooly Sagiv. TVLA: User's Manual (Working Draft), 2004.

[MSSY02] Roman Manevich, Ran Shaham, Mooly Sagiv, and Eran Yahav. J2TVLA: User's Manual, 2002.

[MyS07] MySQL 5.0 Reference Manual. http://dev.mysql.com/doc/refman/5.0/en/, 2007.

[Par05] Sascha Parduhn. Algorithm Animation Using Shape Analysis with Special Regard to Binary Trees. Master's thesis, Saarland University, 2005.

[PW05] Andreas Podelski and Thomas Wies. Boolean Heaps. In Chris Hankin and Igor Siveroni, editors, *Static Analysis, 12th International Symposium, SAS 2005, London, UK, September 7-9, 2005, Proceedings*, volume 3672 of *Lecture Notes in Computer Science*, pages 268–283. Springer, 2005.

[Raj98] Raja Vallee-Rai. The Jimple Framework, 1998.

[Rei] ReiserFS Website. http://www.namesys.com/X0reiserfs.html.

[Rei05] Jan Reineke. Shape Analysis of Sets. Master's thesis, Universität des Saarlandes, Germany, June 2005.

[Rey02] J. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures, 2002.

[RS01] Noam Rinetzky and Mooly Sagiv. Interprocedural Shape Analysis for Recursive Programs. *Lecture Notes in Computer Science*, 2027:133–??, 2001.

[RSW04] Thomas W. Reps, Shmuel Sagiv, and Reinhard Wilhelm. Static Program Analysis via 3-Valued Logic. In Rajeev Alur and Doron Peled, editors, *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings*, volume 3114 of *Lecture Notes in Computer Science*, pages 15–30. Springer, 2004.

[RSY05] Noam Rinetzky, Mooly Sagiv, and Eran Yahav. Interprocedural Shape Analysis for Cutpoint-Free Programs. In Chris Hankin and Igor Siveroni, editors, *Static Analysis, 12th International Symposium, SAS 2005, London, UK, September 7-9, 2005, Proceedings*, volume 3672 of *Lecture Notes in Computer Science*, pages 284–302. Springer, 2005.

[Rug04] Radu Rugina. Quantitative Shape Analysis. In Roberto Giacobazzi, editor, *Static Analysis, 11th International Symposium, SAS 2004, Verona, Italy, August 26-28, 2004, Proceedings*, volume 3148 of *Lecture Notes in Computer Science*, pages 228–245. Springer, 2004.

[SDH+96] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS File System. In *Proceedings of the USENIX 1996 Technical Conference*, pages 1–14, San Diego, CA, USA, January 22–26, 1996.

[SK91] M. Stonebraker and G. Kemnitz. The POSTGRES Next-Generation Database Management System. *Comm. of the ACM, Special Section on Next-Generation Database Systems*, 34(10):78, October 1991.

[SRW98] M. Sagiv, T. Reps, and R. Wilhelm. Solving Shape-Analysis Problems in Languages with Destructive Updating. *ACM Transactions on Programming Languages and Systems*, 20(1):1–50, January 1998.

[SRW02] Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. Parametric Shape Analysis via 3-Valued Logic. *ACM Trans. Program. Lang. Syst*, 24(3):217–298, 2002.

[TV05] Krishna Pradeep Tamma and Shreepadma Venugopalan. Failure Analysis of SGI XFS File System, 2005.

[Č03] P. Černý. Vérification par Interprétation Abstraite de Prédicats Paramétriques, 2003.

[VRHS+99] Raja Vallée-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. Soot - A Java Optimization Framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.

[WCK03a] Chin-Hsien Wu, Li-Pin Chang, and Tei-Wei Kuo. An Efficient B-Tree Layer for Flash-Memory Storage Systems. In *RTCSA*, pages 409–430, 2003.

[WCK03b] Chin-Hsien Wu, Li-Pin Chang, and Tei-Wei Kuo. An Efficient R-Tree Implementation Over Flash-Memory Storage Systems. In *GIS '03: Proceedings of the 11th ACM International Symposium on Advances in Geographic Information Systems*, pages 17–24, New York, NY, USA, 2003. ACM Press.

[Wie04] Thomas Wies. Symbolic Shape Analysis. Master's thesis, Universität des Saarlandes, 2004.

[WKC07] Chin-Hsien Wu, Tei-Wei Kuo, and Li Ping Chang. An Efficient B-Tree Layer Implementation for Flash-Memory Storage Systems. *Trans. on Embedded Computing Sys.*, 6(3):19, 2007.

[WSR00] R. Wilhelm, M. Sagiv, and T. Reps. Shape Analysis. In *International Conference on Compiler Construction*, number 1781 in LNCS, pages 1–16. Springer Verlag, 2000.

[Yah01] Eran Yahav. Verifying Safety Properties of Concurrent Java Programs Using 3-Valued Logic. In *POPL '01: Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 27–40, New York, NY, USA, 2001. ACM Press.

[ZG03] Zhihui Zhang and Kanad Ghose. yFS: A Journaling File System Design for Handling Large Data Sets with Reduced Seeking. In *FAST*. USENIX, 2003.

A Proofs

A.1 Equivalence of Treeness Definitions

Claim 2 *Our instrumentation predicate treeness() gives a definition of treeness equivalent to the following definition:*
A (directed) connected graph $T = (V, E)$ (with a unique root-node) is called a tree if and only if between any two vertices $u, v \in V$ there exists a unique path (by ignoring directions of edges if T is a directed graph).

Proof:

We need to show that

$$\forall\ v_1, v_2, v_3\ leftOfStar(v_1, v_2) \Rightarrow \neg(downStar(v_1, v_3) \wedge downStar(v_2, v_3))$$
$$\Leftrightarrow$$
$$\forall\ u, v \text{ there exists a unique path from } u \text{ to } v$$

$$\Leftarrow: \qquad \text{(contraposition)}$$
$$\exists\ v_1, v_2, v_3\ leftOfStar(v_1, v_2) \wedge (downStar(v_1, v_3) \wedge downStar(v_2, v_3))$$
$$\Rightarrow \exists\ u, v \text{ there is more than 1 path from } u \text{ to } v$$

We first choose nodes u, v_1, v_2 such that $down(u, v_1) \wedge down(u, v_2) \wedge leftOfStar(v_1, v_2)$ holds. We assumed there is a v_3 such that $downStar(v_1, v_3) \wedge downStar(v_2, v_3)$ holds. We can choose this v_3 as v, observe that there exist two paths from u to v, and conclude that this implication holds.
We still have to show the other direction:

$$\Rightarrow: \qquad \text{(contraposition)}$$
$$\exists\ u, v \text{ there is more than 1 path from } u \text{ to } v$$
$$\Rightarrow \exists\ v_1, v_2, v_3\ leftOfStar(v_1, v_2) \wedge (downStar(v_1, v_3) \wedge downStar(v_2, v_3)$$

There are (at least) two paths from u to v (assumption). We call those paths p_1 and p_2. Let v_1 be the first node traversed on p_1 and v_2 the first on p_2. W.l.o.g. let $leftOfStar(v_1, v_2)$ be true (otherwise we can rename v_1 to v_2 and v_2 to v_1, respectively). This construction implies that $downStar(v_1, v) \wedge downStar(v_2, v)$ holds. This completes our proof. ■

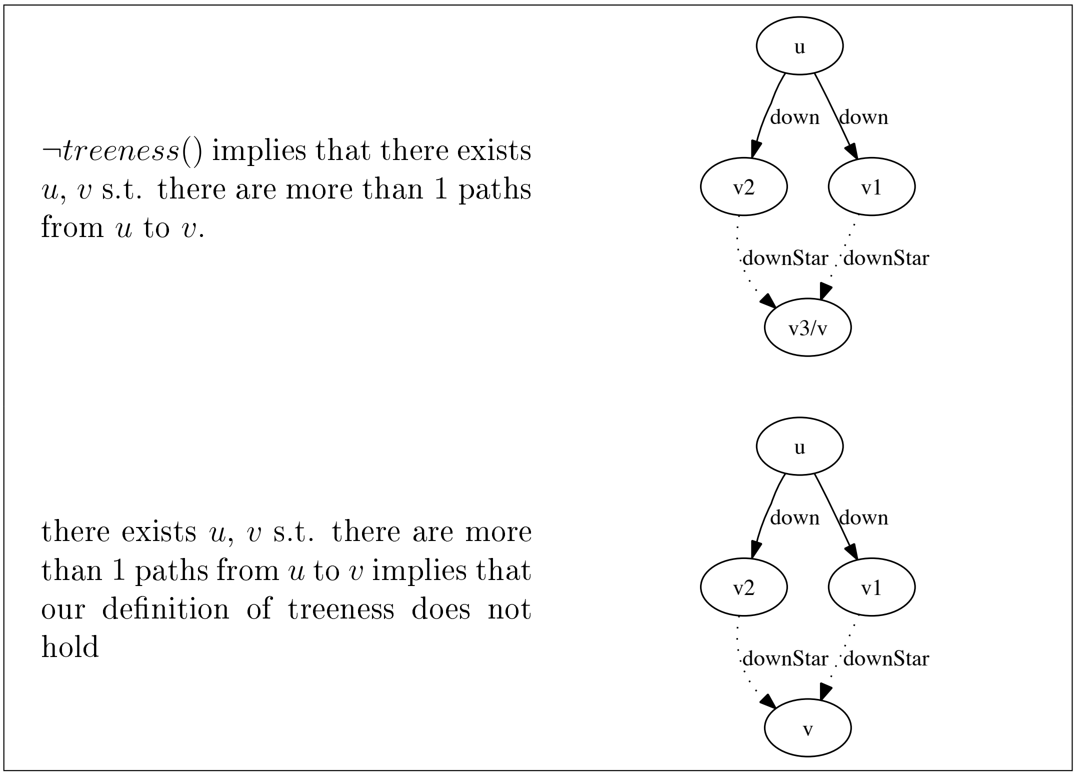
A.2 Formal Proofs for Consistency Constraints

In this section, we give formal proofs that all consistency constraints we used during the analysis are implied by the instrumentation and core predicates.

Let $\mathcal{R} \equiv \varphi_1 \triangleright \varphi_2$ be a consistency rule and $\mathcal{I} = \{p = \varphi_p | \varphi_1 \text{ or } \varphi_2 \text{ contains } p\}$ be the set containing all definitions of instrumentation predicates used in $\mathcal{R}$. In order to safely use $\mathcal{R}$ we need to show that the implication

$$\left(\bigwedge_{p \in \mathcal{I}} p = \varphi_p \right) \Rightarrow (\varphi_1 \Rightarrow \varphi_2)$$

does always hold.



$\neg \mathit{treeness}()$ implies that there exists u, v s.t. there are more than 1 paths from u to v.

there exists u, v s.t. there are more than 1 paths from u to v implies that our definition of treeness does not hold

Figure A.1: Proof sketch.

A.2.1 Consistency Constraints Used in the Analysis of contains

Claim 3 *We can use the* consistency rule $\mathcal{R}$

$$\mathcal{R} \equiv \mathit{isStore}(v) \wedge \mathit{storeProp}(v) \triangleright \neg \mathit{down}(v,u)$$

based on

$$\mathcal{I} = \left\{ \begin{array}{l} \mathit{storeProp}(v) = \exists\ u\ (\mathit{isStore}(v) \wedge \mathit{down}(v,u)) \Rightarrow 0, \\ \mathit{isStore}(v) = \exists\ v'\ \mathit{ie1}(v',v) \vee \mathit{ie2}(v',v) \vee \mathit{ie3}(v',v) \end{array} \right\}$$

Proof:

We assume

$$\mathit{isStore}(v) \wedge \mathit{storeProp}(v) \Rightarrow \neg\exists\ u\ \mathit{down}(v,u)$$

to hold.

$$\begin{array}{rc} & \mathit{isStore}(v) \wedge \mathit{storeProp}(v) \Rightarrow \neg\exists\ u\ \mathit{down}(v,u) \\ \Leftrightarrow & \mathit{isStore}(v) \wedge ((\exists\ u\ \mathit{isStore}(v) \wedge \mathit{down}(v,u)) \Rightarrow 0) \Rightarrow \neg(\exists\ u\ \mathit{down}(v,u)) \\ \Leftrightarrow & \mathit{isStore}(v) \wedge (\neg \mathit{isStore}(v) \vee \neg(\exists\ u\ \mathit{down}(v,u))) \Rightarrow \neg(\exists\ u\ \mathit{down}(v,u)) \\ \Leftrightarrow & \mathit{isStore}(v) \wedge \neg(\exists\ u\ \mathit{down}(v,u)) \Rightarrow \neg(\exists\ u\ \mathit{down}(v,u)) \\ \Leftrightarrow & \neg \mathit{isStore}(v) \vee (\exists\ u\ \mathit{down}(v,u)) \vee \neg(\exists\ u\ \mathit{down}(v,u)) \\ \Leftrightarrow & \neg \mathit{isStore}(v) \vee 1 \\ \Leftrightarrow & 1 \end{array}$$

■

Claim 4 *The consistency constraint*

$$\neg downStar(u,v) \triangleright \neg down(u,v)$$

can be used.

Proof:

Let $down(u,v)$ be a given predicate and $downStar(u,v)$ the reflexive transitive closure of that $down$ predicate.
The reflexive transitive closure $brel^*$ of a binary relation (or in our case predicate) $brel$ is in logical terms defined as

$$brel^*(u,v) \Leftrightarrow u = v \vee brel^+(u,v)$$

where $brel^+(u,v)$ is defined as

$$brel^+(u,v) \Leftrightarrow brel(u,v) \vee \exists\ n \geq 1\ \exists\ w_1,\ldots,w_n\ brel(u,w_1) \wedge brel(w_1,w_2) \wedge \ldots \wedge brel(w_n,v)$$

Using those definitions, we can easily show our claim:

$$\begin{array}{cc}
 & \neg downStar(u,v) \Rightarrow \neg down(u,v) \\
\Leftrightarrow & \neg\,(u = v \vee down^+(u,v)) \Rightarrow \neg down(u,v) \\
\Leftrightarrow & u \neq v \wedge \neg down^+(u,v) \Rightarrow \neg down(u,v) \\
\Leftrightarrow & \neg down(u,v) \vee u = v \vee down^+(u,v) \\
\Leftrightarrow & \neg down(u,v) \vee down(u,v) \vee down^{>1}(u,v) \vee u = v \\
\Leftrightarrow & 1 \vee u = v \vee down^{>1}(u,v) \\
\Leftrightarrow & 1
\end{array}$$

where

$$\begin{array}{rcl}
down^{>1}(u,v) & = & \exists\ n \geq 1\ \exists\ w_1,\ldots,w_n \\
 & & down(u,w_1) \wedge \\
 & & \bigwedge_{1 \leq i \leq n-1} down(w_i,w_{i+1}) \\
 & & \wedge\ down(w_n,v)
\end{array}$$

■

Claim 5 *The consistency constraint*

$$(treeness() \wedge (\exists\ a\ down(a,b) \wedge a \neq c)) \triangleright \neg down(c,b)$$

is implied by the set of instrumentation and core predicates.

Proof:

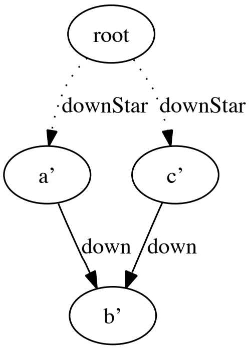


Figure A.2: A contradictory structure.

According to our definition of *treeness*() and our construction of tree structures which guarantees a unique root node, we can conclude that if *treeness*() holds the following formula evaluates to true:

$$\forall\ a', b', c'\ down(a', b') \wedge down(c', b') \Rightarrow a' = c'$$

We can easily convince ourselves that this formula must indeed hold. Suppose it does not hold. Then

$$\exists\ a', b', c'\ down(a', b') \wedge down(c', b') \wedge a' \neq c'$$

But in this case we have a situation as depicted in Figure A.2 in which there are two paths from the root node to b' (a' and b' cannot both be root nodes). This contradicts the fact that the structure is a directed tree.

It remains to show that

$$\forall\ b, c\ \ (treeness() \wedge (\exists\ a\ down(a, b) \wedge a \neq c) \Rightarrow \neg down(c, b))$$

does always hold. We rewrite this as

$$\begin{aligned} & \forall\ b, c\ \ ((\exists\ a\ treeness() \wedge down(a, b) \wedge a \neq c) \Rightarrow \neg down(c, b)) \\ \Leftrightarrow\quad & \forall\ a, b, c\ \ (\neg treeness() \vee \neg down(a, b) \vee a = c \vee \neg down(c, b)) \end{aligned}$$

We can replace treeness by the formula introduced above which we can assume to hold for any triple a', b', c'. Hence, it holds in particular for a, b, and c. Thus:

$$\begin{aligned} & \forall\ a, b, c \quad \neg(down(a, b) \wedge down(c, b) \Rightarrow a = c) \vee \neg down(a, b) \vee a = c \vee \neg down(c, b) \\ \Leftrightarrow & \forall\ a, b, c \quad \neg(\neg down(a, b) \vee \neg down(c, b) \vee a = c) \vee (\neg down(a, b) \vee a = c \vee \neg down(c, b)) \\ \Leftrightarrow & \forall\ a, b, c \quad 1 \end{aligned}$$

■

Claim 6 *We can safely use a consistency constraint*

$$\neg down(u, v) \triangleright \neg sel(u, v)$$

for all sel $\in$ *Sel.*

Proof:

The *down*-predicate was defined as

$$down(u,v) = \bigvee_{sel \in Sel} sel(u,v)$$

Using this definition we can easily verify the claim:

$$\begin{array}{rlr}
& \neg down(u,v) \Rightarrow \neg sel(u,v) & (sel \in Sel) \\
\Leftrightarrow & \neg \bigvee_{sel' \in Sel} sel'(u,v) \Rightarrow \neg sel(u,v) & (sel \in Sel) \\
\Leftrightarrow & \bigvee_{sel' \in Sel} sel'(u,v) \vee \neg sel(u,v) & (sel \in Sel) \\
\Leftrightarrow & 1 \vee \bigvee_{sel' \in Sel \setminus \{sel\}} sel'(u,v) & (sel \in Sel) \\
\Leftrightarrow & 1 &
\end{array}$$

■

Claim 7 *Our definition of treeness implies the following two consistency constraints:*

$$\exists\ u\ treeness() \wedge rightSubTree(u,v) \wedge leftSubTree(u,w) \rhd \neg downStar(v,w)$$

$$\exists\ u\ treeness() \wedge rightSubTree(u,v) \wedge leftSubTree(u,w) \rhd \neg downStar(w,v)$$

Proof:

We limit ourselves to showing the first implication. The second constraint can be shown analogous.

$$\begin{array}{rl}
& \forall\ v,w\ (\exists\ u\ treeness() \wedge rightSubTree(u,v) \wedge leftSubTree(u,w)) \Rightarrow \neg downStar(v,w) \\
\Leftrightarrow & \forall\ u,v,w\ \neg(treeness() \wedge rightSubTree(u,v) \wedge leftSubTree(u,w)) \vee \neg downStar(v,w)
\end{array}$$

Instead of showing that the last formula is true, we show that its negation

$$\exists\ u,v,w\ (treeness() \wedge rightSubTree(u,v) \wedge leftSubTree(u,w)) \wedge downStar(v,w)$$

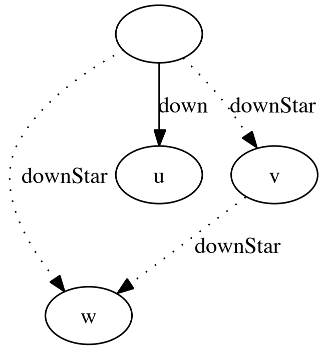
is false. We settle for a proof by contradiction. Suppose such u, v, w do exist. We would end up with a structure as depicted in Figure A.3. However, we already showed that our definition of treeness is equivalent to saying *between any two nodes exists a unique path*. Clearly, the depicted structure shows two paths from the parent node of u to w. We therefore conclude that

$$\forall\ u,v,w\ \neg(treeness() \wedge rightSubTree(u,v) \wedge leftSubTree(u,w)) \vee \neg downStar(v,w)$$

must hold.

■

$$treeness() \wedge rightSubTree(u,v) \wedge$$
$$leftSubTree(u,w) \wedge downStar(v,w)$$



Formula Graphical representation

Figure A.3: A contradictory structure.

Claim 8 *The following consistency constraints are implied by the instrumentation predicates.*

$$\begin{aligned}
\exists\, u\ leftSubTree(u,v) \wedge ie1(w,u) &\ \triangleright\ \neg cleft(w,v)\\
\exists\, u\ leftSubTree(u,v) \wedge ie1(w,u) &\ \triangleright\ \neg cright(w,v)\\
\exists\, u\ leftSubTree(u,v) \wedge ie1(w,u) &\ \triangleright\ \neg right(w,v)\\
\exists\, u\ leftSubTree(u,v) \wedge ie1(w,u) &\ \triangleright\ \neg ie2(w,v)\\
\exists\, u\ leftSubTree(u,v) \wedge ie1(w,u) &\ \triangleright\ \neg ie3(w,v)\\
\exists\, u\ leftSubTree(u,v) \wedge ie2(w,u) &\ \triangleright\ \neg cright(w,v)\\
\exists\, u\ leftSubTree(u,v) \wedge ie2(w,u) &\ \triangleright\ \neg right(w,v)\\
\exists\, u\ leftSubTree(u,v) \wedge ie2(w,u) &\ \triangleright\ \neg ie3(w,v)\\
\exists\, u\ leftSubTree(u,v) \wedge ie3(w,u) &\ \triangleright\ \neg right(w,v)\\
\exists\, u\ rightSubTree(u,v) \wedge ie1(w,u) &\ \triangleright\ \neg left(w,v)\\
\exists\, u\ rightSubTree(u,v) \wedge ie2(w,u) &\ \triangleright\ \neg left(w,v)\\
\exists\, u\ rightSubTree(u,v) \wedge ie2(w,u) &\ \triangleright\ \neg cleft(w,v)\\
\exists\, u\ rightSubTree(u,v) \wedge ie2(w,u) &\ \triangleright\ \neg ie1(w,v)\\
\exists\, u\ rightSubTree(u,v) \wedge ie3(w,u) &\ \triangleright\ \neg left(w,v)\\
\exists\, u\ rightSubTree(u,v) \wedge ie3(w,u) &\ \triangleright\ \neg cleft(w,v)\\
\exists\, u\ rightSubTree(u,v) \wedge ie3(w,u) &\ \triangleright\ \neg cright(w,v)\\
\exists\, u\ rightSubTree(u,v) \wedge ie3(w,u) &\ \triangleright\ \neg ie1(w,v)\\
\exists\, u\ rightSubTree(u,v) \wedge ie3(w,u) &\ \triangleright\ \neg ie2(w,v)
\end{aligned}$$

Proof:

The proofs work similar for all those constraints. Therefore, we exemplarily show the first consistency constraint. For the first constraint to be valid, we need to show that the following implication does always hold:

$$\forall\, v,w\ (\exists\, u\ leftSubTree(u,v) \wedge ie1(w,u)) \Rightarrow \neg cleft(w,v)$$

We argue that we can replace $leftSubTree(u,v)$ by $rightOfStar(u,v)$ because these are the only v reachable via one selector. Only for these can $cleft(w,v)$ potentially be true. Substituting and eliminating the implication symbol in the formula yields

$$\forall\ u, v, w\ \neg rightOfStar(u,v) \vee \neg ie1(w,u) \vee \neg cleft(w,v)$$

We show this by proving that

$$\exists\ u, v, w\ rightOfStar(u,v) \wedge ie1(w,u) \wedge cleft(w,v)$$

cannot be satisfied. Assume that $ie1(w,u) \wedge cleft(w,v)$ holds, otherwise we do not have to show anything further. But if $ie1(w,u) \wedge cleft(w,v)$ is true $rightOfStar(u,v)$ must be false, considering the definition of $rightOfStar(u,v)$. ■

Claim 9 *The p2_5-predicate implies the following consistency constraints:*

1. $\forall\ v_1\quad \nexists\ v\ left(v_1,v) \wedge p2\_5(v) \triangleright \nexists\ v_2\ cleft(v_1,v_2)$
2. $\forall\ v_1\quad \nexists\ v\ cleft(v_1,v) \wedge p2\_5(v) \triangleright \nexists\ v_2\ left(v_1,v_2)$
3. $\forall\ v_1\quad \nexists\ v\ ie1(v_1,v) \wedge p2\_5(v) \triangleright \nexists\ v_2\ ie2(v_1,v_2)$
4. $\forall\ v_1\quad \nexists\ v\ ie2(v_1,v) \wedge p2\_5(v) \triangleright \nexists\ v_2\ ie3(v_1,v_2)$
5. $\forall\ v_1\quad \nexists\ v\ ie1(v_1,v) \wedge p2\_5(v) \triangleright \nexists\ v_2\ left(v_1,v_2)$
6. $\forall\ v_1\quad \nexists\ v\ ie2(v_1,v) \wedge p2\_5(v) \triangleright \nexists\ v_2\ cright(v_1,v_2)$
7. $\forall\ v_1\quad \nexists\ v\ ie3(v_1,v) \wedge p2\_5(v) \triangleright \nexists\ v_2\ right(v_1,v_2)$
8. $\forall\ v_1\quad \nexists\ v\ cleft(v_1,v) \wedge p2\_5(v) \triangleright \nexists\ v_2\ cright(v_1,v_2)$
9. $\forall\ v_1\quad \nexists\ v\ cright(v_1,v) \wedge p2\_5(v) \triangleright \nexists\ v_2\ right(v_1,v_2)$
10. $\forall\ v_1\ \neg(\exists\ u\ right(v_1,u)) \wedge (\exists\ v\ cright(v_1,v)) \wedge p2\_5(v) \triangleright \nexists\ v_2\ ie3(v_1,v_2)$

Proof:

From the definition of $p2\_5(v)$ we can conclude that $p2\_5(v)$ holds if and only if each of the following expressions is true:

i $(\exists\ u\ left(v,u)) \Leftrightarrow (\exists\ u'\ cleft(v,u'))$
ii $(\exists\ u\ cright(v,u)) \Rightarrow (\exists\ u'\ cleft(v,u'))$
iii $(\exists\ u\ cright(v,u)) \Rightarrow (\exists\ u'\ ie2(v,u'))$
iv $(\exists\ u\ right(v,u)) \Rightarrow (\exists\ u'\ cright(v,u'))$
v $(\exists\ u\ right(v,u)) \Rightarrow (\exists\ u'\ ie3(v,u'))$
vi $(\exists\ u\ ie2(v,u)) \Rightarrow (\exists\ u'\ ie1(v,u'))$
vii $(\exists\ u\ ie3(v,u)) \Rightarrow (\exists\ u'\ ie2(v,u'))$
viii $(\exists\ u\ left(v,u)) \Rightarrow (\exists\ u'\ ie1(v,u'))$

ix $(\exists\, u, w\ ie3(v,u) \wedge cright(v,w)) \Rightarrow (\exists\, u'\ right(v,u'))$

In order to show that (1) is implied, we have to show that

$$\forall\, v_1 \quad \nexists\, v\ left(v_1, v) \wedge p2\_5(v_1) \Rightarrow \nexists\, v_2\ cleft(v_1, v_2)$$

always holds. Substituting $p2\_5(v_1)$ by $(\exists\, u\ cleft(v_1,u)) \Rightarrow (\exists\, u'\ left(v_1,u'))$ (which holds because (i) does hold) yields:

$$\begin{array}{ll} \forall\, v_1 & \neg(\exists\, v\ left(v_1,v)) \wedge (\neg(\exists\, u\ cleft(v_1,u)) \vee (\exists\, u'\ left(v_1,u'))) \Rightarrow \neg(\exists\, v_2\ cleft(v_1,v_2)) \\ \forall\, v_1 & (\exists\, v\ left(v_1,v)) \vee ((\exists\, u\ cleft(v_1,u)) \wedge \neg(\exists\, u'\ left(v_1,u'))) \vee \neg(\exists\, v_2\ cleft(v_1,v_2)) \\ \forall\, v_1 & (\exists\, v\ left(v_1,v)) \vee \neg(\exists\, u'\ left(v_1,u')) \vee \neg(\exists\, v_2\ cleft(v_1,v_2)) \Leftrightarrow \forall\, v_1 \quad 1 \end{array}$$

We can show (2) in a similar way by substituting $p2\_5(v_1)$ by $(\exists\, u'\ left(v_1,u')) \Rightarrow (\exists\, u\ cleft(v_1,u))$. The remaining consistency constraints can be easily shown by substituting $p2\_5(v_1)$ by the expression whose selector predicates matches those of the constraint. That is, in order to show (3) we use (ii), to show (4) we use (iii), and so on. ■

Claim 10 *The inOrder()-predicate implies*

$$\begin{array}{rcl} inOrder() \wedge rightSubTree(u,v) & \triangleright & kge(v,u) \\ inOrder() \wedge rightSubTree(u,v) & \triangleright & \neg kge(u,v) \\ inOrder() \wedge leftSubTree(u,v) & \triangleright & kge(u,v) \\ inOrder() \wedge leftSubTree(u,v) & \triangleright & \neg kge(v,u) \end{array}$$

Proof:

Again, we will just show the first implied constraint. The remaining three are shown analogously.

$$inOrder() \wedge rightSubTree(u,v) \Rightarrow kge(v,u)$$

We substitute $inOrder()$ by $rightSubTree(u,v) \Rightarrow kge(v,u) \wedge \neg kge(u,v)$ which, by definition of $inOrder()$, holds if $inOrder()$ holds

$$\begin{array}{rc} & (rightSubTree(u,v) \Rightarrow kge(v,u) \wedge \neg kge(u,v)) \wedge rightSubTree(u,v) \Rightarrow kge(v,u) \\ \Leftrightarrow & \neg(\neg rightSubTree(u,v) \vee (kge(v,u) \wedge \neg kge(u,v))) \vee \neg rightSubTree(u,v) \vee kge(v,u) \\ \Leftrightarrow & rightSubTree(u,v) \wedge (\neg kge(v,u) \vee kge(u,v)) \vee \neg rightSubTree(u,v) \vee kge(v,u) \\ \Leftrightarrow & 1 \wedge (\neg kge(v,u) \vee kge(u,v) \vee \neg rightSubTree(u,v)) \vee kge(v,u) \\ \Leftrightarrow & \neg kge(v,u) \vee kge(u,v) \vee \neg rightSubTree(u,v) \vee kge(v,u) \\ \Leftrightarrow & 1 \end{array}$$

■

Claim 11 *The consistency constraint*

$$inOrder() \wedge (\exists\, b\ kge(a,b) \wedge kge(b,c) \wedge a \neq b \wedge a \neq c \wedge b \neq c) \triangleright \neg kge(c,a)$$

follows from our definition of inOrder().

Proof:

It suffices to show that the *inOrder*-predicate guarantees that the keys stored in a tree are unique. This is quite obvious. Let u and v be two arbitrary heap cells storing a value. We can always find heap cells p such that $downStar(p,u) \wedge downStar(p,v) \wedge \neg(downStar(p',u) \wedge downStar(p',v))$ for all child nodes p' of p holds. This p is root for at least two subtrees where one contains u and the other v. $inOrder()$ implies that keys from different subtrees cannot contain equal keys. Hence, the arbitrarily chosen nodes u and v must store different keys. ■

Claim 12 *The greRelation-predicate implies*

$$\forall\ u,v\ greRelation() \wedge \neg kge(u,v) \rhd kge(v,u)$$

Proof:

Replacing $gerRelation()$ with its defining formula in the implication we want to show yields:

$$\begin{aligned} &\forall\ u,v && (\neg kge(u,v) \Rightarrow kge(v,u)) \wedge \neg kge(u,v) \Rightarrow kge(v,u) \\ \Leftrightarrow &\forall\ u,v && (kge(u,v) \vee kge(v,u)) \wedge \neg kge(u,v) \Rightarrow kge(v,u) \\ \Leftrightarrow &\forall\ u,v && \neg(kge(u,v) \vee kge(v,u)) \vee (kge(u,v) \vee kge(v,u)) \\ \Leftrightarrow &\forall\ u,v && 1 \end{aligned}$$

■

Claim 13 *The following two consistency constraints are implied by our definitions of the $leftSubTree$- and $rightSubTree$-predicates.*

$$\begin{aligned} \forall\ u,w\ ((\exists\ p\ \neg rightSubTree(u,w) \wedge downStar(p,w) \wedge \\ down(p,u) \wedge \neg downStar(u,w) \\ \wedge w \neq u \wedge w \neq p \wedge u \neq p) \quad &\rhd \quad leftSubTree(u,w)) \\ \forall\ u,w\ ((\exists\ p\ \neg leftSubTree(u,w) \wedge downStar(p,w) \wedge \\ down(p,u) \wedge \neg downStar(u,w) \\ \wedge w \neq u \wedge w \neq p \wedge u \neq p) \quad &\rhd \quad rightSubTree(u,w)) \end{aligned}$$

Proof:

As both proofs work analogously we again limit ourselves to showing just the first one. We assume that the implication

$$\begin{aligned} \forall\ u,w\ ((\exists\ p\ \neg rightSubTree(u,w) \wedge downStar(p,w) \wedge \\ down(p,u) \wedge \neg downStar(u,w) \\ \wedge w \neq u \wedge w \neq p \wedge u \neq p) \quad &\Rightarrow \quad leftSubTree(u,w)) \end{aligned}$$

would not hold. I.e. there may exist a triple (u,w,p) of pairwise different heap cells such that

$$\neg rightSubTree(u,w) \wedge downStar(p,w) \wedge down(p,u) \wedge \neg downStar(u,w)$$

$$\wedge \neg leftSubTree(u, w)$$

holds.
This, however, leads to a contradiction. The heap cell w is reachable from heap cell p because of the satisfied $downStar(p, w)$ predicate. u is directly reachable from p. Every other cells directly reachable from p are either left of u or right of u. As neither $leftSubTree(u, w)$ nor $rightSubTree(u, w)$ holds per assumption, we conclude that w must be rooted at u otherwise it cannot be reachable from p. But this also cannot be because $\neg downStar(u, w)$ does hold. Hence, our assumption leads to a contradiction and the implication for our consistency constraint must hold. ■

A.2.2 Consistency Constraints Used in the Analysis of insert

Claim 14 *The consistency constraint*

$$treeness() \wedge downStar(u, v) \wedge u \neq v \triangleright \neg level(u, v)$$

is implied by the definition of level and downStar.

Proof:

We need show that the implication

$$treeness() \wedge (u = v \vee down^{+}(u, v)) \wedge u \neq v \Rightarrow \neg\,(u = v \vee \exists\; a, b\; level(a, b) \wedge down(a, u) \wedge down(b, v))$$

does always hold. Some simplifications yield:

$$\forall\; a, b, u, v\; \neg(down^{+}(u, v)) \vee u = v \vee \neg level(a, b) \vee \neg down(a, u) \vee \neg down(b, v) \vee \neg treeness()$$

We proof that this formula is always satisfied by showing that its negation leads to a contradiction. Hence, suppose such a, b, u, and v satisfying

$$(down^{+}(u, v)) \wedge u \neq v \wedge level(a, b) \wedge down(a, u) \wedge down(b, v) \wedge treeness()$$

would exist. Then u is reachable from a, v from b, v also from u and a and b both from the root node. Now, we have two paths to reach v from the root node, one over a and u - as u and v are different nodes - and one directly over b. Note, that a and b might be the same node, due to one path going over u, we still have found two paths to reach v. This, however, contradicts the treeness of the structure. And thus, we may conclude that the implication must hold. ■

B Source Code

B.1 TVLA Input Files

B.1.1 Predicate Files

Predicate File for Contains

```
/* Variables we don’t want to have shown in the output, most of these were
 * introduced by the Java compiler.
 */
%s PVarInvisible {$r3, $r4, $r5, $r6, $r7, $r8, $r9, $r10, $r11, $r12, $r13,
    $i0, $i1, $i2, $i3, $i4, $i5, $i6, $i7, $i8, $i9, $i10, $i11, $i12, $i13,
    $return}

// The program variables we are interested in
%s PVarVisible {$parameter0, $parameter1, return, r1, r2, r0}

// selector predicates (child pointers and pointers to stored values)
%s TSelEl {left, cleft, cright, right, ie1, ie2, ie3}

/*****************************************************************************/
/*** core predicates                                                       ***/
/*****************************************************************************/

foreach ( z in PVarInvisible ) {
    %p z(v1) unique pointer abs {}
}

foreach ( z in PVarVisible ) {
    %p z(v1) unique pointer abs
}

foreach (sel in TSelEl) {
  %p sel(v1, v2) function
}

// key(s) greater equal
%p kge(v1, v2) reflexive transitive {}

//
%p heapcell(v) nonabs {}
/*****************************************************************************/
/*** instrumentation predicates                                            ***/
/*****************************************************************************/
// The down predicate represents the union of selector predicates.
%i down(v1, v2) = |/{ sel(v1, v2) : sel in TSelEl } {}

// 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 antisymmetric
```

```
// For every program variable z the predicate r[z] holds for individual
// v when v is reachable from variable z along the selector/store fields.
foreach (x in PVarVisible - {r0}) {
  %i r[x](v) = E(v1) (x(v1) & downStar(v1, v)) nonabs {}
}
foreach (x in PVarInvisible) {
  %i r[x](v) = E(v1) (x(v1) & downStar(v1, v)) nonabs {}
}
// reachability from r0
%i r[r0](v) = E(v1) (r0(v1) & downStar(v1, v)) abs

%i isStore(v) = E(v1) (ie1(v1, v) | ie2(v1, v) | ie3(v1, v)) nonabs
//%i storeProp(v) = isStore(v) -> !(E(v1)down(v, v1)) nonabs {}
%i storeProp(v) = (isStore(v) & E(v1)(down(v, v1))) -> 0 nonabs {}
//////////
%i leftOf(u,v) = ( E(w) (left(w,u) & ie1(w,v)) )
                |
                 ( E(w) (ie1(w,u) & cleft(w,v)) )
                |
                 ( E(w) (cleft(w,u) & ie2(w,v)) )
                |
                 ( E(w) (ie2(w,u) & cright(w,v)) )
                |
                 ( E(w) (cright(w,u) & ie3(w,v)) )
                |
                 ( E(w) (ie3(w,u) & right(w,v)) )

                |
                 ( E(w) (ie1(w,u) & ie2(w,v)) & !E(l) left(w,l) )
                |
                 ( E(w) (ie2(w,u) & ie3(w,v)) & !E(l) left(w,l) )
                {}

%i leftOfStar(u,v) = leftOf+(u,v) transitive {}
%i rightSubTree(u,v) = E(w) (leftOfStar(u,w) & downStar(w,v)) {}

//
%i rightOf(u,v) = ( E(w) (right(w,u) & ie3(w,v)) )
                |
                 ( E(w) (ie3(w,u) & cright(w,v)) )
                |
                 ( E(w) (cright(w,u) & ie2(w,v)) )
                |
                 ( E(w) (ie2(w,u) & cleft(w,v)) )
                |
                 ( E(w) (cleft(w,u) & ie1(w,v)) )
                |
                 ( E(w) (ie1(w,u) & left(w,v)) )

                |
                 ( E(w) (ie2(w,u) & ie1(w,v)) & !E(l) left(w,l) )
                |
                 ( E(w) (ie3(w,u) & ie2(w,v)) & !E(l) left(w,l) )
                {}

%i rightOfStar(u,v) = rightOf+(u,v) transitive {}
%i leftSubTree(u,v) = E(w) (rightOfStar(u,w) & downStar(w,v)) {}
//
%i treeness() =
    (A(v1,v2,v3)leftOfStar(v1,v2) -> !(downStar(v1,v3) & downStar(v2,v3)))


%i inOrder() = (A(v,u) (leftSubTree(u,v) -> kge(u,v) & !kge(v,u)))
                    &
                (A(v,u) (rightSubTree(u,v) -> kge(v,u) & !kge(u,v)))

%i kge[$parameter1, left](v) = E(u) $parameter1(u) & kge(v,u) & !kge(u,v) abs
%i kge[$parameter1, right](v) = E(u) $parameter1(u) & !kge(v,u) & kge(u,v) abs

//
%i isElement(u,v) = E(w) (downStar(v,w) & kge(w,u) & kge(u,w) & /*v!=w &*/ isStore(w))

//
%i p2_5(v) = ((E(u) left(v,u)) <-> (E(u1) cleft(v,u1)))     // min 2 child nodes
                    &
            ((E(u) cright(v,u)) -> (E(u1) cleft(v,u1)))
                    &
            ((E(u) cright(v,u)) -> (E(u1) ie2(v,u1)))
                    &
            ((E(u) right(v,u)) -> (E(u1) cright(v,u1)))
                    &
            ((E(u) right(v,u)) -> (E(u1) ie3(v,u1)))
                    &
            ((E(u) ie2(v,u) -> (E(u1) ie1(v,u1))))
                    &
            ((E(u) ie3(v,u) -> (E(u1) ie2(v,u1))))
                    &
            ((E(u) left(v,u) -> (E(u1) ie1(v,u1))))
                    &
            ((E(u,w) ie3(v,u) & cright(v,w)) -> (E(u1) right(v,u1)))
            nonabs {}

%i greRelation() = A(u,v) !kge(u,v) -> kge(v,u)
/*****************************************************************************/
/*** consistency rules                                                     ***/
/*****************************************************************************/
// follows from storeProp
%r isStore(v) & heapcell(u) & storeProp(v) ==> !down(v,u)

// follows from down/downStar
%r !downStar(u,v) ==> !down(u,v)
```

```
// follows from treeness and down
%r treeness() & E(u)(down(u,v1) & u != v2) ==> !down(v2,v1)
foreach (sel in TSelEl) {
    %r !down(v1,v2) ==> !sel(v1,v2)
}

// follows from treeness
foreach (s1 in TSelEl) {
    foreach (s2 in TSelEl - {s1}) {
        %r treeness() & s1(v1,v2) ==> !s2(v1,v2)
    }
}

%r E(u)(treeness() & rightSubTree(u,v) & leftSubTree(u,w)) ==> !downStar(v,w)
%r E(u)(treeness() & rightSubTree(u,v) & leftSubTree(u,w)) ==> !downStar(w,v)

//
%r E(u)(leftSubTree(u,v) & ie1(w,u)) ==> !cleft(w,v)
%r E(u)(leftSubTree(u,v) & ie1(w,u)) ==> !cright(w,v)
%r E(u)(leftSubTree(u,v) & ie1(w,u)) ==> !right(w,v)
%r E(u)(leftSubTree(u,v) & ie1(w,u)) ==> !ie2(w,v)
%r E(u)(leftSubTree(u,v) & ie1(w,u)) ==> !ie3(w,v)

%r E(u)(leftSubTree(u,v) & ie2(w,u)) ==> !cright(w,v)
%r E(u)(leftSubTree(u,v) & ie2(w,u)) ==> !right(w,v)
%r E(u)(leftSubTree(u,v) & ie2(w,u)) ==> !ie3(w,v)

%r E(u)(leftSubTree(u,v) & ie3(w,u)) ==> !right(w,v)

%r E(u)(rightSubTree(u,v) & ie1(w,u)) ==> !left(w,v)

%r E(u)(rightSubTree(u,v) & ie2(w,u)) ==> !left(w,v)
%r E(u)(rightSubTree(u,v) & ie2(w,u)) ==> !cleft(w,v)
%r E(u)(rightSubTree(u,v) & ie2(w,u)) ==> !ie1(w,v)

%r E(u)(rightSubTree(u,v) & ie3(w,u)) ==> !left(w,v)
%r E(u)(rightSubTree(u,v) & ie3(w,u)) ==> !cleft(w,v)
%r E(u)(rightSubTree(u,v) & ie3(w,u)) ==> !cright(w,v)
%r E(u)(rightSubTree(u,v) & ie3(w,u)) ==> !ie1(w,v)
%r E(u)(rightSubTree(u,v) & ie3(w,u)) ==> !ie2(w,v)

//
%r !(E(v) left(v1, v)) & heapcell(w) & p2_5(v1) ==> !cleft(v1,v2)
%r !(E(v) cleft(v1, v)) & heapcell(w) & p2_5(v1) ==> !left(v1,v2)


%r !(E(v) ie1(v1, v)) & heapcell(v2) & p2_5(v1) ==> !ie2(v1, v2)
%r !(E(v) ie2(v1, v)) & heapcell(v2) & p2_5(v1) ==> !ie3(v1, v2)

%r !(E(v) ie1(v1, v)) & heapcell(v2) & p2_5(v1) ==> !left(v1,v2)
%r !(E(v) ie2(v1, v)) & heapcell(v2) & p2_5(v1) ==> !cright(v1,v2)
```

```
%r !(E(v) ie3(v1, v)) & heapcell(v2) & p2_5(v1) ==> !right(v1,v2)

%r !(E(v) cleft(v1, v)) & heapcell(v2) & p2_5(v1) ==> !cright(v1, v2)
%r !(E(v) cright(v1, v)) & heapcell(v2) & p2_5(v1) ==> !right(v1, v2)

%r !(E(u)right(v1,u))&(E(v)cright(v1, v)) & heapcell(v2) & p2_5(v1) ==> !ie3(v1,v2)

//
%r inOrder() & rightSubTree(u,v) ==> kge(v,u)
%r inOrder() & rightSubTree(u,v) ==> !kge(u,v)
%r inOrder() & leftSubTree(u,v) ==> kge(u,v)
%r inOrder() & leftSubTree(u,v) ==> !kge(v,u)
//
//
%r inOrder() & E(b)(kge(a,b) & kge(b,c) & a != b & a != c & b != c) ==> !kge(c,a)
//
%r greRelation() & !kge(u,v) ==> kge(v,u)

%r E(p)(!rightSubTree(u,w) & downStar(p,w) & down(p,u) & !downStar(u,w)
        & w != u & w != p & u != p) ==> leftSubTree(u,w)
%r E(p)(!leftSubTree(u,w) & downStar(p,w) & down(p,u) & !downStar(u,w)
        & w != u & w != p & u != p) ==> rightSubTree(u,w)
```

Predicate File for Insert (Partial)

This file can be found on the enclosed CD-ROM.

B.1.2 Action Files

Action File for Contains

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

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

%action AssignFieldRefToRef(x1, x2, x3) {
  %t x1 + " = " + x2 + "." + x3
  %f {
        x2(v_1) & x3(v_1, v),
        x2(v_1) & x3(v_1, v_2) & downStar(v_2, v)
  }
  {
```

```
x1(v) = E(v_1) x2(v_1) & x3(v_1, v)
r[x1](v) = E(v_1,v_2) x2(v_1) & x3(v_1, v_2) & downStar(v_2, v)
  }
}

%action SetNull(x1) {
  %t x1 + " = null"
  {
    x1(v)    = 0
    r[x1](v) = 0
  }
}

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

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

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

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

%action GreaterEqualKey(x1, x2) {
  %t x1 + "->key >= " + x2 + "->key"
  %f {
        x1(v_1) & x2(v_2) & kge(v_1, v_2),
        x2(u) & rightSubTree(u,v),
        x2(u) & leftSubTree(u,v)
     }
  %p E(v_1, v_2) x1(v_1) & x2(v_2) & kge(v_1, v_2)
  %message (!(E(v) x2(v)) | !(E(v) x1(v))) -> "null pointer exception\n"
}

%action NotGreaterEqualKey(x1, x2) {
  %t x1 + "->key < " + x2 + "->key"
  %f {
        x1(v_1) & x2(v_2) & !kge(v_1, v_2),
        x2(u) & rightSubTree(u,v),
        x2(u) & leftSubTree(u,v)
     }
  %p E(v_1, v_2) x1(v_1) & x2(v_2) & !kge(v_1, v_2)
      %message (!(E(v) x2(v)) | !(E(v) x1(v))) -> "null pointer exception\n"
}

%action EqualKey(x1, x2) {
    %t x1 + " != " + x2
    %f { x1(v_1) & x2(v_2) & kge(v_1, v_2) & kge(v_2, v_1) }
    %p ( E(v_1, v_2) x1(v_1) & x2(v_2) & kge(v_1, v_2) & kge(v_2, v_1) )
    %message (!(E(v) x2(v)) | !(E(v) x1(v))) -> "null pointer exception\n"
}

%action NotEqualKey(x1, x2) {
    %t x1 + " != " + x2
    %f { x1(v_1) & x2(v_2) & kge(v_1, v_2) & kge(v_2, v_1) }
    %p !( E(v_1, v_2) x1(v_1) & x2(v_2) & kge(v_1, v_2) & kge(v_2, v_1) )
    %message (!(E(v) x2(v)) | !(E(v) x1(v))) -> "null pointer exception\n"
}


%action StructuresOK() {
    %p treeness() & inOrder() & (A(v) p2_5(v)) &
        (
(E(r,k1,k2, root)$parameter0(root) & $parameter1(k1) & isElement(k1,root) & return(r) & down(r, k2)
        |
(E(k1,k2, root)$parameter0(root) & $parameter1(k1) & !isElement(k1,root) & !(E(r)return(r)))
        )
}

%action StructuresNOK() {
    %p !(treeness() & inOrder() & (A(v) p2_5(v)) &
        (
(E(r,k1,k2, root)$parameter0(root) & $parameter1(k1) & isElement(k1,root) & return(r) & down(r, k2)
        |
(E(k1,k2, root)$parameter0(root) & $parameter1(k1) & !isElement(k1,root) & !(E(r)return(r)))
        ))
}
```

Action File for Insert

This file can be found on the enclosed CD-ROM.

B.1.3 Input-Structures Files

Input-Structures for Contains

```
%n = {r,ie}
%p = {
    heapcell = {r, ie}
    $parameter0     = {r}
    $parameter1     = {ie}

    downStar        = {r->r, ie->ie}
    r[$parameter0]  = {r}
    r[$parameter1]  = {ie}

    kge             = {r->ie:1/2, ie->r:1/2, ie->ie, r->r:1/2}

    storeProp       = {r, ie}

    p2_5 = {r,ie}

    kge[$parameter1, left]  = {r:1/2, ie:1/2}
    kge[$parameter1, right] = {r:1/2, ie:1/2}

    greRelation = 1
    treeness    = 1
    inOrder     = 1
}

%n = {r, ts, ie}
%p = {

    heapcell        = {r, ts, ie}

    sm              = {ts:1/2}

    $parameter0     = {r}
    $parameter1     = {ie}

    down            = {r->ts:1/2, ts->ts:1/2}
    downStar        = {r->r, ie->ie, ts->ts:1/2, r->ts}

    r[$parameter0]  = {r, ts}
    r[$parameter1]  = {ie}

    ie1             = {r->ts:1/2, ts->ts:1/2}
    ie2             = {r->ts:1/2, ts->ts:1/2}
    ie3             = {r->ts:1/2, ts->ts:1/2}

    left             = {r->ts:1/2, ts->ts:1/2}
    cleft            = {r->ts:1/2, ts->ts:1/2}
    cright           = {r->ts:1/2, ts->ts:1/2}
    right            = {r->ts:1/2, ts->ts:1/2}

    kge              = { r->r:1/2, ie->ie, ts->ts:1/2,
                         r->ie:1/2, ie->r:1/2,
                         r->ts:1/2, ts->r:1/2,
                         ie->ts:1/2, ts->ie:1/2 }
    isStore          = {ts:1/2}
    storeProp        = {r, ts, ie}

    leftOf                      = {ts->ts:1/2, r->ts:1/2, ts->r:1/2, r->r:1/2}
    leftOfStar                  = { ts->ts:1/2, r->ts:1/2,
                                    ts->r:1/2, r->r:1/2, ie->ie:1/2}
    rightSubTree                = {ts->ts:1/2, r->ts:1/2, ts->r:1/2, r->r:1/2}

    rightOf                     = {ts->ts:1/2, r->ts:1/2, ts->r:1/2, r->r:1/2}
    rightOfStar                 = { ts->ts:1/2, r->ts:1/2,
                                    ts->r:1/2, r->r:1/2, ie->ie:1/2}
    leftSubTree                 = {ts->ts:1/2, r->ts:1/2, ts->r:1/2, r->r:1/2}

    p2_5                        = {r,ts,ie}

    kge[$parameter1, left]      = {r:1/2, ts:1/2, ie:1/2}
    kge[$parameter1, right]     = {r:1/2, ts:1/2, ie:1/2}

    isElement   = {ie->r:1/2, ie->ts:1/2, ts->r:1/2, ts->ts:1/2}

    greRelation = 1
    inOrder     = 1
    treeness    = 1
}
```

Input-Structures for Insert (Partial)

```
%n = {r,ie}
%p = {
    heapcell = {r, ie}
    $parameter0     = {r}
    $parameter1     = {ie}

    downStar        = {r->r, ie->ie}
    r[$parameter0]  = {r}
    r[$parameter1]  = {ie}

    kge             = { r->ie:1/2, ie->r:1/2, ie->ie, r->r:1/2}

    storeProp       = {r, ie}

    p2_5 = {r, ie}

    kge[$parameter1, left]  = {r:1/2, ie:1/2}
    kge[$parameter1, right] = {r:1/2, ie:1/2}

    greRelation = 1
    treeness    = 1
    inOrder     = 1
    level = {r->r, ie->ie}
    p1=1
}
///////////
%n = {r, ts, ie}
%p = {

    heapcell        = {r, ts, ie}

    sm              = {ts:1/2}

    $parameter0     = {r}
    $parameter1     = {ie}

    down            = {r->ts:1/2}
    downStar        = {r->r, ie->ie, ts->ts:1/2, r->ts}

    r[$parameter0]  = {r, ts}
    r[$parameter1]  = {ie}

    ie1             = {r->ts:1/2}
    ie2             = {r->ts:1/2}
    ie3             = {r->ts:1/2}

    kge              = { r->r:1/2, ie->ie, ts->ts:1/2,
                         r->ie:1/2, ie->r:1/2,
                         r->ts:1/2, ts->r:1/2,
                         ie->ts:1/2, ts->ie:1/2 }

    isStore          = {ts:1/2}
    storeProp        = {r, ts, ie}

    leftOf                      = {ts->ts:1/2, r->ts:1/2, ts->r:1/2, r->r:1/2}
    leftOfStar                  = { ts->ts:1/2, r->ts:1/2,
                                    ts->r:1/2, r->r:1/2, ie->ie:1/2}
    rightSubTree                = {ts->ts:1/2, r->ts:1/2, ts->r:1/2, r->r:1/2}

    rightOf                     = {ts->ts:1/2, r->ts:1/2, ts->r:1/2, r->r:1/2}
    rightOfStar                 = { ts->ts:1/2, r->ts:1/2,
                                    ts->r:1/2, r->r:1/2, ie->ie:1/2}
    leftSubTree                 = {ts->ts:1/2, r->ts:1/2, ts->r:1/2, r->r:1/2}

    p2_5                        = {r,ts,ie}

    kge[$parameter1, left]      = {r:1/2, ts:1/2, ie:1/2}
    kge[$parameter1, right]     = {r:1/2, ts:1/2, ie:1/2}

    isElement   = {ie->r:1/2, ie->ts:1/2, ts->r:1/2, ts->ts:1/2}

    greRelation = 1
    inOrder     = 1
    treeness    = 1

    level = {r->r, ie->ie, ts->ts:1/2}
    p1 = 1
}
```

B.2 B-Tree Implementations

B.2.1 2-3-4 Tree Implementation

Structures

The Node Class

```
public class Node234 {
  public Node234 left;
  public Node234 cleft;
  public Node234 cright;
  public Node234 right;

  public IndexElement ie1;
  public IndexElement ie2;
  public IndexElement ie3;
}
```

The Index Element Class

```
public class IndexElement {
  // This object's key.
  public int key;
  // This objects data.
  public Object content = null;
}
```

Algorithms

The contains-Method Note: The enclosed CD-ROM contains an additional implementation of the contains-operation.

```
  /**
   * Scans the 2-3-4 tree rooted at <i>node</i> for index element <i>ie</i>.
   * <p/>
   * If an index element whose key equals that of <i>ie</i> is found,
   * the node at which this element is stored is returned,
   * otherwise <code>null</code> is returned.
   *
   * @param  node  root node at which the search starts
   * @param  ie    index element for which to scan the tree
   *
   * @return  the <code>Node234</code> object referencing <i>ie</i>
   *      or <code>null</code>
   */
  public static Node234 contains( Node234 node, IndexElement ie ) {
    Node234 result = null;
    if( node.ie1 == null )
      node = null;
    while( node != null ) {
      if( node.ie3 != null && node.ie3.key <= ie.key) {
        if( node.ie3.key == ie.key){
          result = node;
          node = null;
        }
        else
          node = node.right;
      }
      else if( node.ie2 != null && node.ie2.key <= ie.key ) {
        if( node.ie2.key == ie.key){
          result = node;
          node = null;
        }
        else
          node = node.cright;
      }
      else if( node.ie1 != null && node.ie1.key <= ie.key ) {
        if( node.ie1.key == ie.key){
          result = node;
          node = null;
        }
        else
          node = node.cleft;
     }
      else if( node.ie1 != null && node.ie1.key > ie.key )
        node = node.left;
      else
        node = null;
    }
return result;
  }
```

The insert-Method (Iterative)

```
static public Node234 insert( Node234 root, IndexElement ie ) {
if( root.ie3 != null ) {
// save reference to old root
Node234 child = root;
// create new root with old root as leftmost child
root = new Node234();
Node234 z = new Node234();

// collect
IndexElement iel1, iel2, iel3;
Node234 left1, left2, cleft1, cleft2;
```

```
iel1 = child.ie1;
iel2 = child.ie2;
iel3 = child.ie3;
left1 = child.left;
cleft1 = child.cleft;
left2 = child.cright;
cleft2 = child.right;
// spill
child.ie3 = null;
child.ie2 = null;
child.ie1 = null;
child.left = null;
child.cleft = null;
child.cright = null;
child.right = null;
root.ie1 = iel2;
child.ie1 = iel1;
z.ie1 = iel3;

root.left = child;
root.cleft = z;

child.left = left1;
child.cleft = cleft1;
z.left = left2;
z.cleft = cleft2;
}
Node234 node = root;
Node234 child = null;
while( node.left != null ) {
// find correct child to insert the index element into
if( node.ie3 != null && node.ie3.key < ie.key)
child = node.right;
else if( node.ie2 != null && node.ie2.key < ie.key )
child = node.cright;
else if( node.ie1 != null && node.ie1.key < ie.key )
child= node.cleft;
else
child = node.left;
// if child is already full, split it first
if( child.ie3 != null ) {
Node234 parent = node;
Node234 z = new Node234();

z.ie1 = child.ie3;
if( child.left != null ) {
z.left = child.cright;
z.cleft = child.right;
child.cright = null;
child.right  = null;
}
```

```
// add new child and new index element to parent
if( child == parent.left ) { // node at left was split
parent.right = parent.cright;
parent.cright = null;
parent.cright = parent.cleft;
parent.cleft = null;
parent.cleft = z;
parent.ie3 = parent.ie2;
parent.ie2 = null;
parent.ie2 = parent.ie1;
parent.ie1 = null;
parent.ie1 = child.ie2;
}
else if( child == parent.cleft ) { // node at cleft was split
parent.right = parent.cright;
parent.cright = null;
parent.cright = z;
parent.ie3 = parent.ie2;
parent.ie2 = null;
parent.ie2 = child.ie2;
}
else if( child == parent.cleft ) { // node at cright was split
parent.right = z;
parent.ie3 = child.ie2;
}

child.ie2 = null;
child.ie3 = null;

// refind correct child to insert the index element into
if( node.ie3 != null && node.ie3.key < ie.key)
child = node.right;
else if( node.ie2 != null && node.ie2.key < ie.key )
child = node.cright;
else if( node.ie1 != null && node.ie1.key < ie.key )
child= node.cleft;
else
child = node.left;
}
//
node = child;
}
// insert into non-full leaf
IndexElement a,b,c;
a = node.ie1; b = node.ie2; c = node.ie3;
// collect
if( node.ie2 != null ) {
if( node.ie2.key < ie.key  ) {
c = ie;
ie = null;
```

```
}
else
c = node.ie2;
}
if( node.ie1 != null && ie != null ) {
if( node.ie1.key < ie.key  ) {
b = ie;
ie = null;
}
else
b = node.ie1;
}
if( ie != null ) {
a = ie;
}
// spill
node.ie3 = null;
node.ie2 = null;
node.ie1 = null;
node.ie1 = a;
node.ie2 = b;
node.ie3 = c;

return root;
}
```

The insert-Method (Interprocedural)

```
  static public void split( Node234 parent, Node234 child, int index ) {
    Node234 z = new Node234();
    z.iesStored = 1;

    z.ie1 = child.ie3;
    if( child.left != null ) {
      z.left  = child.cright;
      z.cleft  = child.right;
    }
    child.iesStored = 1;

    // add new child to parent
    int i;
    for( i = parent.iesStored + 1; i >= index + 1; i-- ) {
      if( i == 3 )
        parent.right = parent.cright;
      else if( i == 2 )
        parent.cright = parent.cleft;
      else if( i == 1 )
        parent.cleft = parent.left;
    }
    if( index == 3 )
      parent.right = z;
    else if( index == 2 )
```

```
      parent.cright = z;
    else if ( index == 1 )
      parent.cleft = z;
    // add new index element to parent
    for( i = parent.iesStored; i >= index; i-- ) {
      if( i == 2 )
        parent.ie3 = parent.ie2;
      else if( i == 1 )
        parent.ie2 = parent.ie1;
    }

    if( index == 3 )
      parent.ie3 = child.ie2;
    else if( index == 2 )
      parent.ie2 = child.ie2;
    else if ( index == 1 )
      parent.ie1 = child.ie2;

    parent.iesStored++;
    child.ie2 = null;
    child.ie3 = null;
  }

  static public void insertNonFull( Node234 node, IndexElement ie ) {
    int i = node.iesStored;
    // non-full leaf -> just insert ie
    if( node.left == null ) {
      while( i >= 1 ) {
        if( i == 2 ) {
          if( node.ie2.key > ie.key )
            node.ie3 = node.ie2;
          else
            break;
        }
        if( i == 1 ) {
          if( node.ie1.key > ie.key)
            node.ie2 = node.ie1;
          else
            break;
        }
        i--;
      }
      i++;
      // insert key
      if( i == 3 )
        node.ie3 = ie;
      else if( i == 2 )
        node.ie2 = ie;
      else if( i == 1 )
        node.ie1 = ie;
      node.iesStored++;
    }
    else {
      // find correct child of node to insert the index element
      while( i >= 1 ) {
        if( i == 3 ) {
          if( node.ie3.key <= ie.key )
            break;
        }
        else if( i == 2 ) {
          if( node.ie2.key <= ie.key )
            break;
        }
        else if( i == 1 ) {
          if( node.ie1.key <= ie.key )
            break;
        }
        i--;
      }
      i++;
      Node234 child = null;
      if( i == 4 )
        child = node.right;
      else if( i == 3 )
        child = node.cright;
      else if( i == 2 )
        child = node.cleft;
      else if( i == 1 )
        child = node.left;
      // if child is already full, split it first
      if( child.iesStored == 3 ) {
        split( node, child, i );
        if( i == 3 && node.ie3.key < ie.key )
          i++;
        else if( i == 2 && node.ie2.key < ie.key )
          i++;
        else if( i == 1 && node.ie1.key < ie.key )
          i++;
      }
      if( i == 4 )
        child = node.right;
      else if( i == 3 )
        child = node.cright;
      else if( i == 2 )
        child = node.cleft;
      else if( i == 1 )
        child = node.left;
      insertNonFull( child, ie );
    }
  }

  static public Node234 insert( Node234 root, IndexElement ie ) {
    if( root.iesStored == 3 ) {
      // save reference to old root
      Node234 r = root;
      // create new root with old root as leftmost child
      root = new Node234();
      //root.isLeaf = false;
      root.iesStored = 0;
      root.left = r;
      split( root, r, 1 );
    }
    insertNonFull( root, ie );
    return root;
  }
```

B.2.2 B-Tree Implementation

Structures

The Node Class

```
public class Node {
  // order of B-trees this node fits into
  public int t;

  // boolean, indicating whether this is a leaf node
  public boolean isLeaf;

  // number of index elements stored at this node
  public int iesStored = 0;

  // array of child pointers
  public Node[] childs;
  // array of stored values
  public IndexElement[] elements;

  /**
   * Constructor.<p/>
   *
   * Constructs a new <code>Node</code>-element fitting
   * into B-trees of order <i>t</i>.
   *
   * @param  t  integer value indicating the order of the
   *        B-tree for which this node object is created
   */
  public Node( int t ) {
    this.t = t;
    childs = new Node[ 2*t ];
    elements = new IndexElement[2*t - 1];
  }
}
```

Algorithms

The contains-Method

```
  /**
   * Scans the B-tree rooted at <i>n</i> for index element <i>ie</i>.
   * <p/>
   * If an index element whose key equals that of <i>ie</i> is found,
   * the node at which this element is stored is returned,
   * otherwise <code>null</code> is returned.
   *
   * @param  node  root node at which the search starts
   * @param  ie    index element for which to scan the tree
   *
   * @return  the <code>Node</code> object referencing <i>ie</i>
   *      or <code>null</code>
   */
  public static Node contains( Node n, IndexElement ie ) {
    int i = 0;
    while( i < n.iesStored && ie.key > n.elements[i].key )
      i++;
    if( i < n.iesStored && ie.key == n.elements[i].key )
      return n;
    if( n.isLeaf )
      return null;
    else
      return contains( n.childs[i], ie );
  }
```

The insert-Method

```
public static Node insert( Node tree, IndexElement ie ) {
if( tree.iesStored == T * 2 - 1 ) {
Node n = tree;
Node s = new Node( T );
tree = s;
s.isLeaf = false;
s.iesStored = 0;
s.childs[0] = n;
split( s, 1, n );
insertNonFull( s, ie );
return tree;
}
else {
insertNonFull( tree, ie );
return tree;
}
}

public static void split( Node node, int i, Node ithChild ) {
Node z = new Node( T );
z.isLeaf = ithChild.isLeaf;
z.iesStored = T - 1;
int j;
for( j = 0; j < T - 1; j++ )
z.elements[j] = ithChild.elements[j+T];
if( !ithChild.isLeaf ) {
for( j = 0; j < T; j++ )
z.childs[j] = ithChild.childs[j+T];
}
ithChild.iesStored = T - 1;
for( j = node.iesStored; j >= i; j-- )
node.childs[j+1] = node.childs[j];
node.childs[i] = z;
for( j = node.iesStored - 1; j >= i - 1; j-- )
node.elements[j+1] = node.elements[j];
node.elements[i-1] = ithChild.elements[T-1];
node.iesStored++;
for( j = T - 1; j < 2 * T - 1; j++ )
ithChild.elements[j] = null;
}

public static void insertNonFull( Node n, IndexElement ie ) {
int i = n.iesStored - 1;
if( n.isLeaf ) {
while( i >= 0 && ie.key < n.elements[i].key ) {
n.elements[ i + 1 ] = n.elements[ i ];
i--;
}
n.elements[i+1] = ie;
n.iesStored++;
}
else {
while( i >= 0 && ie.key < n.elements[i].key )
i--;
i++;
if( n.childs[i].iesStored == 2*T-1 ) {
split( n, i+1, n.childs[i] );
if( ie.key > n.elements[i].key )
i++;
}
insertNonFull( n.childs[i], ie );
}
}
```

B.3 J2TVLA Classes and Files

btreeTemplates.properties

```
################################################################################
# Modified version of defaultTemplates.properties to cope with B-Trees.
#
```

```
# author    Joerg Herter
# date      2008/01/01
#
# NOTE
# ----
# It is assumed that the action "Skip()" is defined somewhere.
# This is used to ensure that the CFG produced by the translation is a connected graph.
################################################################################


        ########################################################################
        # Templates for assignment (and more generally, definition) statements #
        ########################################################################
# Assignments of the form x = x OP CONST, where OP is an integer operator and
# CONST some integer constant
AssignStmtInt = \n// line %LINE : %JIMPLE (AssignStmt)\n\
                %LBEGIN %OP(%RHS_VAR); AssignRefToRef(%LHS, %RHS_VAR) %LEND


# The default template of an assignment statement.
AssignStmt = \n// line %LINE : %JIMPLE (AssignStmt)\n\
         %LBEGIN AssignRefToRef(%LHS, %RHS_BASE) %LEND

# The default template of an assignment statement.
AssignNewClassStmt = \n// line %LINE : %JIMPLE (AssignNewClassStmt)\n\
                 %LBEGIN new(%LHS) %LEND


# A template for statements of the form : x = y
# where x and y are local reference variables.
AssignRefToRefStmt = \n// line %LINE : %JIMPLE (AssignRefToRefStmt)\n\
             %LBEGIN AssignRefToRef(%LHS, %RHS) %LEND


# A template for statements of the form : x = null
# where x is a local reference variable.
AssignNullToLocalStmt = \n// line %LINE : %JIMPLE (AssignNullToLocalStmt)\n\
                %LBEGIN SetNull(%LHS) %LEND

# A template for statements of the form : x.n = y
# where y is a reference variable and n is a reference field.
AssignRefToInstanceFieldRefStmt = \n// line %LINE : %JIMPLE (AssignRefToInstanceFieldRefStmt)\n\
                      %LBEGIN AssignRefTo%LHS_FIELDStmt(%LHS_BASE, %RHS) %LEND

# %LBEGIN AssignNullToInstanceFieldRefStmt(%LHS_BASE, %LHS_FIELD);
# AssignRefToInstanceFieldRefStmt(%LHS_BASE, %LHS_FIELD, %RHS) %LEND



# A template for statements of the form : x.n = null
# where n is a reference field.
```

```
AssignNullToInstanceFieldRefStmt = \n// line %LINE : %JIMPLE (AssignNullToInstanceFieldRefStmt)\n\
                       %LBEGIN AssignNullToInstanceFieldRefStmt(%LHS_BASE, %LHS_FIELD) %LEND

# A template for statements of the form : x = y.n
# where x is a reference variable and n is a reference field.
AssignInstanceFieldRefToRefStmt = \n// line %LINE : %JIMPLE (AssignInstanceFieldRefToRefStmt)\n\
                      %LBEGIN AssignFieldRefToRef(%LHS, %RHS_BASE, %RHS_FIELD) %LEND

        ########################################
        # Templates for conditional statements #
        ########################################

# This statement is handled conservatively, by adding transitions to both targets
# (the fall though target and the one intended by the condition).
IfStmt = \n// line %LINE : %JIMPLE (IfStmt)\n\
     %LBEGIN %PRED_TRUE(%LHS, %RHS) %LTRUE\n\
     %LBEGIN %PRED_FALSE(%LHS, %RHS) %LFALSE

# A template for statements of the form : if (x == null).
# The macro %LHS corresponds to x.
IfEqualToNullStmt = \n// line %LINE : %JIMPLE (IfEqualToNullStmt)\n\
                %LBEGIN IsNullVar(%LHS) %LTRUE\n\
                %LBEGIN IsNotNullVar(%LHS) %LFALSE

# A template for statements of the form : if (b).
# The macro %LHS corresponds to b.
IfBoolStmt = \n// line %LINE : %JIMPLE (IfBoolStmt)\n\
         %LBEGIN IsTrue(%LHS) %LTRUE\n\
         %LBEGIN IsFalse(%LHS) %LFALSE

# A template for statements of the form : if (b1 == b2).
# The macro %LHS corresponds to b1 and the macro %RHS corresponds to b2.
IfEqualBoolStmt = \n// line %LINE : %JIMPLE (IfEqualBoolStmt)\n\
              %LBEGIN IsEqualBool(%LHS, %RHS) %LTRUE\n\
              %LBEGIN IsNotEqualBool(%LHS, %RHS) %LFALSE

# A template for statements of the form : if (x == y)
# where both x and y are reference variables.
# The macro %LHS corresponds to b1 and the macro %RHS corresponds to b2.
IfEqualRefStmt = \n// line %LINE : %JIMPLE (IfEqualRefStmt)\n\
              %LBEGIN IsEqualRef(%LHS, %RHS) %LTRUE\n\
              %LBEGIN IsNotEqualRef(%LHS, %RHS) %LFALSE

# A template for statements of the form : return v.
ReturnStmt = \n// line %LINE : %JIMPLE (ReturnStmt)\n\
         %LBEGIN AssignRefToRef(return, %RETURN_ACTUAL) %%LCOMMON\n\
         %MULTIPLE{%%LCOMMON Skip() %LTARGET\n}
```

BTreeTranslator.java and Main.java

The contents of these files can be found on the enclosed CD.

C Contents of the Enclosed CD-ROM

The enclosed CD-ROM contains all files, programs, tools, and documents to reproduce the analyses described in this thesis. The following pages list the contents of the CD-ROM.

./towards_shape_analysis_of_btrees.pdf This document as a PDF file.

./analyses This folder contains all TVLA input files to redo our analyses.

./analyses/contains(1) This folder contains all TVLA input files to redo our shape analysis of the first implementation of a contains-operation on 2-3-4 trees.

./analyses/contains(2) This folder contains all TVLA input files to redo our shape analysis of the second, more intuitive implementation of a contains-operation on 2-3-4 trees.

./analyses/insert This folder contains all TVLA input files to redo our shape analyses of the insert operation.

./analyses/excursus This folder contains all TVLA input files to redo the shape analysis described in the excursus about integer arithmetics.

./java All Java sources created in the course of this work are residing in this folder.

./java/mytranslators Java package containing our modified translator classes used with J2TVLA.

./java/tree_implementations Folder containing our Java implementations of various operations on 2-3-4 trees and general B-trees.

./java/integer_arithmetics/ This folder contains the Java method we analyzed in the excursus about integer arithmetics.

./software This folder contains software and tools we used.

./software/tvla TVLA Framework version 3.0-alpha.

./software/j2tvla The modified version of the J2TVLA Framework we used.