Automating Abstraction Functions

by

Derek F. Rayside

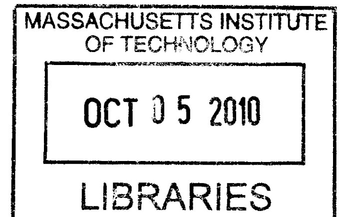
Submitted to the Department of
Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2010



© Derek F. Rayside, MMX. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part.

Author ....
Department of
Electrical Engineering and Computer Science
August 31, 2010

Certified by ....
Daniel Jackson
Professor
Thesis Supervisor

Accepted by ....
Professor Terry P. Orlando
Chairman, Department Committee on Graduate Students

Automating Abstraction Functions

by

Derek F. Rayside

Submitted to the Department of
Electrical Engineering and Computer Science
on August 31, 2010, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Electrical Engineering and Computer Science

Abstract

Data abstraction has been the dominant structuring paradigm for programs for decades. The essence of a data abstraction is the abstraction function, which relates the concrete program representation to its abstract meaning. However, abstraction functions are not generally considered to be a part of the executing program. We propose that making abstraction functions an executable part of the program can enable programmers to write clearer and more concise programs with fewer errors.

In particular, we show that the object equality and hashing operations (which programmers are required to write), can often be expressed more clearly and more concisely in terms of the abstract state of the object. Getting these methods right has proven to be difficult for programmers at all skill levels, from novice through expert. In a case study of the standard Java libraries we show that rewriting the code with explicit declarative abstraction functions (and generating equality and hashing methods automatically) removed object-contract compliance faults previously found by Pacheco et al.

To make abstraction functions part of the executing program we develop four techniques for the dynamic evaluation of abstraction functions written in a declarative first-order logic with relations and transitive closure. We observe that the abstraction functions programmers write in practice may often be viewed as navigation queries on the heap, and two of our techniques exploit this insight to synthesize executable code from declarative abstraction functions. The performance of our research prototype is within striking distance of hand-written code.

Thesis Supervisor: Daniel Jackson
Title: Professor

Acknowledgements

I have had the good fortune to be surrounded by my betters for the last seven years; an opportunity for which I am very grateful. A great many people have participated in discussions of this work, in discussions of other projects, and in my intellectual development while at MIT. The process of writing a document such as this consumes a certain amount of one's energy: so, while I would like to write a small poem of appreciation for each and every person mentioned here (and a few whom I have surely inadvertently omitted), such emotional flare is beyond my current metabolic reserves.

My advisor, Prof Daniel Jackson, deserves accolades for his patience and good humour in putting up with me for so long. During my time at MIT his intellect has been widely acknowledged by external sources; I hereby confirm from the inside that it is all true. He is an exemplary role model as a researcher, as a teacher, and as a family man. If I can be half as accomplished as him in all this I will have a very good life indeed.

Prof Martin Rinard has been a challenging and engaging intellectual sparring partner since I arrived at MIT. He is fiercely devoted to his students and always pushes them to excel.

Prof Steve Ward has been more generous with his time than I deserve and has also asked me questions that I should know the answers to.

The structure of this section follows the acknowledgements section of Sarfraz Khurshid's dissertation [57]. I particularly like that he had a co-authors paragraph, in recognition of the fact that a work such as this is always accomplished within a social setting and not in a mythical garret of the ivory tower. In this instance the people listed as 'co-authors' are indeed co-authors of papers based on this dissertation. Other people with whom I have written papers not related to this dissertation are included elsewhere in this section. Within each paragraph individuals are listed in alphabetical order by surname.

Co-authors: Zev Benjamin, Aleksander Milicivec, Joseph P. Near, Rishabh Singh, Kuat Yessenov.

Haskell Tutors: Alec Heller, Joe Near, Derek Thurn, Jean Yang.

Prof Daniel Jackson's group: Zev Benjamin, Felix Chang, Greg Dennis, Jonathan Edwards, Eunsuk Kang, Sarfraz Khurshid, Lucy Mendel, Aleksander Milicivec, Joseph P. Near, Mana Taghdiri, Robert Seater, Ilya Shlyakhter, Emina Torlak, Mandana Vaziri, Kuat Yessenov.

Prof Michael Ernst's group: Shay Artzi, Danny Dig, Adam Donovan, Stephen J. Garland, David Glasser, Philip Guo, Adam Kiezun, Sung Kim, Carlos Pacheco, Matt Papi, Jeff Perkins, Stephen McCamant, Toh Ne Win, David Saff, Matthew Tschantz, Amy Williams, Chen Xiao, Yoav Zibin.

Prof Martin Rinard's group: Scott Ananian, Chandra Boyapati, Michael Carbin, Vijay Ganesh, Maria Cristina Marinescu, Darko Marinov, Patrick Lam, Alexandru Salcianu, Stelios Sidiroglou, Karen Zee.

Prof Robert Miller's group: Michael Bolin, Lydia Chilton, Simson Garfinkel, Max Goldman, Ryan Jazayeri, Greg Little, Vineet Sinha, Chen-Hsiang (Jones) Yu.

Prof Steve Ward's group: Justin Paluska, Hubert Pham.

Prof Armando Solar-Lezama's group: Rishabh Singh, Jean Yang.

Prof Charles Leiserson's group: Angelina Lee.

Prof Anant Agarwal's group: Michael B. Taylor.

Visitors: Christian Estler, Sol Greenspan, Carroll Morgan, Tahina Ramananandro.

Support staff: Maria Brennan, Cree Bruins, Henry Gonzalez, Mary McDavitt, Maria Rebelo, Tyrone Sealy, Ron Wiken, Anthony Zolnik.

Media Lab: Leonardo Bonanni, Ted Selker.

Engineering Systems Division: Ed Crawley, Ben Koo, Willard Simmons, Olivier de Weck.

Northeastern: Alec Heller, Felix Klock, Viera Proulx.

On the Origins of Specification Fields: Gary Leavens, Rustan Leino, and Carroll Morgan were generous with their time and experience in my efforts to track down the origins of specification fields. Gary Leavens scanned Jones [55] for me.

Dedicated To:
JWR, HSL, ISL, & SWF
for teaching me a few technical things when I was but a wee lad
&
MELR
for common sense, good humour, and steadfast support
&
Stephanie & Colin
for safety and adventure

Contents

1 Introduction **10**
- 1.1 Specifications 10
  - 1.1.1 Specification Fields 13
  - 1.1.2 The JForge Specification Language 14
  - 1.1.3 Extensions to Abstraction Functions 14
- 1.2 Ongoing Example: Binary Search Tree of Integers 16
- 1.3 Executable Specifications 19
- 1.4 The Object Contract 20
- 1.5 Thesis Statement 21
- 1.6 Contributions 21
- 1.7 Architectural Overview of Prototype Implementation 22
- 1.8 Summary 24

2 Expressing Abstraction Functions **25**
- 2.1 Choosing a Specification Language 26
- 2.2 JForge Specification Language Overview 28
  - 2.2.1 Specification Fields and Abstraction Functions 29
- 2.3 Traversal Predicates 31
  - 2.3.1 Tree-Traversal Predicate Definitions 34
  - 2.3.2 Tree Traversal Predicates Are Functional 37
  - 2.3.3 Tree Traversal Predicates Differ 39

3 Executing Abstraction Functions **42**
- 3.1 Navigable Expressions 43
- 3.2 Syntax-Directed Execution of Abstraction Functions 45
- 3.3 Conservative Navigation 46
- 3.4 Two Navigations With Sequences 49
- 3.5 Streamlined Navigation 50
- 3.6 Optimization Condition Generation 52
- 3.7 Soundness of Optimization Condition Generation 54
- 3.8 Predicate Substitution 56
- 3.9 Constraint Solving 57
- 3.10 Performance Characterization 59
- 3.11 Annotating the JDK Collections v1.4 63
  - 3.11.1 Interesting Abstraction Functions from the JDK Collections 65

4 Equality **66**
4.1 A Simple Example . . . 67
4.2 Object-Contract Compliance is Difficult . . . 68
4.2.1 Difficulty: Simple Errors . . . 68
4.2.2 Difficulty: Subclassing . . . 68
4.2.3 Difficulty: Delegation . . . 70
4.2.4 Difficulty: Objects of Different Classes . . . 72
4.2.5 Difficulty: Cyclic Object Graphs . . . 72
4.3 Object Equality in Terms of Abstract State . . . 72
4.3.1 Cyclic References . . . 73
4.3.2 A Syntactic Shorthand: @ConcreteEquality . . . 74
4.3.3 Equality Types . . . 74
4.3.4 Inferring Equality Types . . . 75
4.4 Empirical Evaluation . . . 77
4.4.1 Expressiveness . . . 77
4.4.2 Correctness . . . 78
4.5 Related Work . . . 82
4.6 Object Equality and Observational Equivalence . . . 83

5 Hashing **85**
5.1 Design Objectives . . . 86
5.1.1 Design Objective: Generality . . . 86
5.1.2 Design Objective: Correctness . . . 86
5.1.3 Design Objective: Performance . . . 88
5.2 Design Space . . . 89
5.3 Object Hashing Strategies . . . 90
5.3.1 Constant Value . . . 91
5.3.2 Finite Unrolling . . . 91
5.3.3 Constant When Cyclic . . . 92
5.3.4 Ignore Cycles . . . 92
5.3.5 Canonical Set . . . 93
5.3.6 Canonical Tree . . . 94
5.4 Evaluation . . . 94
5.4.1 Micro-Benchmarks . . . 94
5.4.2 Strategy Evaluation in The Wild . . . 98
5.4.3 The Worst Possible Strategy . . . 100
5.5 Conclusions . . . 101

6 Conclusions **103**
6.1 Zave Evaluation Criteria . . . 104
6.2 The Programmer as Relational Navigator . . . 106
6.3 Future Directions . . . 107

List of Figures

1-1 Algebraic specification of a set of integers by Zilles [48] 11
1-2 Model-based specification of a set of integers by Hoare [46] 11
1-3 A set of integers implemented with an array (Hoare [46]) 12
1-4 Abstraction function for a set of integers represented by an array (Hoare [46]) 12
1-5 Commutative diagram describing data refinement 13
1-6 Model-based specification of a set of integers (written in JFSL) 16
1-7 JForge abstraction function for binary search tree 16
1-8 Binary Search Tree example with manual implementation of object contract methods 18
1-9 The Object Contract as an algebraic specification 20
1-10 Architectural overview of prototype implementation 23

2-1 JML abstraction function for binary search tree (adapted from [23]) 27
2-2 Jahob abstraction function for binary search tree (adapted from [124]) . . . 27
2-3 JForge abstraction function for binary search tree (adapted from Figure 1-7) 27
2-4 Summary of JForge Specification Language (JFSL) annotations 28
2-5 Abstraction function for SingletonSet 29
2-6 Abstraction function for ConsCell 30
2-7 Abstraction function for LinkedList 30
2-8 Abstraction function for ArrayList 30
2-9 Various abstraction functions for ongoing binary search tree example 31
2-10 Abstraction function for EmptyConsCell 31
2-11 Traversal predicates 32
2-12 Linked-list structure 33
2-13 Naïve forwards traversal of linked-list 33
2-14 Forwards traversal of a potentially cyclic linked-list 33
2-15 Standard recursive definitions of pre-order, post-order, in-order, and level-order traversals of binary trees [101] 34
2-16 Declarative definition of in-order traversal of a binary tree 35
2-17 Declarative definition of post-order traversal of an n-ary tree 36
2-18 Declarative definition of pre-order traversal of an n-ary tree 36
2-19 Declarative definition of level-order traversal of an n-ary tree 37
2-20 Theorem: definition of in-order traversal is functional 38
2-21 Theorem: definitions of pre-order and post-order traversals differ 39

2-22 Example trees for which pre-order and level-order traversals are equivalent. Nodes are numbered in traversal order. 40
2-23 Example trees for which pre-order and level-order traversals differ. Nodes for which the orders differ are circled. 40
2-24 Theorem: definitions of pre-order and level-order traversals differ 41

3-1 Pareto Front of abstraction function evaluation strategies: generality of expression versus runtime speed 43
3-2 Abstract syntax of navigable expressions (in Haskell) 44
3-3 Denotation of this.root.*(left+right) on simple example tree data structure. . . 45
3-4 Sequence chart of evaluation of example α on example data structure. . . . 46
3-5 Haskell description of conservative navigation (type definitions) 47
3-6 Haskell description of conservative navigation (unary productions) 47
3-7 Haskell description of conservative navigation (helper functions) 48
3-8 Haskell description of conservative navigation (binary productions) 48
3-9 The set of elements in a sequence 49
3-10 Joining a sequence to a field 49
3-11 Haskell description of streamlined navigation (type definitions) 50
3-12 Haskell description of streamlined navigation (unary productions) 51
3-13 Haskell description of streamlined navigation (helper functions) 51
3-14 Haskell description of streamlined navigation (binary productions) 51
3-15 Optimization condition generation for optimized navigation 53
3-16 Multiple navigation entry points into a tree data structure 53
3-17 Theorem: the optimization conditions imply that the computation is equal to the denotation for the direct model of streamlined navigation 55
3-18 Theorem: the optimization conditions imply that the computation is equal to the denotation for the indirect model of streamlined navigation 55
3-19 Theorem: direct and indirect models of streamlined navigation are equivalent 56
3-20 Ongoing example: use of inorder predicate to define nodeseq specification field 56
3-21 Iterative definition of in-order traversal predicate [115] 57
3-22 Example abstraction function that requires constraint solving (ArrayPointSet) 58
3-23 Example abstraction function for which constraint solving would not produce the expected result (RangePointSet) 59
3-24 Traversal times for binary search tree abstraction function this.root.*(left+right) 61
3-25 Count of interesting classes in JDK Collections v1.4 63
3-26 Abstraction functions for java.util.TreeMap 65
3-27 Abstraction functions for java.util.HashMap 65

4-1 Example of two equal string objects 67
4-2 Simple Cartesian Point class and equals method (Bloch [17]) 68
4-3 ColourPoint1 extends Point, violates symmetry (Bloch [17]) 69
4-4 ColourPoint2 extends Point, violates transitivity (Bloch [17]) 70

4-5 A simple wrapper class and its equals implementation 70
4-6 Test revealing reflexivity fault in common wrapper implementation of equals 71
4-7 Test revealing symmetry fault in common wrapper implementation of equals 71
4-8 Test revealing non-termination in common wrapper implementation of equals 71
4-9 Concrete and abstract cyclic references 73
4-10 Declaration of equalityType specification field 74
4-11 Illustrations of equality type inference 76
4-12 Number of manually written equals and hashCode implementations in original program, and remaining after code was converted to use our technique. 77
4-13 Test results 78
4-14 Classification of faults found (and fixed) 81

5-1 Equivalent cyclic object structures and their unrollings 87
5-2 A fully connected object structure and its unrolling 89
5-3 Notation for hash strategy listings 91
5-4 Finite Unrolling Hash Strategy 91
5-5 Constant When Cyclic Hash Strategy 92
5-6 Ignore Cycles Hash Strategy 93
5-7 Canonical Set Hash Strategy 93
5-8 Canonical Tree Hash Strategy 94
5-9 Example Heat Map Plot 96
5-10 Heat maps plots of micro-benchmark experiments 97
5-11 Performance evaluation 99
5-12 Programmer-written hashing methods versus Finite Unrolling (depth 1) comparison in DaCapo benchmarks 99
5-13 Performance of returning a constant (the "worst possible hash function") . 101

Chapter 1

Introduction

This dissertation is a small part of a broader vision, shared by many researchers over many years, of executable specifications. The goal of such research is to produce programs that are more correct with less programmer effort. Such techniques may be applicable in domains where the specifications are clearer and more concise than hand-written imperative implementations would be.

As Hoare [47] has pointed out, executing specifications is, in general, next to impossible. However, in the last twenty-five years or so researchers have found a variety of ways to specialize particular aspects of this general problem, including: restricting the specification language; restricting the task for which the execution is to be used for; executing only a subset of the specification language; executing only particular kinds of specifications; placing finite bounds on the execution; *etc.* We employ many of these kinds of specializations in this dissertation.

The work presented here executes *abstraction function* specifications for the purpose of *object-contract compliance*. This chapter introduces these concepts, gives a general overview of the dissertation, and provides some historical context. Chapters 2 & 3 explain our approaches to executing abstraction functions. Chapters 4 & 5 deal with object-contract compliance. Chapter 6 concludes.

1.1 Specifications

In the early 1970's researchers in specification languages became interested in separating the meaning of an abstract data type from its concrete representation. For example, a set of integers could be represented by an array or a binary tree or some other concrete structure. It should be possible to specify a set of integers independently of the choice of representation.

Two solutions to this challenge were developed, one in which the abstract values were represented as terms in an algebra, and one in which the abstract values were represented by sets and relations.

The algebraic approach was first presented by Steve Zilles on October 3, 1973, at a workshop at MIT organized by Barbara Liskov [40]. Zilles presented the algebraic specification of a set of integers listed in Figure 1-1 (as recorded by Horning [48], who was in attendance at the talk). In the algebraic approach operations on the abstract datatype are defined by equations that relate them to other ADT operations. The algebraic approach was developed by a number of people, including in the doctoral dissertations of Guttag [39] and Wing [116], which eventually culminated in the Larch system [40].

Figure 1-1 Algebraic specification of a set of integers by Zilles [48]

```
Operators
    Insert:  Sets × Ints → Sets
    Remove:  Sets × Ints → Sets
    Has:     Sets × Ints → Bools
    Null:                → Sets
Relations
    Has(Insert(s, i), j) = if i=j then True  else Has(s, j)
    Has(Remove(s, i), j) = if i=j then False else Has(s, j)
    Has(Null, j) = False
```

In 1972 Hoare [46] proposed instead a model-based approach where the abstract values are represented set-theoretically. Figure 1-2 lists the specification for a set of integers given by Hoare [46]. In the model-based approach operations on the abstract datatype are defined by set-theoretic operators such union ($\cup$), intersection ($\cap$), and membership ($\in$). Our work follows in this model-based tradition.

Figure 1-2 Model-based specification of a set of integers by Hoare [46]

$$s \cdot \mathsf{insert}(x) \equiv s' := s \cup \{x\}$$
$$s \cdot \mathsf{remove}(x) \equiv s' := s \cap \neg\{x\}$$
$$s \cdot \mathsf{has}(x) \equiv x \in s$$

Hoare [46] then shows how to verify that an implementation of a set of integers as an array conforms to the specification of Figure 1-2. Hoare's example implementation is listed in Figure 1-3, where we have translated it into Java from Hoare's original Simula 67 [28] listing. This program has an array, a, and a variable named size that records how many slots of the array are currently being used. Integers are stored in the array in the order in which they are added to the set. Lookup is by linear search.

Figure 1-3 A set of integers implemented with an array (Hoare [46])

```
class ArrayIntSet implements IntSet {
    int[] a = new int[100];
    int size = 0;

    void insert(int x) {
        for (int i = 0; i < size; i++) { if (a[i] == x) return; }
        a[size] = x;
        size++;
    }
    void remove(int x) {
        for (int i = 0; i < size; i++) {
            if (a[i] == x) {
                for (j=i+1; j < size; j++) {a[j-1] = a[j];}
                size--;
                return;
            }
        }
    }
    boolean contains(int x) {
        for (int i = 0; i < size; i++) { if (a[i] == x) return true; }
        return false;
    }
}
```

Hoare [46] verifies that the example implementation of Figure 1-3 meets the specification of Figure 1-2 by introducing an *abstraction function*, which we name $\mathcal{A}$, that maps the concrete program variables to the mathematical domain of the specification. (Elsewhere in the literature this is sometimes referred to as an 'abstraction relation' or 'refinement relation' or 'simulation relation' [81] or 'coupling invariant'.) Figure 1-4 lists the abstraction function that maps the concrete program variables of Figure 1-3 to the abstract values of the specification in Figure 1-2. This example abstraction function is a set comprehension that returns all integers in the array a between indices 0 and size-1.

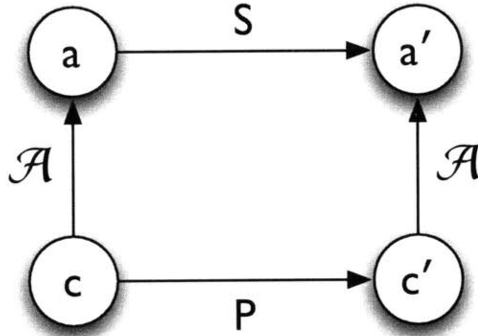
Figure 1-4 Abstraction function for a set of integers represented by an array (Hoare [46])

$$\mathcal{A}(\mathsf{a}, \mathsf{size}) = \{\, x : \mathsf{int} \mid \exists\, i\ (0 \leq i < \mathsf{size} \wedge \mathsf{a}[i] = x) \,\}$$

Before performing the verification, Hoare [46] also introduces the idea of an *invariant*, which is a formula that serves as both a pre-condition and a post-condition for every procedure that operates on the abstract datatype. In this array-implementation example the invariant is simply the condition that $0 \leq \mathsf{size} \leq 100$.

The verification that Hoare [46] performs is what is now known as a *forwards (or downwards) simulation* [118], and is depicted in the commutative diagram in Figure 1-5. We start at concrete pre-state c and attempt to get to abstract post-state a' by two different paths: if the two different paths both lead to a', then we say that the program P refines the specification S. The first path is to apply the program P followed by the abstraction function $\mathcal{A}$. The second path is to apply the abstraction function $\mathcal{A}$ followed by the specification S.

Figure 1-5 Commutative diagram describing data refinement



Dennis [32] states the objective of this proof in the following formula (where I is the invariant introduced above):

$$\forall\, c, c', a \mid (I(c) \wedge \mathcal{A}(c, a) \wedge P(c, c')) \Rightarrow (\exists a' \mid I(c') \wedge \mathcal{A}(c', a') \wedge S(a, a'))$$

This kind of verification of a program against its model-based specification was the original purpose of abstraction functions. The verification is usually done statically, as a mathematical exercise, and so there is no need to make the abstraction function executable. In this dissertation we use abstraction functions as part of a definition of object equality, and so they become an integral part of the program's computation. To support this integration we develop techniques for executing declarative abstraction functions.

1.1.1 Specification Fields

In Hoare's original 1972 paper [46] an abstract data type has a single abstract value. In modern specification languages, such LM3[40], Larch/C++ [62], JML [21, 63], Jahob [59, 123], Spec# [11, 12] and JFSL [32, 119], an abstract data type may have multiple *specification fields* (also known as *model fields*), each of which has a value.

For example, the set of integers discussed above might have a specification field for the elements of the set and another specification field for the size of the set. Each specification field, then, requires its own abstraction function to compute its value from the concrete state. We use the term *field abstraction function* and the symbol α for these functions. We use the term *object abstraction function* and the symbol $\mathcal{A}$ for the combination of these individual specification field abstraction functions. If we consider the field abstraction functions as predicates, as Yessenov [119] does, and also consider object abstraction functions as predicates, as Dennis [32] does, then we could consider that an object abstraction function is the conjunction of the relevant field abstraction functions.

If we use the term *abstraction function* without qualification then the intended meaning

should be clear from context. For example, Chapter 3 is about executing field abstraction functions, whereas Chapter 4 is about defining object equality in terms of object abstraction functions.

A bit of history. The idea of having variables that exist only in the specification and not in the program is at least as old as Hoare's classic 1972 paper on data abstraction [46]. However, the particular incarnation of this idea as specification fields seem to have been developed on the LM3 (Larch-Modula-3) project in the early 1990s. The preliminary design of LM3 in 1991 [14, 54] has specification fields, although not by that name and only private. The first use of the term *specification field* that we can find is the paper [55, §3.4.4] on the semantics of LM3 that Kevin D. Jones presented at the first Larch workshop [77] in 1992. The 1993 description of LM3 included in the Larch book [40] also includes specification fields as we now know them.

To our knowledge, other Larch interface languages that pre-date LM3 do not have specification fields: for example, Larch/CLU [116, 117], Larch/Smalltalk [24], and LCL [109]. While later versions of Larch/C++ do have specification fields [62], the Larch/C++ documentation that pre-dates specification fields in LM3 does not [61].

Leino [65] developed techniques for reasoning (during verification) about scopes that contain both specification fields and the concrete variables that they abstract. Similarly, our users sometimes want to write abstraction functions that contain both abstract and concrete variables. While our system executes these abstraction functions without employing Leino's ideas, our objective of synergy with static verification tools would be diminished if such tools could not verify with these functions. So we have a debt of gratitude to his work for helping our users have their cake and eat it too.

1.1.2 The JForge Specification Language

We use the JForge Specification Language (JFSL) in this work. JFSL is a model-based behavioural interface specification language [41] developed by our colleagues and collaborators, Dennis [32] and Yessenov [119] of MIT's Software Design Group. JFSL may be considered as a variant of Alloy [52] customized for the purpose of specifying Java programs. A key point of the JFSL semantics is that Java fields are modelled as binary relations, following Jackson [51]. For example, the assignment $x.\mathsf{f} = y$ creates the heap reference $x \rightarrow y$, which is modelled as tuple $\langle x, y \rangle$ in binary relation f. Details of JFSL will be introduced by example as we proceed.

1.1.3 Extensions to Abstraction Functions

Since Hoare's original paper introducing abstraction functions [46] researchers have generalized the idea in two different ways: into abstraction *relations* [43], or by augmenting

the specification with auxiliary *history* [74, 92] or *prophecy* [1] variables. These generalizations support the verification of more sophisticated programs or specifications.

History variables are used when the abstract state is not directly computable from the concrete state, but is computable from some combination of the current concrete state and some past concrete state(s). For example, consider a program that computes the mean of a sequence of integers [84]. The program might be implemented so that the concrete state contains only the sum of the sequence and the count of the number of elements in the sequence. However, the specification might be defined in terms of the entire sequence. History variables can be used to provide the extra information needed to map the concrete state to its corresponding abstract state.

Prophecy variables are used when the specification involves non-determinism. Vaandrager [112] gives the example of verifying a distributed summation algorithm. The algorithm involves a spanning tree of distributed nodes. This spanning tree is not known to the code until the algorithm has almost completed: its construction depends on the (unknown) communication links, as well as the latency of the communication between nodes. The specification includes a prophecy variable that predicts what the spanning tree will be.

Abstraction relations might be used in either of these cases instead of the auxiliary variables. There is now a well-established theory of *data refinement* formalizing abstraction relations [9, 29, 46, 84, 85, 87].

Our work is based on only abstraction functions, and so is restricted to deterministic programs in which the abstract state is computable from the concrete state. The Forge [32] tool that we use is also restricted in this way. In the words of Liskov and Wing [72]: 'for most practical purposes, abstraction functions are adequate.'

1.2 Ongoing Example: Binary Search Tree of Integers

Like Zilles [48] and Hoare [46], we will also use a set of integers as our main example. Figure 1-6 shows a Java interface IntSet for a set of integers annotated with specifications written in JFSL that are similar to those given by Hoare [46] above in Figure 1-2. The @SpecField annotation in Figure 1-6 introduces the specification field elts that represents the set of integers modelled by the IntSet. To support verification, concrete implementations of IntSet must provide a declarative abstraction function that defines the elts specification field in terms of the concrete implementation variables.

Figure 1-6 Model-based specification of a set of integers (written in JFSL)

```
@SpecField("elts : set int")
interface IntSet {
    @Ensures("x in this.elts")        void insert(int x);
    @Ensures("x not in this.elts")    void remove(int x);
    @Returns("x in this.elts")        boolean has(int x);
}
```

However, we will use a concrete implementation of this integer set specification based on a binary search tree, rather than the array-based implementation used by Hoare [46] (Figure 1-3). Figure 1-7 lists the Java class declaration and its associated abstraction function for our binary search tree example. The method bodies are elided because they are not germane to our work (since we are not doing verification).

Figure 1-7 JForge abstraction function for binary search tree

```
@SpecField("elts : set int | this.elts = this.root.*(left+right).value")
class BinarySearchTree implements IntSet {
    Node root;
    void insert(int x) { ... }
    void remove(int x) { ... }
    boolean has(int x) { ... }
}
class Node { Node left, right; int value; }
```

The abstraction function for our binary search tree example is defined in the formula to the right of the vertical bar in Figure 1-7: it is this.elts = this.root.*(left+right).value. The sub-expression this.root selects the root node of the tree. The sub-expression *(left+right) denotes a binary relation that includes a tuple from every parent node to each of its descendants (including itself, since * is reflexive transitive closure). The sub-expression this.root.*(left+right) selects all nodes that are descendants of the root. The expression this.root.*(left+right).value returns the set of integers associated with the tree's nodes. In Chapter 3 we will discuss techniques for translating this expression into executable code.

Figure 1-8 lists our binary search tree example with manual implementations of the object contract methods equals and hashCode and helper methods (*e.g.*, iterator). This is the page

of code that the application programmer would have to write without our system. With our system the programmer only needs to write the one line @SpecField annotation listed in Figure 1-7 and the equals and hashCode methods are automatically synthesized.

The equals and hashCode method of Figure 1-8 are written in terms of a iterator method that returns all of the integers stored in the set. The iterator uses the well-known text-book non-recursive algorithm for in-order traversal. In this example, the iterator method serves in the role of the abstraction function.

This iterator-based style is a relatively common way to write object contract methods for well-known data structures: for example, the JDK Collections classes are written in this style. In our observation, this style is not commonly used for other kinds of classes. The more common style is to duplicate the iteration code (*i.e.*, abstraction function) into each of the object contract methods, which sometimes leads to discrepancies between equals and hashCode as the program evolves and the dual maintenance requirement is forgotten about.

One major complexity that the hand-written code for this ongoing example does not have to contend with is cyclic heap references. Because the example is a set of integers, rather than objects, it is not possible to store the set inside itself. However, for many programmer-defined classes the type system does not prevent heap cycles. For these classes extra work must be done to ensure that the object-contract methods terminate. Writing this extra code by hand is difficult and we have not observed programmers doing it in practice. For example, the Javadoc comment for java.util.List says the following:

> While it is permissible for lists to contain themselves as elements, extreme caution is advised: the equals and hashCode methods are no longer well defined on such a list.

One of the advantages of our system is that it terminates correctly in the presence of cyclic heap references, as discussed in Chapter 4 on equality and in Chapter 5 on hashing.

Figure 1-8 Binary Search Tree example with manual implementation of object contract methods

```
class BinarySearchTree implements IntSet {
    Node root;
    void insert(int x) { ... }
    void remove(int x) { ... }
    boolean has(int x) { ... }

    Iterator iterator() {
        return new Iterator() {
            Stack stack = new Stack();
            Node node = (BinarySearchTree.this).root;

            boolean hasNext() { return !(node == null && stack.isEmpty()); }

            Object next() {
                assert hasNext();
                while (node != null) {
                    stack.push(node);
                    node = node.left;
                }
                assert !stack.isEmpty();
                Node n = stack.pop();
                node = n.right;
                return n.value;
            }

            void remove() { throw new UnsupportedOperationException(); }
        };
    }

    boolean equals(Object other) {
        if (other == null) return false;
        if (other == this) return true;
        if (!(other instanceof IntTree)) return false;
        IntTree that = (IntTree)other;
        if (this.size() != that.size()) return false;
        for (Iterator it = iterator(); it.hasNext(); ) {
            if (!that.has(it.next())) return false;
        }
        return true;
    }

    int hashCode() {
        int hash = 17;
        for (Iterator it = iterator(); it.hasNext(); ) {
            hash += it.next();
        }
        return hash;
    }
}

class Node { Node left, right; int value; }
```

1.3 Executable Specifications

The desire to automatically execute specifications is an old one and it has been approached from many angles by many researchers. Here we discuss some parts of the literature that have influenced this work.

In the 1980's Back [8], Back and von Wright [9], Hehner [44], Morgan [83, 85], Morris [86], and others showed how considering code and specifications in the same light made the calculus of program refinement simpler and more systematic. Morgan [85] names programs that include both specification statements and imperative statements *mixed programs*. While the researchers of this time considered mixed programs to be executable in theory, they were primarily concerned with pencil-and-paper mathematical reasoning about such programs rather than mechanized execution or analysis.

In the 1990's there was much discussion about specification animation and analysis. Hayes and Jones [42] argued that attempting to make specifications executable in the general case would result in compromised specification languages in which programmers would write poor specifications that were biased towards particular implementation strategies. Fuchs [37] argued that the benefits to the programmer in terms of specification validation and understanding were strong and that executable specifications should be pursued as a research direction. One might view systems such as Alloy [52] as evolving from these debates. The kind of specification analysis and animation it performs are what Fuchs [37] had in mind in terms of specification 'execution': that is, a model-theoretic evaluation that produces concrete examples. In all of this, the discussion was focused on specifications rather than programs — and these weren't mixed.

More recently, there has been some work executing specifications for software engineering purposes, such as testing, rather than for computing particular outputs: for example, the JML runtime assertion checker by Cheon [23], test-input generation strategies by Khurshid [57] and Marinov [76], debugger support in the Eclipse IDE, and data-structure repair by Demsky [31] and Elkarablieh et al. [35]. Many of these more recent works focus on executing invariants or abstraction functions rather than procedure post-conditions. These invariant and abstraction function specifications are sometimes written in a declarative specification language and sometimes in stylized imperative code.

These works based on executing invariants and abstraction functions for software engineering purposes are usually not considered to be examples of mixed programs. However, we have elsewhere argued that they can be viewed in this way [100], and we have found that viewpoint to be fruitful for this dissertation. The reason that they are not considered to be examples of mixed programs is that the result of evaluating an invariant or an abstraction function is not usually part of the output computed by the program. However, because we define object equality in terms of abstraction functions, the result of evaluating the abstraction function at runtime is part of the computed output of the program.

There has been much other work on executable specifications that has not been a direct inspiration for our work here. For example, Pamela Zave developed the PAISLey system [120] for highly-parallel systems, possibly with real-time constraints, and without unbounded data structures. Mandayam Srivas [108] wrote his 1982 dissertation on syn-

thesizing implementations for abstract data types from their algebraic specifications with a term re-writing system, and work in this spirit continues today (*e.g.*, Henkel *et alia*'s 2008 paper [45]). Our work is from model-based, rather than algebraic, specifications, and we use a syntax-directed translation rather than term-rewriting.

1.4 The Object Contract

The Object Contract is the name commonly used for the specification that all objects are supposed to comply with in languages such as Java and C#. In short, the Object Contract states that the equals method must define a mathematical equivalence relation, and that the hashCode method must return results that are consistent with the equals method.

Object-contract compliance is important to the correct functioning of any program (written in these languages). Any lookup operation will require that the desired key object and the stored key object are correctly able to compare with each other for equality. Hash-based lookups will additionally require that the hashcodes of equal objects are equal.

The essence of the Object Contract can be written as an algebraic specification in the Larch Shared Language, as listed in Figure 1-9 (adapted from examples in Appendix A of the Larch book [40]). The specification in Figure 1-9 says that an Object has an equals method that compares two objects and returns a boolean, and a hashCode method that maps objects to integers. The equals method defines a relation that is reflexive, symmetric, and transitive (*i.e.*, is a mathematical equivalence relation). The consistency requirement for the hashCode method is that if two objects are equal then they must also hash to the same value. The equals and hashCode methods are expected to be side-effect free and to terminate on all inputs.

Figure 1-9 The Object Contract as an algebraic specification

```
Object : trait
    introduces:
        _ equals _ : Object, Object → Bool
        _ hashCode : Object → Int
    asserts ∀ x, y, z : Object
        x equals x ;                                    % reflexive
        (x equals y) == (y equals x) ;                  % symmetric
        (x equals y ∧ y equals z) ⇒ (x equals z) ;      % transitive
        (x equals y) ⇒ (x hashCode == y hashCode) ;     % consistent
```

While the Object Contract appears relatively simple, numerous studies have documented that programmers at all levels of ability have difficulty writing compliant code [17, 49, 91, 93, 94, 97, 114]. Some of the challenges of object-contract compliance are discussed by Bloch [17], and Odersky et al. [91], and below in Chapter 4.

1.5 Thesis Statement

The problem of making specifications executable, in its full generality, is not (at this time) practical. In this dissertation we narrow our focus in a number of ways: to a particular kind of specification, *abstraction functions*; to a particular use for executable specifications, *object-contract compliance*; and to a subset of a general-purpose specification language, *JForge* [32, 119]. The thesis that we establish in this dissertation is:

> Executable abstraction functions enable object-oriented programming languages to provide better support for object-contract compliance, resulting in clearer and more concise programs that have fewer errors.

Our aspiration is that, as a consequence of this work, programmers will no longer be obliged to implement the equals and hashCode methods by hand. Implementing these methods manually is both tedious and time consuming: in our opinion it offers a poor return on investment for the programmer's effort. We think that application programmer attention is better spent on matters more closely connected to the application domain. We hope that the automatic synthesis of object-contract compliance will remove one more technology-related distraction from the application programmer's attention space and thereby improve programmer productivity.

1.6 Contributions

This dissertation makes two observations:

1. Expert programmers sometimes define object equality in terms of the abstract state of the object. When doing this they sometimes write an imperative procedure to embody the conceptual abstraction function, and then write the equality comparison in terms of this procedure.
2. The field abstraction functions that expert programmers most commonly employ can be considered as navigation queries on the heap.

We develop the following ideas to put these insights to good use:

- *Four techniques for evaluating declarative field abstraction functions at runtime.* We develop four techniques for executing declarative field abstraction functions written in JFSL at runtime. These techniques are applicable to different subsets of possible abstraction functions and have different runtime speeds. The most general technique executes the slowest, and the least general technique executes the fastest. Generality here refers to the class of JFSL expressions that the technique is applicable to: each technique is applicable to a different subset of possible JFSL expressions. A static analysis is used to determine which technique to apply for a given abstraction function. The techniques are:

- *Conservative Navigation:* The conservative navigation technique is based on a syntax-directed translation from a subset of JFSL to executable code. We refer to this subset as *navigable expressions:* in broad outline, they begin with a set, involve forwards traversal of fields, and result in a set. Navigable expressions are characterized more precisely in Chapter 3.
  - *Streamlined Navigation:* The streamlined navigation technique is an optimized version of the conservative navigation technique. If certain invariants, which we call *optimization conditions*, are true of the program, then faster alternative computations may be used. For example, list concatenation may be used in place of set union if the optimization conditions establish that the lists do not contain duplicates and have no intersection with each other.
  - *Predicate Substitution:* If the abstraction function is written using a pre-defined predicate with a known imperative implementation, then that implementation is used at runtime. This technique is usually employed for abstraction functions returning sequences of objects (rather than sets of objects). This technique is applicable in the smallest number of cases but is the fastest to execute.
  - *Constraint Solving:* A constraint solver (Kodkod [110, 111]) is used to compute the result of the abstraction function. This technique is the most general and is also the slowest to execute.

- *A library of logical predicates for use in declarative abstraction functions.* These predicates have well-known imperative implementations. We argue that the logical definitions we provide here are consistent with the well-known imperative versions.

- *A generic definition of object equality.* We say that two objects are equal if they have the same specification fields with the same values, and if those specification fields were declared in a common supertype.

- *An exploration and evaluation of object hashing strategies.* We argue that programming language implementations should automatically provide object hashing. This provision is facilitated by our generic notion of object equality. We evaluate the performance of half a dozen object hashing strategies and show that it is possible for automatically provided hashing to perform acceptably in comparison to manually implemented hashing in most cases.

We evaluate these ideas by annotating a number of real programs and measuring the resulting change in code size, performance, *etc.*.

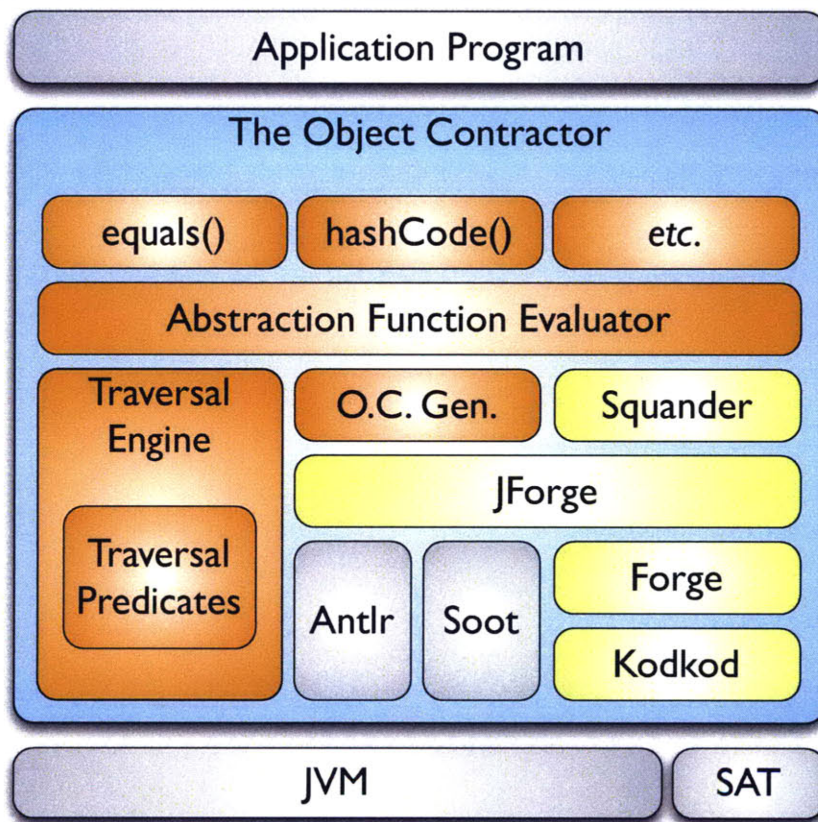
1.7 Architectural Overview of Prototype Implementation

Figure 1-10 gives an architectural overview of our prototype implementation. This prototype has been used for all of the case studies in this dissertation and, additionally, in a Bayes net solver and a multi-objective optimization solver [99] written by the author.

While there is very limited experience of other people programming with this prototype, around ten people have used the multi-objective optimization solver in which all classes with programmer-defined equality make use of the prototype.

Each box in Figure 1-10 represents a component. Components depend on the components depicted below it in the diagram. Containment indicates that the contained box is a part of the containing box. The blue box labelled *The Object Contractor* represents this work as a whole. The grey boxes represent third-party components. The yellow boxes represent components developed by my colleagues in the Software Design Group at MIT. The orange boxes represent specific parts of this work.

Figure 1-10 Architectural overview of prototype implementation



The overall picture is that the application program uses The Object Contractor to provide implementations of equals, hashCode and other common methods such as toString. Both The Object Contractor and the application program run on top of a Java Virtual Machine (JVM).

The Object Contractor includes generic implementations of equals and hashCode in terms of the abstract state of the objects. These generic implementations are discussed in Chapters 4 and 5, respectively, and they depend on our field abstraction function evaluator to compute the object's abstract state.

Our field abstraction function evaluator is summarized above and described in detail in Chapter 3.

Our constraint solving execution strategy is applicable to any field abstraction function written in the JForge specification language. It is implemented in the Squander tool, which was developed jointly by myself and Aleksander Milicevic, with contributions from Kuat Yessenov. Squander also makes use of JForge, but only for its semantic mapping between Java and Forge, rather than for verification purposes.

JForge [119] defines the JForge Specification Language and provides a translator from that surface syntax to the Forge Intermediate Representation (FIR) [32]. JForge was written by Kuat Yessenov [119] in the Software Design Group at MIT, with some minor contributions by myself as part of this work. JForge makes use of the Antlr [96] parser-generator written by Terrence Parr at the University of California, San Francisco. JForge also uses the Soot [113] Java bytecode analysis and transformation framework developed by Laurie Hendren's students at McGill University.

Forge [32] is a program analysis framework primarily intended for performing bounded program verification: *i.e.*, verifying that an imperative object-oriented program meets its logical specification, given some a priori bounds on the size of the heap and the number of loop unrollings. Forge was written by Greg Dennis [32] in the Software Design Group at MIT.

Kodkod [110] is a relational model finder. It translates formulæ from the 'Alloy logic' (*i.e.*, a first-order logic with relations and transitive closure) to propositional logic. The propositional logic formulas are then given to a SAT-solver such as MiniSAT [34]. Kodkod is the backend of the fourth version of the Alloy Analyzer tool [22]. Kodkod was written by Emina Torlak [110] in the Software Design Group at MIT.

1.8 Summary

Programmers often define object equality (and other object contract methods) implicitly in terms of the abstract state of the object. We show that providing an explicit means to express *abstraction functions* effectively separates these concerns and facilitates generic versions of the object contract methods. We develop several novel techniques for executing abstraction functions written in a declarative specification language. The key insight behind these techniques is that the kinds of abstraction functions that programmers often write may be considered as queries on the reachable heap.

In the literature, abstraction functions are usually considered to be relevant only to specification and verification: they are generally not considered to be part of the computation of the program. However, by defining the object contract methods explicitly in terms of abstraction functions we make the execution of abstraction functions an integral part of the computation. We hope that this dynamic use of abstraction functions will entice programmers to write specifications more often.

Chapter 2

Expressing Abstraction Functions

In our system the imperative code of each abstract data type (ADT) is accompanied by a declarative abstraction function written in a modern specification language (JFSL [32, 119]). In most modern specification languages the abstract state of an ADT is described by a set of specification fields. Each specification field has an associated *field abstraction function* that computes its abstract value from the concrete representation. We understand the *object abstraction function* to be the amalgamation of these field abstraction functions.

The abstract value of each specification field is a set or a sequence of concrete objects. If we give a concrete representation to this abstract mathematical structure (containing concrete objects) then we can have a concrete representation of the abstract state. The concrete representations we use for these abstract structures are described below in Chapter 3.

Abstraction functions for most data types can be expressed as a *navigation* over the heap, starting with the receiver object, and following a path that collects a set or sequence of objects to be returned as the abstract value. This kind of navigation, it turns out, is much more easily expressed using relational navigation operators (notably join and closure) than with conventional imperative code, and this is the source of much of the succinctness of our abstraction functions. In some cases, though – notably tree traversals that return a sequence of objects in a particular order, as well as some operations on lists – these relational operators are insufficient. While it is possible to express these more complex traversals using relational logic, it is more cumbersome than we would like it to be. We therefore show how a repertoire of standard traversals can be represented as a library of predicates, enabling abstraction functions to be expressed simply by invoking them in expressions. Matching imperative implementations allow these predicates to be executed efficiently, and their full logical formulations are available for verification.

The technical contribution of this chapter is the logical definitions of the predicates. We do not use these definitions for execution: for execution we simply substitute the well-known imperative version of the predicate name. These logical definitions are for use by a verification tool.

It is our hope that our system will provide some immediate and tangible motivation for programmers to write partial specifications, and that this will be a stepping stone for them

to write more complete specifications and use formal verification tools. We provide these predicate definitions to support this potential future verification task.

This chapter also provides a brief overview of the JForge Specification Language (JFSL), which is described in greater detail in Kuat Yessenov's master's thesis [119].

2.1 Choosing a Specification Language

Abstraction functions are most commonly written in a specification language and used during verification. We considered three specification languages for use in our system: JML [21, 23, 63], Jahob [18, 121, 122], and JForge [32, 119]. To illustrate the comparison we examine our ongoing example of a set of integers represented as a binary search tree in each language. These examples are shown in Figures 2-1, 2-2, and 2-3, respectively.

As seen in Figure 2-1, although JML is a specification language, it has the programmer express abstraction functions as imperative code. Cheon [23] developed this facility in his dissertation to support runtime verification. For our purposes this is essentially the same as an older version of our system [98] in which the programmer wrote the abstraction functions with regular imperative code. The shortcomings of this approach are that: (1) the abstraction functions tend to be longer than they would be if written in a declarative specification language; (2) the programmer must handle cyclic heap references manually in an ad-hoc manner; (3) it is less compatible with static verification.

Jahob [59, 123] uses the Isabelle higher-order logic [89] to express abstraction functions (and other parts of specifications), as shown by the example in Figure 2-2. The purpose of the Jahob project is to integrate various back-end approaches to static verification, such as shape analysis, decision procedures, and theorem provers.

JForge [32, 119] uses a variant of the Alloy [52] relational logic for expressing abstraction functions (and other parts of specifications), as shown by the example in Figure 2-3. JForge is also designed for static verification.

The Alloy relational logic is, in a technical sense, less powerful than the Isabelle higher-order logic. However, the Alloy relational logic has proven, through the experience of other researchers, to be adequately powerful for many software specification tasks.

We chose to work with Alloy/JFSL because its relational operators let programmers write powerful abstraction functions with a simple syntax: *e.g.*, many abstraction functions can be written without quantifiers or set comprehensions. Our code synthesis techniques (discussed in Chapter 3) are substantially enabled by this syntactic simplicity.

Figure 2-1 JML abstraction function for binary search tree (adapted from [23])

```
class BinarySearchTree {
    Node root;
    /*@
    model Set elts;
    represents elts ← eltsValue();
    model pure Set eltsValue() throws IllegalStateException {
        Set ret = new HashSet();
        if (root != null) { ret.addAll(root.subtree()); }
        return ret;
    }
    @*/
}
class Node {
    Node left, right;
    int value;
    /*@
    model pure Set subtree() throws IllegalStateException {
        Set ret = new HashSet();
        ret.add(new Integer(value));
        if (left != null) { ret.addAll(left.abstractValue()); }
        if (right != null) { ret.addAll(right.abstractValue()); }
        return ret;
    }
    @*/
}
```

Figure 2-2 Jahob abstraction function for binary search tree (adapted from [124])

```
specvar elts :: int set;
vardefs elts == {x. ∃ n. n : subtree & n..value = x}

specvar subtree :: objset;
vardefs subtree == {n0. n0≠null ∧ (this,n0) ∈ {(x,y). x..left=y ∨ x..right=y}^*};

class Node { Node left, right; int value; }
class BinarySearchTree { Node root; }
```

Figure 2-3 JForge abstraction function for binary search tree (adapted from Figure 1-7)

```
@SpecField("elts : set int | this.elts = this.root.*(left+right).value")
class BinarySearchTree { Node root; }
class Node { Node left, right; int value; }
```

2.2 JForge Specification Language Overview

The JForge Specification Language (JFSL) [32, 119] is a specification language for Java (*i.e.*, imperative object-oriented) programs based on the Alloy [52] relational logic. JFSL was developed by Yessenov [119] and Dennis [32] to support static verification of functional properties of single-threaded programs.

This section gives a brief overview of the pertinent features of JFSL. The expressions and formulæ of JFSL are essentially Alloy. JFSL adds on top of this special annotations to connect these expressions and formulæ to code. Yessenov's master's thesis [119] discusses the syntax [119, §3] and semantics [119, §4] of JFSL in detail.

Figure 2-4 briefly summarizes the JFSL annotations. We are primarily interested in the @SpecField annotation, which both declares specification fields and defines their associated field abstraction function. The other annotations listed in Figure 2-4 are included to give the reader an overall sense of JFSL; they are not discussed further in this document. In the notation of Figure 2-4 we sometimes use Greek characters such as ϕ as place-holders for Alloy formulæ. JFSL uses only the common 7-bit ASCII characters, and does not use Greek letters. For example, in Figure 2-4 we write @Invariant("ϕ"), where ϕ stands for some Alloy formula such as this.size = #(this.root.*(left+right)).

Figure 2-4 Summary of JForge Specification Language (JFSL) annotations

Types	
@SpecField("f : r from g | α")	defines a specification field f, characterized by the range-type expression r, and its abstraction function α; g is the list of concrete fields to which f corresponds; more detail in the main text
@Invariant("ϕ")	defines an invariant (ϕ is a JFSL formula)
Fields	
@NonNull	indicates that this field may never refer to null
@Nullable	indicates that this field may refer to null
Methods	
@Requires("ϕ")	pre-condition (ϕ is a JFSL formula)
@Ensures("ϕ")	post-condition (ϕ is a JFSL formula)
@Modifies("g")	frame condition (g is a list of fields that may change as a result of executing this method)
@Returns("e")	return value (e is a JFSL expression)
@Pure	indicates that this method has no side-effects

2.2.1 Specification Fields and Abstraction Functions

As shown in Figure 2-4, the general form of the @SpecField annotation is:

$$@SpecField("\ f : r \text{ from } g \mid \alpha \ ")$$

where f is the name of the specification field being defined, r is an expression that describes the multiplicity and type of f's range, g is f's data group, and α is f's field abstraction function. We will now explain these in overview. The exact grammar of the specification field annotation is discussed in detail in §3.5.2 of Yessenov's master's thesis [119]. To begin, Figure 2-5 shows a simple example @SpecField annotation for a Java class that represents a mathematical set that contains at most one object.

Figure 2-5 Abstraction function for SingletonSet

```
@SpecField("elts : set Object")
interface Set { }

@SpecField("elts : set Object | this.elts = this.o - null")
class SingletonSet implements Set { Object o; }
```

In the example of Figure 2-5 the range expression r is set Object. In principle r can be more complex than this, but in practice it rarely is. The declaration of the elts specification field on the Set class and the set Object range expression mean that elts is a binary relation from Set to Object. The possible multiplicity keywords are [119]:

- **one** — the field has a singleton value, similar to Java fields;
- **some** — the value is a non-empty set;
- **lone** — the value is either an empty-set or a singleton;
- **set** — the value is a set;
- **seq** — the value is a sequence.

As in Alloy [52], if r has arity 1 (which is always the case in practice), then the default (unspecified) multiplicity is **one**; otherwise it is **set** [119].

The **seq** multiplicity keyword is special [119]. When it is used, then f is a ternary relation from C (the Java type that declares f), to **int** (the sequence indices), to the type(s) named in r. The use of **seq** also implies extra constraints on f that the **int**s in the relation start at zero, are contiguous, each appear in the relation at most once, *etc.*, so that the programmer gets a relation that corresponds with what one would intuitively expect from a sequence.

These multiplicity constraints are implicitly part of the specification field's abstraction function and so are conjoined with what the programmer explicitly writes in α.

The data group [66] of specification field f, g, is a list of concrete fields from which the value of specification field f is derived. This list is used during the static verification of methods: if a concrete field on this list is mutated by the method, then it may be possible

that the value of specification field f has also been changed. We do not make use of this list (g) in our work, and so we have excised it from all of our example listings and will not discuss it further.

Finally, α is a JFSL formula that defines the field abstraction function. It is most commonly of the form:

$$\text{this}.f = e$$

where e is some JFSL expression. In the small example from Figure 2-5, this expression is simply this.o − null.

However, the abstraction function α does not need to be in this assignment-like equational form. Any valid JFSL formula will do. Another common form for α is:

$$p(e_1, e_2, ..., e_n, \text{this}.f)$$

where p is some logical predicate and e_i are JFSL expressions. We provide a library of such predicates, mostly for working with sequences, for the programmer's convenience. For example, Figure 2-6 shows an abstraction function for a cons cell class using the prepend predicate (explained below). Similarly, Figures 2-7 and 2-8 show abstraction functions for a linked list and an array list, using the link2seq and subseq predicates, respectively. More complete descriptions of the available predicates are given below.

Figure 2-6 Abstraction function for ConsCell

```
@SpecField("elts : seq Object | prepend(this.head, this.rest.elts, this.elts)")
class ConsCell extends AbstractImmutableLinkedList {
    final Object head;
    final AbstractImmutableLinkedList rest; }
```

Figure 2-7 Abstraction function for LinkedList

```
@SpecField({
    "private nodeseq : seq Node | link2seq(this.head, Node@next, this.nodeseq)",
    "elts : seq Object | this.elts = this.nodeseq.elt"})
class LinkedList implements List { Node head; }
class Node { Object elt; Node next; }
```

Figure 2-8 Abstraction function for ArrayList

```
@SpecField("elts : seq Object | subseq(this.a.elems, 0, this.size, this.elts))
class ArrayList implements List { Object[] a; int size; }
```

Recall our ongoing example of a binary search tree, originally introduced in Figure 1-7. In Figure 1-7 the specification field elts was defined as a set of integers with the abstraction

function **this**.elts = **this**.root.*(left+right).value. One could imagine the programmer wanting to have some other specification fields as well. For example, it may be useful to define a private field nodes for writing invariants. Such a field is introduced in Figure 2-9, and an alternative definition of the elts field is given in terms of it. Since this is a binary search tree, the programmer may also wish to have the elements in order: this is done with the nodeseq and eltseq fields.

Figure 2-9 Various abstraction functions for ongoing binary search tree example

```
@SpecField({
    "private nodes : set Node | this.nodes = this.root.*(left+right)-null",
    "public elts : set int | this.elts = this.nodes.value",
    "private nodeseq : seq Node |
        inorder(this.root, Node@left, Node@right, this.nodeseq)",
    "private eltseq : seq int | this.eltseq = this.nodeseq.value"})

class BinarySearchTree { Node root; }
class Node { Node left, right; int value; }
```

Finally, occasionally the programmer will have cause to write an abstraction function that is not in either of the above forms. For example, the abstraction function for empty cons cell in Figure 2-10 is simply no this.elts. If one were to model a Lisp-style linked list as an algebraic datatype in Java, then one might introduce a class EmptyConsCell, as in Figure 2-10, to represent the case of a node with no element and no further links.

Figure 2-10 Abstraction function for EmptyConsCell

```
@SpecField("elts : seq Object | no this.elts")
class EmptyConsCell extends AbstractImmutableLinkedList {}
```

2.3 Traversal Predicates

We have written a library of traversal predicates, primarily for working with sequences, that are summarized in Figure 2-11. Some of these predicates are adapted from the Alloy standard predicate libraries and some of them are new. This library of predicates serves two purposes: programmer convenience and efficiency of execution. As will be seen below, the logical definitions of some of these predicates can be both lengthy and subtle. Having a predefined library is easier for the programmer than developing these from scratch on an ad hoc basis. Since these predicates have well-understood imperative implementations, our translator (discussed in the next chapter) can simply substitute these well-known imperative versions when executing the abstraction functions.

Figure 2-11 Traversal predicates

Predicate signature	Description
link2seq(*head* : one Object , *next* : Object → Object , *result* : seq Object)	Constrain *result* to be the sequence of objects starting with *head* and following the links in relation *next*. Expects *next* to be linear (potentially cyclic).
obj2seq(*obj* : one Object , *result* : seq Object)	*result* = (0→obj)
prepend(*first* : one Object , *rest* : seq Object , *result* : seq Object)	Constrain *result* to be the sequence *rest* prepended with object *first* at position 0.
subseq(*seq* : one Object , *index* : int , *length* : int , *result* : seq Object)	Constrain *result* to be the subsequence of *seq* of length *length* starting at index *index*.
subseqeven(*seq* : one Object , *result* : seq Object)	Constrain *result* to be a sequence that includes only objects that were at even indices in *seq*.
subseqodd(*seq* : one Object , *result* : seq Object)	Constrain *result* to be a sequence that includes only objects that were at odd indices in *seq*.
preorder(*root* : one Object , *tree* : Object → Object , *result* : seq Object)	Constrain *result* to be the preorder (*i.e.*, depth-first) traversal of tree *tree*, which is rooted at *root*.
postorder(*root* : one Object , *tree* : Object → Object , *result* : seq Object)	Constrain *result* to be the postorder traversal of tree *tree*, which is rooted at *root*.
levelorder(*root* : one Object , *tree* : Object → Object , *result* : seq Object)	Constrain *result* to be the levelorder (*i.e.*, breadth-first) traversal of tree *tree*, which is rooted at *root*.
inorder(*root* : one Object , *left* : Object → Object , *right* : Object → Object , *result* : seq Object)	Constrain *result* to be the inorder traversal of the binary tree rooted at *root*, with left children indicated by relation *left* and right children indicated by relation *right*.

The link2seq predicate produces a JFSL sequence from a Java linked-list structure. In order to discuss the details of link2seq we first introduce a simple linked-list structure, given in Figure 2-12, to use as the basis of the discussion.

Figure 2-12 Linked-list structure

```
sig List { head : lone Node }
sig Node { next : lone Node, val : lone Object }
```

A naïve definition of link2seq is given in Figure 2-13. This definition works only for acyclic lists, and is essentially the same as the one given in [57, §6.4.2]. This naïve definition will result in an infinite sequence if applied to a cyclic list. (In practice the sequence will not actually be infinite: it will be bounded by the scope of the int type. Nevertheless, this predicate will not produce the result the programmer intends when applied to a cyclic list.)

Figure 2-13 Naïve forwards traversal of linked-list

```
pred link2seq(head : Node, next : Node→Node, result : seq Node) {
  result[0] = head
  all i : Int | i > 0 ⇒ result[i] = result[i−1].next
}
```

Some previous work claims that it is difficult or impossible to define linear traversals of cyclic structures with Alloy [57, §6.4.4 & §6.5]. An argument that has been given is that a list of visited nodes must be maintained in order to detect cycles, and that while this is easy in an imperative implementation, it is difficult to do declaratively [57, §6.5]. We agree that it is difficult to maintain a list of visited nodes in a declarative context, but we now show that there is at least one other way to approach this challenge: first, compute what the length of the result should be, and then constrain the result to be exactly that length.

Figure 2-14 shows an improved version of the linked-list traversal predicate from Figure 2-13. The key difference is that we compute the size of the list using set cardinality and use this to differentiate between in-bounds and out-of-bounds indices. This improved predicate will produce the desired result even for cyclic lists.

Figure 2-14 Forwards traversal of a potentially cyclic linked-list

```
pred link2seq(head : Node, next : Node→Node, result : seq Node) {
  result[0] = head
  let size = #(head.^next) | {
    -- appropriately constrain result at in-bounds indices
    all i : Int | (i > 0) and (i < size) ⇒ result[i] = result[i−1].next
    -- ensure that garbage is not injected to result out of bounds
    all i : Int | (i < 0) or (i ≥ size) ⇒ no result[i]
  }
}
```

2.3.1 Tree-Traversal Predicate Definitions

Trees and tree-like structures are common in many kinds of programs. Consequently we have developed a library of common tree-traversal predicates to make these expressions easier for the programmer. The signatures of these predicates were given above in Figure 2-11, along with the other traversal predicates. The tree-traversal predicates are preorder (*i.e.*, depth-first), postorder, levelorder (*i.e.*, breadth-first), and inorder. The first three are applicable to trees of any arity, whereas inorder is only applicable to binary trees. Here we discuss the logical definitions of these predicates, and make some efforts to establish that our logical definitions correspond to the well-known recursive meanings.

For reference we give the standard recursive definitions of these common tree traversals in Figure 2-15. Figure 2-15 is written in the Haskell [105] programming language because it is clear and concise for this kind of task. In Haskell [] constructs a list and ++ concatenates lists. Iterative worklist-based implementations of these traversals are also commonly known (*e.g.*, [115]).

Figure 2-15 Standard recursive definitions of pre-order, post-order, in-order, and level-order traversals of binary trees [101]

```
data Tree a = Empty
            | Node { value :: a,
                     left  :: Tree a,
                     right :: Tree a }

preorder, inorder, postorder, levelorder :: Tree a → [a]

preorder Empty         = []
preorder (Node v L R)  = [v] ++ preorder L ++ preorder R

inorder Empty          = []
inorder (Node v L R)   = inorder L ++ [v] ++ inorder R

postorder Empty        = []
postorder (Node v L R) = postorder L ++ postorder R ++ [v]

levelorder x = loop [x]
    where loop []                  = []
          loop (Empty        : xs) = loop xs
          loop (Node v L R : xs)   = v : loop (xs ++ [L,R])
```

Figures 2-17, 2-18, 2-19, and 2-16 give our declarative definitions for post-order, pre-order, level-order, and in-order tree traversals, respectively. While the standard recursive definitions given in Figure 2-15 are all written in terms of an ordered binary tree, in principle only the in-order traversal needs to be defined this way: all of the other traversals easily generalize to unordered n-ary trees. Consequently, we give our definitions of post-order (Figure 2-17), pre-order (Figure 2-17), and level-order (Figure 2-17), traversals in terms of unordered n-ary trees.

These four tree-traversal predicates all begin by declaring the name tnodes to refer to all of the nodes in the tree. The expression for determining all of the nodes in the tree differs depending on whether it is an unordered tree of arbitrary arity (root +root.^children) or an ordered binary tree (root +root.^(left+right)). The name tnodes is used in almost all subsequent clauses.

All four definitions also constrain the resulting sequence (result) to have each of the nodes in the tree (tnodes) exactly once. The remaining parts of the predicates differ as they constrain the ordering of the resulting sequences.

The analyses we have performed (described below in the next section) have focused on establishing that these logical predicates properly formalize their well-known intuitive meanings. It may be possible that, for each predicate, there is a more concise yet logically equivalent formulation.

Figure 2-16 lists the predicate for an in-order traversal of an ordered binary tree. The ordering constraint says that for every node in the tree, none of the nodes to the left of it occur later in the resulting sequence, and none of the nodes to the right of it occur earlier in the resulting sequence.

Figure 2-16 Declarative definition of in-order traversal of a binary tree

```
pred inorder [ root : Node, left, right : Node->Node, result : seq Node ] {
  let tnodes = root +root.^(left+right) | {
    -- every node in the tree is in the result exactly once
    all n : tnodes | one result.n
    Int.result = tnodes
    -- left, parent, right
    all n : tnodes | {
      no lc : (n.left).^(left+right) | result.lc > result.n
      no rc : (n.right).^(left+right) | result.rc < result.n
    }
  }
}
```

Figure 2-17 lists our declarative definition of a post-order traversal of an unordered n-ary tree. The general idea of a post-order traversal is that children/descendants come before their parents. However, this property alone is not enough, as we learned through conducting the analyses described in the subsequent sections of this chapter. We must also specify that the children of a node appear earlier in the resulting sequence than the siblings of that node. Furthermore, every parent node must appear immediately after one of its children in the resulting sequence.

Figure 2-17 Declarative definition of post-order traversal of an n-ary tree

```
pred postorder [ root : one Node, children : Node→Node, result : seq Node ] {
  let tnodes = root +root.^children | {
    -- every node in the tree is in the result
    Int.result = tnodes
    all n : tnodes | one result.n
    -- descendants come before parents
    all disjoint p, c : tnodes | ((p→c) in ^children) ⇒ (result.p > result.c)
    -- children before siblings
    all disjoint p, c : tnodes | (p→c) in children ⇒ all s : siblings[p, children] | (result.c < result.s)
    -- parent comes immediately after some child
    all p : tnodes | some p.children ⇒ some c : p.children | result.p = ((result.c)+1)
  }
}
```

Figure 2-18 lists our declarative definition of a pre-order (*i.e.*, depth-first) traversal of an unordered n-ary tree. The general idea of a pre-order traversal is that parent nodes are visited before their descendants. However, this condition alone is insufficient to ensure the correct result. The additional conditions that we have added include: the immediate successor of every node is a child of that node (if a child exists); sibling nodes must occur before uncle nodes; children nodes must occur before uncle nodes; and, finally, younger generations must respect older generations. This last condition means that for any two distinct nodes n_1 and n_2 that are siblings of each other, if n_1 occurs before n_2 in the output sequence, then all descendants of n_1 occur before n_2 in the output sequence.

Figure 2-18 Declarative definition of pre-order traversal of an n-ary tree

```
pred preorder[ root : one Node, children : Node→Node, result : seq Node ] {
  let tnodes = root +root.^children | {
    -- every node in the tree is in the result
    Int.result = tnodes
    all n : tnodes | one result.n
    -- parents come before descendants
    all disjoint p, c : tnodes | ((p→c) in ^children) ⇒ (result.p < result.c)
    -- some child is the immediate successor
    all p : tnodes | some p.children ⇒ some c : p.children | result.c = ((result.p)+1)
    -- siblings before uncles
    all n : tnodes | all s : siblings[n, children] | all u : uncles[n, children] | result.s < result.u
    -- children before uncles
    all n : tnodes | all c : n.children | all u : uncles[n, children] | result.c < result.u
    -- younger generations respect older generations
    all n : tnodes | all s : siblings[n, children] | {
      (result.n < result.s) ⇒ all c : n.^children | (result.c < result.s)
      (result.n > result.s) ⇒ all c : n.^children | (result.c > result.s)
    }
  }
}
```

Figure 2-19 lists our declarative definition of a level-order (*i.e.*, breadth-first) traversal. As

with a pre-order (*i.e.*, depth-first) traversal, the general idea is that parents nodes occur before their descendants. Given this similarity, there is a (perhaps surprisingly) large class of trees for which the breadth-first and depth-first traversals produce the same resulting sequence (discussed in more detail below). There can be significant differences in the order in which descendants are visited though. The additional conditions that we have added to define level-order traversal include: child nodes occur before grandchildren; children are all consecutive; and younger generations respect older generations. This last condition is the same as the condition added to the pre-order definition above.

Figure 2-19 Declarative definition of level-order traversal of an n-ary tree

```
pred levelorder [ root : one Node, children : Node→Node, result : seq Node ] {
  let tnodes = root +root.^children | {
    -- every node in the tree is in the result
    Int.result = tnodes
    all n : tnodes | one result.n
    -- parents come before descendants
    all disjoint x, y : tnodes | ((x→y) in ^children) ⇒ (result.x < result.y)
    -- children before grandchildren
    all disjoint p, c, g : tnodes | {
      (p→c) in children and (c→g) in children ⇒ (result.c < result.g)
    }
    -- children are all consecutive
    all p : tnodes | { #(p.children) ≥ 2 ⇒ (
      -- no monkey in the middle
      no x : tnodes−p.children | some disjoint c1, c2 : p.children | {
        result.c1 < result.x and result.x < result.c2
      }
    )}
    -- younger generations respect older generations
    all n : tnodes | all s : siblings[n, children] | {
      (result.n < result.s) ⇒ all c : n.^children | (result.c < result.s)
      (result.n > result.s) ⇒ all c : n.^children | (result.c > result.s)
    }
  }
}
```

2.3.2 Tree Traversal Predicates Are Functional

In this section we describe how we checked that our declarative definitions of the tree traversals are *functional*: *i.e.*, for a given input tree they produce exactly one output traversal order. We show only the theorem that the in-order traversal definition is functional (Figure 2-20) because the others look almost identical.

These 'theorems' are theorems in the sense that we believe them to be true on the basis of Alloy's exhaustive analysis of all possible trees with up to ten nodes. There is a remote possibility that one of our traversal definitions may be non-functional on a tree of a larger size, in which case our 'theorems' would not be theorems. A proof technique without Alloy's finite bounds could be applied to these theorems.

The property as stated above glosses over a key subtlety that would confound most of these theorems if not addressed: the pre-order, post-order, and level-order traversals are defined over unordered trees. This lack of ordering in the inputs leads to non-determinism in the outputs. Non-deterministic output means that there may be more than one possible output for a given input (*i.e.*, more than one possible traversal for a given unordered tree).

We work around this wrinkle by imposing an ordering on the children of each node in the input tree, and constraining the traversals to be consistent with this ordering. Note that we are not imposing an ordering on the input tree as a whole: we are just ensuring the local property that the children of a node occur in some order. Under these conditions our traversal definitions are indeed functional: *i.e.*, for a given (now ordered) input tree, they produce exactly one output traversal order.

The effort to establish these theorems led to substantial revisions and elaborations of our initial versions of the definitions. Without the benefit of mechanical analysis we would have had no way to know what elaborations were necessary and our definitions would have remained under-constrained and hence non-functional. Such weaker, under-constrained, definitions may have created problems for future users of these definitions if they attempted to use these weaker definitions to establish invariants or post-conditions of their programs.

All of these theorems share a common structure, which can be seen in Figure 2-20. Each theorem establishes two traversals, t1 and t2, of the same tree. These traversals are constrained to be consistent with the arbitrary ordering placed on the tree. Recall that our predicates are defined over unordered trees, and so this arbitrary ordering is necessary for the property we are interested in to make sense. The formula Node = OrderedNode ensures that every node in the tree is part of this arbitrary ordering.

The theorem that our in-order traversal predicate is functional contains an additional constraint that restricts the tree to be binary. Whereas the other traversals are well-defined for trees of any arity, in-order traversal is only well-defined for binary trees.

Figure 2-20 Theorem: definition of in-order traversal is functional

```
assert InorderIsFunctional {
  no disjoint t1, t2 : Traversal | {
    t1.elems ≠ t2.elems
    inorder[ Root, left, right, t1.elems ]
    inorder[ Root, left, right, t2.elems ]
    consistentWithOrderedChildren[ t1.elems ]
    consistentWithOrderedChildren[ t2.elems ]
    Node = OrderedNode
    -- restrict to a binary tree
    Node.(orderedChildren.Node) in {0} +{1}
  }
}
check InorderIsFunctional for 10 but exactly 2 Traversal
```

2.3.3 Tree Traversal Predicates Differ

To further ensure that our predicate definitions do in fact correspond to their well-known intuitive meanings we also checked that they differ from each other. The key subtlety necessary to establish these properties is the size of the tree for which the definitions produce different traversal orderings. For example, on a tree of size one all definitions produce the same traversal order.

Figure 2-21 lists the Alloy theorem that for any tree with more than one node our definition of a pre-order traversal produces a different result than our definition of a post-order traversal does.

Figure 2-21 Theorem: definitions of pre-order and post-order traversals differ

```
assert PreNotEqualPost {
  no disjoint t1, t2 : Traversal | {
    -- equality
    t1.elems = t2.elems
    -- orders
    preorder[Root, children, t1.elems]
    postorder[Root, children, t2.elems]
    -- minimum size constraints
    #(Node) > 1
  }
}
check PreNotEqualPost for 10 but exactly 2 Traversal
```

There is a surprisingly large class of trees for which pre-order (*i.e.*, depth-first) and level-order (*i.e.*, breadth-first) traversals produce the same results. Figure 2-22 depicts eight example trees for which the pre-order and level-order traversals are identical. Figure 2-23 depicts two examples of small trees on which pre-order and level-order traversals differ.

The class of trees for which the pre-order and level-order traversals are the same are characterized by the nonTrivialTree predicate in Figure 2-24, as part of the theorem establishing that our definitions of pre-order and level-order produce, in most cases, different traversal orderings. This predicate says that there are two disjoint cousin nodes, x and y, in the tree that both have multiple children.

Checking these properties via mechanical analysis led to refinements of our initial definitions, generally improved our understanding of tree traversals, and increased our confidence that our declarative traversal definitions are consistent with the intuitions.

Figure 2-22 Example trees for which pre-order and level-order traversals are equivalent. Nodes are numbered in traversal order.

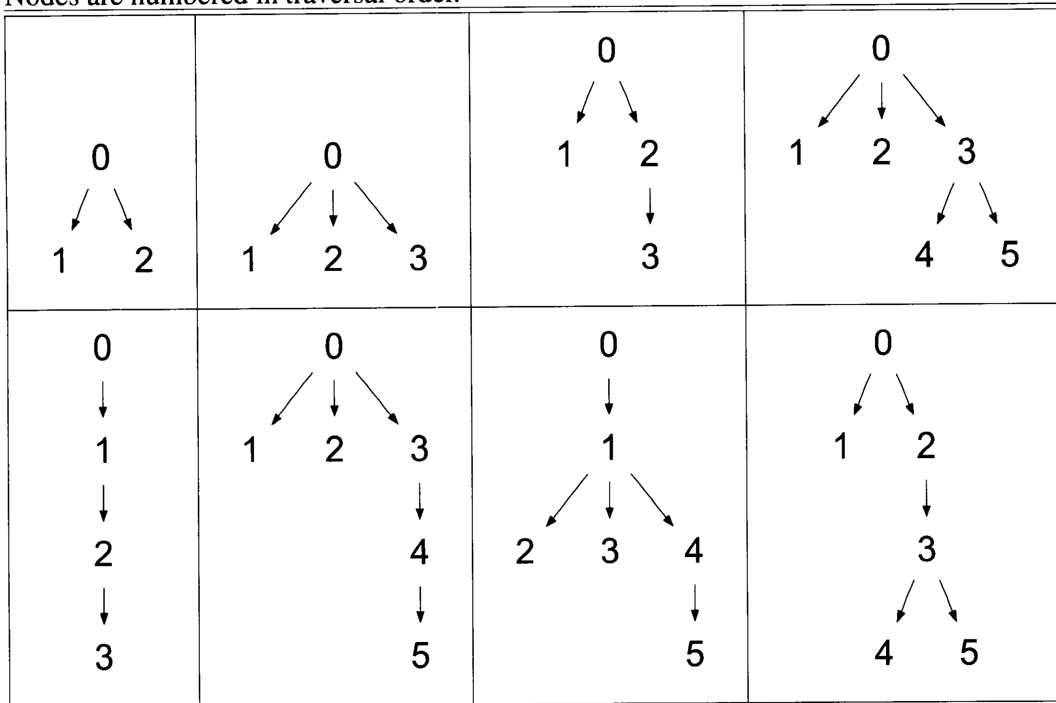


Figure 2-23 Example trees for which pre-order and level-order traversals differ. Nodes for which the orders differ are circled.

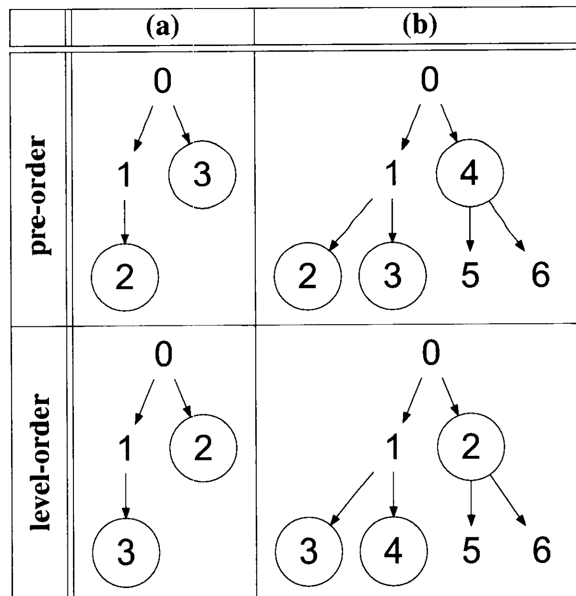


Figure 2-24 Theorem: definitions of pre-order and level-order traversals differ

```
assert PreNotEqualLevel {
  no disjoint t1, t2 : Traversal | {
    -- equality
    t1.elems = t2.elems
    -- orders
    preorder[Root, children, t1.elems]
    levelorder[Root, children, t2.elems]
    -- minimum size constraints
    nonTrivialTree
  }
}
pred nonTrivialTree {
  some disjoint x, y : Node | {
    #(x.children) > 1
    #(y.children) > 1
    (x->y) not in ^children
    (y->x) not in ^children
  }
}
check PreNotEqualLevel for 10 but exactly 2 Traversal
```

Chapter 3

Executing Abstraction Functions

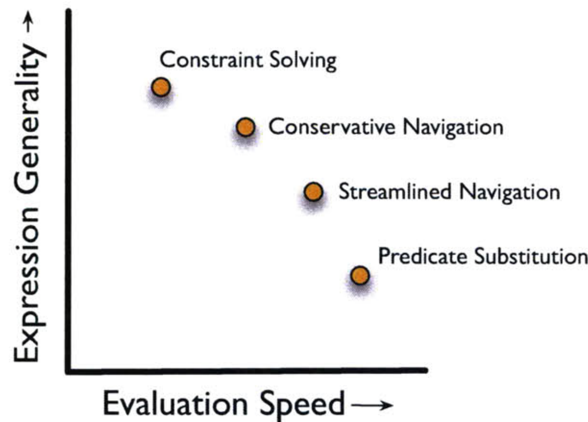
This chapter presents four techniques for the dynamic execution of set or sequence-valued field abstraction functions. The execution of a field abstraction function computes the abstract value for an object's specification field from that object's concrete representation. This abstract value is composed of concrete objects, already existent in the heap, in a mathematical set or sequence. We choose to represent this mathematical set or sequence structure at runtime with an iterator that returns the next element of the set (or sequence) on demand.

Our four techniques share the following general structure. A translation step produces an executable higher-order function from the field abstraction function written by the programmer in the declarative specification language JFSL [32, 119]. At runtime this higher-order function is applied to an object of interest in order to compute the abstract value of one of its specification fields from its concrete representation. The result of this function application is an iterator. This iterator yields the elements of the set or sequence on demand.

Figure 3-1 shows a Pareto Front of these techniques, illustrating the trade-off between expression generality and evaluation speed. At one extreme of the Pareto front, *constraint solving* works for any abstraction function but is the slowest to execute. Constraint solving is used for abstraction functions that written with inherently expensive language feature such as set comprehensions. At the other extreme of the Pareto front, *predicate substitution* runs the fastest but only works for the predicates defined above in Chapter 2.

There is a relatively simple subset of the JForge Specification language that we translate directly to imperative code. We call this subset *navigable expressions.* When navigating the heap it may be possible to encounter the same object via multiple paths. If the result of the navigation is intended to be a set then these duplicates must be detected at runtime and removed. However, if we know a priori that no duplicate objects will be encountered then we no longer need to do dynamic duplicate detection, and so can save both space and time during execution. To determine if it is safe to remove dynamic duplicate detection we perform a static analysis of the field abstraction function to produce a set of *optimization conditions* (desired invariants of the ADT). If these optimization conditions can be discharged by a verification system then we know that the same object will never be encountered twice during the navigation of the field abstraction function.

Figure 3-1 Pareto Front of abstraction function evaluation strategies: generality of expression versus runtime speed



This chapter explains the evaluation strategies according to the order in which they depend on each other. First we define navigable expressions and generally characterize how navigation is performed. Then we explain conservative navigation, streamlined navigation, and the optimization condition generation process that determines when streamlined navigation is applicable. Finally, predicate substitution and constraint solving are described.

The chapter concludes with a case study of the collections classes in the standard Java library. The main result of this case study is that our system can synthesize code for 25 of the 29 abstraction functions that were implemented imperatively by the original programmers. All of these abstraction functions were expressible with navigable expressions or the special predicates defined above in Chapter 2. Constraint solving was not required to execute any of the abstraction functions in this case study.

3.1 Navigable Expressions

Our two navigation strategies, streamlined navigation and conservative navigation, are applicable to only a syntactically-defined subset of the JForge Specification Language. We call this subset *navigable expressions*. The intuition of navigable expressions is that they start from the receiver object (this), traverse fields only in the forwards direction, and have a unary result. In this section we define navigable expressions; in the subsequent sections we describe streamlined navigation and conservative navigation.

An abstract syntax of navigable expressions is listed in Figure 3-2. The listing is given in the Haskell [105] programming language. This listing is part of a program that specifies the computations (described below) performed in evaluating navigable expressions. In Haskell, the data keyword introduces a new user-defined datatype, and the first word on each line after the equals sign introduces a new constructor function for that datatype.

This abstract syntax of navigable expressions has five important features:

1. Productions are classified according to the arity of their output as either unary or binary. The abstract syntax of Alloy (or JFSL) is usually presented in an arity-independent manner. However, here we are defining a subset of JFSL that is amenable to particular computations, and this division is useful for describing these computations.

2. The root production of any AST is always unary: in other words, a navigable expression always computes a unary result.

3. All of the unary productions with binary children involve a join: UnaryJoin, JoinClosure, ReflexiveJoinClosure.

4. The leaves of a navigable expression are always fields or the distinguished singletons this and null.

5. The productions for transitive closure are coupled with a join: *e.g.*, JoinClosure and ReflexiveJoinClosure. This coupling is not the case in the standard Alloy grammar because join and closure are considered as separate operators [52]. However it is the case in the JFSL [32, 119] implementation because it simplifies certain analyses.

Figure 3-2 Abstract syntax of navigable expressions (in Haskell)

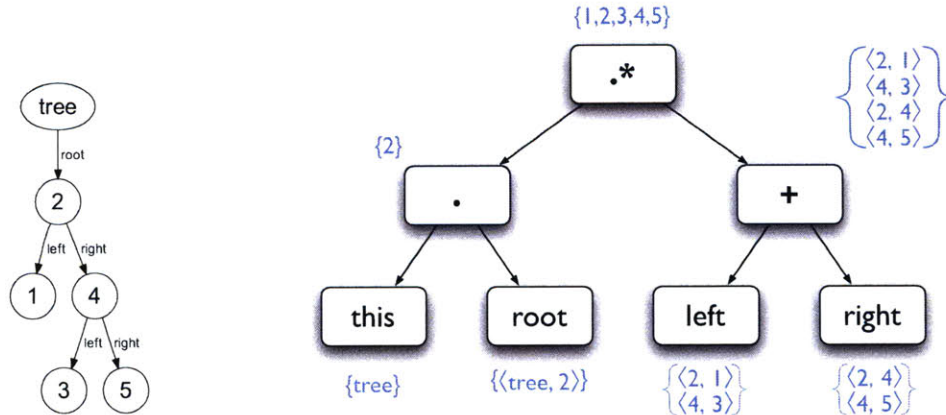
```
data NavigableExpr  = UnaryExpr

data UnaryExpr      = ThisExpr                                          this
                    | NullExpr                                          null
                    | UnaryUnion UnaryExpr UnaryExpr                    u+u
                    | UnaryDiff UnaryExpr UnaryExpr                     u-u
                    | UnaryJoin UnaryExpr BinaryExpr                    u.b
                    | JoinClosure UnaryExpr BinaryExpr                  u.^b
                    | ReflexiveJoinClosure UnaryExpr BinaryExpr         u.*b

data BinaryExpr     = FieldExpr String                                  f
                    | BinaryUnion BinaryExpr BinaryExpr                 b+b
                    | BinaryDiff BinaryExpr BinaryExpr                  b-b
```

Recall our ongoing example of a set of integers represented by a binary search tree, introduced in Figure 1-7, with abstraction function this.root.*(left+right). Figure 3-3b shows the AST for this navigable expression. The leaves are concrete field references (root, left, right) and the distinguished name this. The root node is a reflexive-join-closure (.*). Recall that join is syntactically represented by a dot.

Figure 3-3a shows an example instance of a our binary search tree with five nodes, numbered 1 through 5. Figure 3-3b shows the abstract syntax tree of our example abstraction function annotated with the denotation of each syntax-node in the context of the example

data structure in Figure 3-3a. We see, for example, that the root field of Figure 3-3a is represented as a binary relation with the tuple {⟨tree,2⟩} in Figure 3-3b. We also see that the union of the left and right relations results in the binary relation {⟨2,1⟩, ⟨4,3⟩, ⟨2,4⟩, ⟨4,5⟩}. Finally, at the root node of the AST (a reflexive-join-closure), we see the result set {1, 2, 3, 4, 5}.

Figure 3-3 Denotation of this.root.*(left+right) on simple example tree data structure.



(a) Example tree data structure in heap.

(b) Abstraction function AST annotated with denotation of example at left.

3.2 Syntax-Directed Execution of Abstraction Functions

Our two navigation execution strategies (streamlined and conservative) are produced by a top-down syntax-directed translation of the text of the (navigable) field abstraction function. These translations do some restructing of the AST. Let us name each node of the abstraction function AST a_i, and each corresponding part of the executable abstraction function α_i.

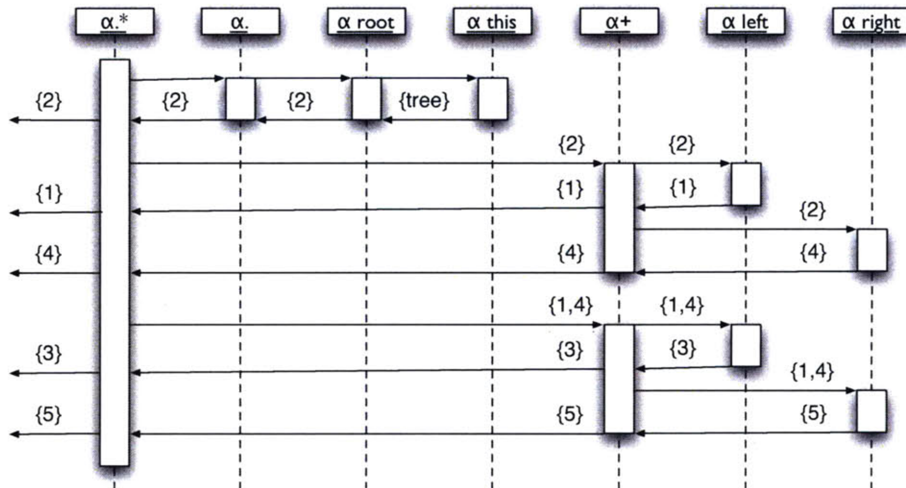
The key insight behind our translations is that, because of the grammatical properties of navigable expressions, the result of each α_i can be represented as a unary set — even if that α_i represents an expression with a binary result. The reasoning is as follows: the only binary leaf nodes are fields; the root is always unary; and there is always a join on the path between the field leaf and the unary root. Therefore, from a computational standpoint, we can push the joins down into the leaves. At runtime then the only binary tuples are the referencing relationships already extant in the heap.

In our example, the computed output of α_{root} is not the binary tuple ⟨tree,2⟩ (which is what a_{root} denotes) but, rather, simply the unary 2 (by which we mean node #2 in the example of Figure 3-3a). What then is the role of $\alpha_{.}$? In the runtime computation it is a no-op: its work is done by α_{root}. The important work related to $\alpha_{.}$ happens during the translation where the

AST is restructured from $a_{.}(a_{this}, a_{root})$ to $\alpha_{root}(\alpha_{this})$. This translation and computation are described formally below using the Haskell programming language [105].

Figure 3-4 shows a sequence chart of the execution of the abstraction function from our ongoing example (this.root.*(left+right)) on the example data structure of Figure 3-3a. In this chart each α_i is lazy and returns partial results as they are computed. The final result is the set {2, 1, 4, 3, 5} — which is the result of a level-order traversal of the example tree data structure. Note that in choosing to write a navigable expression that includes transitive closure for an abstraction function the programmer is leaving the traversal order unspecified. The translator is free to employ whatever traversal strategy it chooses; our translator usually performs a level-order traversal.

Figure 3-4 Sequence chart of evaluation of example α on example data structure.



In our actual implementation each α_i is realized as a Java iterator. This implementation choice gives us some flexibility: the α_i may be 'lazy' or 'demand-driven', which opens the door for some potential performance improvements.

In subsequent sections we construct some formal models of our translation and mechanically check that they are consistent with the semantics.

3.3 Conservative Navigation

Conservative navigation is applicable to any navigable expression. Streamlined navigation, by contrast, is applicable to only a semantic subset of navigable expressions. We use the Haskell programming language [105] to describe both conservative navigation (here) and streamlined navigation (below).

Figure 3-5 introduces five type definitions that are used by the functions below. First, Obj is defined as an alias for the Haskell built-in type String. Second, the type UnaryRelation

is defined to be a set of Objs. Third, a BinaryRelation is defined as a set of pairs of Objs. Fourth, a Field is defined as a pair of a string (that names the field) and the binary relation (that represents the mappings of the field). Finally, a Heap is defined as a pair of an object and a list of fields. The fields represent all of the referencing relations in the heap. The distinguished object in a heap pair represents the receiver object: *i.e.*, this; the object on which we are going to evaluate the abstraction function.

Figure 3-5 Haskell description of conservative navigation (type definitions)

```
type Obj            = String
type UnaryRelation  = Set.Set Obj
type BinaryRelation = Set.Set (Obj, Obj)
type Field          = (String, BinaryRelation)
type Heap           = (Obj, [Field])
```

Figure 3-6 describes conservative navigation for the unary productions of our navigable expression grammar. The function being defined here is named tu which stands for 'translate unary'. The inputs to tu are a UnaryExpr and a Heap, and the output is a UnaryRelation, which is the result of the evaluation of the unary expression with respect to the given heap. Functions are defined piecewise in Haskell, with pattern matching on the arguments used to differentiate the different cases. We will now discuss the various cases for unary productions.

The simplest case is when the programmer writes 'this' in the abstraction function: the computation returns a singleton set with the distinguished receiver object. The computations for a unary union and unary difference are simply set union and set difference, respectively.

Figure 3-6 Haskell description of conservative navigation (unary productions)

```
tu :: UnaryExpr → Heap → UnaryRelation
tu ThisExpr (this, _)          = Set.fromList [this]
tu (UnaryUnion lhs rhs) h     = (tu lhs h) `Set.union` (tu rhs h)
tu (UnaryDiff lhs rhs) h      = (tu lhs h) `Set.difference` (tu rhs h)
tu (UnaryJoin lhs rhs) h      = tb rhs h (tu lhs h)
tu (JoinClosure lhs rhs) h    = levelorder rhs h (tu lhs h)
tu (ReflexiveJoinClosure lhs rhs) h =
      (tu lhs h) `Set.union` (tu (JoinClosure lhs rhs) h)
```

The computations for unary join, join-closure, and reflexive join-closure all make use of helper functions that are listed below in Figure 3-7. Recall that what we refer to as a 'unary join' is an expression like u.b where u is a unary relation and b is binary relation. We call such an expression a unary join because its output is unary. The function unaryJoin in Figure 3-7 selects (filter) each pair out of the binary relation (rhs) where the first element of the pair is in the unary relation (lhs); it then returns a set of the second elements in these pairs (map snd).

The computation for join-closure composes the computation for unary join and the helper

function levelorder from Figure 3-7; levelorder performs a level-order, or breadth-first, traversal. The tree that it traverses is created dynamically by repeatedly applying unary join to its own result. Any other traversal, such as pre-order or post-order, would also suffice. When the programmer writes a JFSL expression with a join-closure they are leaving the traversal unspecified. The computation for reflexive join-closure (Figure 3-6) simply unions the result of evaluating its left-hand argument to the result of computing a regular join-closure.

Figure 3-7 Haskell description of conservative navigation (helper functions)

```
doJoin :: UnaryRelation → BinaryRelation → UnaryRelation
doJoin lhs rhs = Set.map snd (Set.filter ((`Set.member` lhs ) . fst) rhs)

levelorder :: BinaryExpr → Heap → UnaryRelation → UnaryRelation
levelorder _ _ set | Set.null set = Set.empty
levelorder rhs h u = let x = (tb rhs h u) in x `Set.union` levelorder rhs h x
```

Figure 3-8 lists the computations for binary productions in our grammar of navigable expressions. The function being described is tb, which stands for 'translate binary'. One might expect the type of tb to be BinaryExpr → Heap → BinaryRelation, following the structure of the type of tu. However, as discussed above, the type of tb is actually BinaryExpr → Heap → UnaryRelation → UnaryRelation, where the first UnaryRelation is the input and the second UnaryRelation is the output. This type signature is the manifestation of our key insight that the result type of every navigable α_i can be unary.

In Figure 3-8 we see that binary union and binary difference are, like unary union and unary difference, defined in terms of set union and set difference. This is possible because of our key insight that the result of a semantically-binary expression can in computation be unary. As in Alloy, field dereference is considered as a unary join.

Figure 3-8 Haskell description of conservative navigation (binary productions)

```
tb :: BinaryExpr → Heap → UnaryRelation → UnaryRelation
tb (BinaryUnion lhs rhs) h u = (tb lhs h u) `Set.union` (tb rhs h u)
tb (BinaryDiff lhs rhs) h u = (tb lhs h u) `Set.difference` (tb rhs h u)
tb (FieldExpr f) (_, fields) u = doJoin u (fromJust (lookup f fields))
```

In the context of data structure repair Demsky et al. [30] perform two optimizations that have the same motivation as what we do here: *relation elimination*, like our restructuring of the translate binary function, aims to avoid instantiating tuples for (binary) relations; *set elimination* aims to avoid instantiating sets for computed results, as our streamlined navigation described next does. They found that these optimizations, combined with a third optimization only relevant in their data structure repair context, sped up execution time by a factor of 800. The technical details of our work differs: our specification language is different, the analyses we perform on it are different, and our software engineering objectives are different. However, we share the common basic point that instantiating relations and sets for computed values is expensive and is to be avoided if possible.

3.4 Two Navigations With Sequences

We have observed that there are two kinds of expressions involving sequences that programmers commonly want to include in their abstraction functions. One is to extract the set of elements from the sequence with an expression of the form int.s, where s is a sequence, and int names the set of all signed 32-bit integers. The other is to use a sequence as an index to another field with an expression of the form seq.field, where seq is a sequence, and field is a field. Both of these kinds of expressions are easily navigable.

Figure 3-9 shows the extra line we added to the definition of the translate-unary (tu) function to handle expressions of the form int.s, which we call SeqSetJoin expressions. The computation is simply to compute the set of the elements in the sequence. We here model a sequence as a Haskell list of objects: [Obj]. The elements in the sequence are computed by the translate-sequence (ts) function, described next.

Figure 3-9 The set of elements in a sequence

```
tu (SeqSetJoin i rhs) h = Set.fromList (ts rhs h)
```

Figure 3-10 lists the translate-sequence (ts) function, which includes the line evaluating expressions of the form seq.field. We name these SeqMapJoin expressions. A sequence expression is either an array expression or a SeqMapJoin expression. The evaluation of SeqMapJoin makes us of a utility function named unaryJoin2. This function is simply a wrapper for the unaryJoin function seen above. The wrapper packs up the arguments to unaryJoin appropriately and unpacks the result. We know that the result of this call to unaryJoin will result in a singleton because a Java field can refer to only a single object or null. The wrapper uses the Set.findMin function to extract the single object from the result set of unaryJoin.

Figure 3-10 Joining a sequence to a field

```
ts :: SeqExpr → Heap → [Obj]
ts (ArrayExpr a) h = a
ts (SeqMapJoin s f) h = map (unaryJoin2 f h) (ts s h)

unaryJoin2 :: FieldName → Heap → Obj → Obj
unaryJoin2 f (_, fields) x =
    Set.findMin ( unaryJoin (Set.singleton x) (fromJust (lookup f fields)) )
```

Allowing expressions of the form seq.field also requires us to extend the definition of a navigable expression to be either a unary expression or a sequence expression. The addition of this extra case does not change the reasoning behind navigating unary expressions.

3.5 Streamlined Navigation

Streamlined navigation may be used to execute navigable expressions that are applied only to a tree-like subset of the heap. We formalize the conditions under which streamlined navigation is applicable below. The process for determining whether streamlined navigation may be used is as follows:

1. A static analysis of the field abstraction function, described below, derives a specific set of *optimization conditions* (desired invariants).
2. A program verification system attempts to discharge the optimization conditions via a static analysis of the program. We use the JForge [32, 119] verification system.
3. If the optimization conditions hold, then streamlined navigation may be used; otherwise conservative navigation must be used.

An intuition about the difference between streamlined and conservative navigation is that streamlined navigation performs lazy computation with lists, whereas conservative navigation performs eager computation with sets. For example, if the abstraction function contains a union operator (+), streamlined navigation will perform list concatenation whereas conservative navigation will perform set union.

We use the Haskell programming language [105] to describe streamlined navigation, as with the conservative navigation described. Figure 3-11 lists the type definitions used in our description of streamlined navigation. They are almost the same as the type definitions given above in Figure 3-5 for conservative navigation, except now UnaryRelation is defined to be a list of objects (rather than a set of objects) and BinaryRelation is defined to be a list of pairs (rather than a set of pairs).

Figure 3-11 Haskell description of streamlined navigation (type definitions)

```
type Obj            = String
type UnaryRelation  = [Obj]
type BinaryRelation = [(Obj, Obj)]
type Field          = (String, BinaryRelation)
type Heap           = (Obj, [Field])
```

Figure 3-12 describes the streamlined translate-unary (tu) function. As compared to the conservative tu above in Figure 3-6, the set union operations have been replaced with list concatenation. (In Haskell, the ++ operator performs list concatenation.) For the unary difference production the right-hand side is simply dropped: *i.e.*, the streamlined form of x-y is just x.

Figure 3-12 Haskell description of streamlined navigation (unary productions)

```
tu :: UnaryExpr → Heap → UnaryRelation
tu ThisExpr (this , _)        = [this]
tu (UnaryUnion lhs rhs) h     = (tu lhs h) ++ (tu rhs h)
tu (UnaryDiff lhs rhs) h      = tu lhs h
tu (UnaryJoin lhs rhs) h      = tb rhs h (tu lhs h)
tu (JoinClosure lhs rhs) h    = levelorder rhs h (tu lhs h)
tu (ReflexiveJoinClosure lhs rhs) h =
      (tu lhs h) ++ (tu (JoinClosure lhs rhs) h)
```

Similarly, the helper functions unaryJoin and levelorder have been re-defined in terms of lists and concatentation rather than sets and union: contrast Figure 3-13 with Figure 3-7.

Figure 3-13 Haskell description of streamlined navigation (helper functions)

```
unaryJoin :: UnaryRelation → BinaryRelation → UnaryRelation
unaryJoin lhs rhs = map snd (filter (('elem' lhs ) . fst) rhs)

levelorder :: BinaryExpr → Heap → UnaryRelation → UnaryRelation
levelorder _ _ [] = []
levelorder rhs h u = let x = (tb rhs h u) in x ++ levelorder rhs h x
```

Figure 3-14 lists the streamlined translate-binary (tb) function. As above, set union has been replaced by list concatenation (++), and set difference is achieved by dropping the right-hand side. Otherwise the listing in Figure 3-14 is quite similar to the conservative definition listed above in Figure 3-8.

Figure 3-14 Haskell description of streamlined navigation (binary productions)

```
tb :: BinaryExpr → Heap → UnaryRelation → UnaryRelation
tb (FieldExpr f) (_, fields) u = unaryJoin u (fromJust (lookup f fields))
tb (BinaryUnion lhs rhs) h u = (tb lhs h u) ++ (tb rhs h u)
tb (BinaryDiff lhs rhs) h u = tb lhs h u
```

Obviously streamlined navigation does not produce the same results as conservative navigation for all inputs: list concatenation and set union are not generally equivalent operations. But consider the case of two sets with no common elements: if we represent these sets as lists, then the concatenation of the lists is equivalent to the union of the sets. We can only substitute list concatenation (streamlined navigation) for set union (conservative navigation) if we know *a priori* that the two inputs have no intersection. We call this 'no intersection' criterion an *optimization condition*, and the process of generating these desired heap invariants from field abstraction functions is described next.

3.6 Optimization Condition Generation

This section formalizes the conditions under which streamlined navigation is semantically correct: *i.e.*, formalizes the notion of a 'tree-like' portion of the heap. A static analysis of the field abstraction function produces a set of *optimization conditions*. An optimization condition is a desired heap invariant. If these desired invariants are in fact true of the program, then the navigation described by the field abstraction function is tree-like and streamlined navigation may be used.

Like our translation strategies for navigable expressions, optimization condition generation follows a syntax-directed approach. The optimization conditions generated for each production are listed, as an Alloy model, in Figure 3-15. In these Alloy models every expression atom is constrained to be part of *the* abstraction function AST. We never compare two ASTs, and so the AST itself is implicit.

The Alloy model in Figure 3-15 makes use of one signature, JObject, and two relations, udenotes and bdenotes. The JObject signature models Java objects in an execution of the program. The udenotes and bdenotes relations model the denotation of unary and binary productions in the abstraction function. The values of these denotation relations are given by the semantics of navigable expressions, which are essentially Alloy semantics.

Many of the optimization conditions are fairly simple. For example, the optimization condition for unary difference is that there is no intersection between the denotation of the left-hand side and the denotation of the right-hand side: **no** e.lhs.udenotes & e.rhs.udenotes. The optimization condition for unary union is exactly the same. These conditions are also essentially the same in the binary case.

The optimization condition for unary join is slightly more complicated. It essentially says that performing the join will not lead to the same target object via two different paths. The starting point of those paths are o1 and o2, from the denotation of the LHS. The ending point of those paths are found by joining the starting points to the denotation of the RHS.

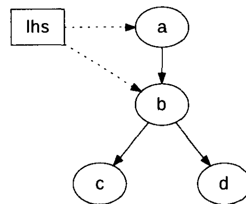
The optimization conditions for join-closure and join-reflexive-closure are the same, and are rather involved. Again, the intuition is to ensure that the navigation is tree-like. A join-closure expression has both a right-hand side, which names the structure to be traversed, and a left-hand side, which names the starting points for that traversal. The first condition says that there is no object o that has two incoming edges (one from x and one from y). The second condition says that there is no cycle in the right-hand side relation that can be reached from a starting point in the left-hand side set.

The first two conditions establish that the relation on the right-hand side is sufficiently tree-like. The third condition checks that the left-hand side enters that tree appropriately. Consider the tree a, b, c, d in Figure 3-16. If we start traversing at a we visit all four nodes. If we subsequently start traversing at b we then re-visit nodes b, c, d. The starting-points a and b are hierarchically related to each other. The third condition ensures that no starting-points for the traversal (*i.e.*, objects in the denotation of the left-hand side) are hierarchically related. We learned of the necessity for this third condition through the formalization and mechanical analysis of optimization condition generation.

Figure 3-15 Optimization condition generation for optimized navigation

```
pred OptimizationConditions {
  all e : BinaryExpr | BinaryOC[e]
  all e : UnaryExpr−GoalExpr | UnaryOC[e]
}

pred UnaryOC [ e : one UnaryExpr ] {
  e in UnaryDiffExpr ⇒ no e.lhs.udenotes & e.rhs.udenotes
  e in UnaryUnionExpr ⇒ no e.lhs.udenotes & e.rhs.udenotes
  e in UnaryJoinExpr ⇒ no disjoint o1, o2 : e.lhs.udenotes | let r=e.rhs.bdenotes | some o1.r & o2.r
  e in JoinClosureExpr ⇒ JoinClosureOC[e]
  e in JoinReflexiveClosureExpr ⇒ JoinClosureOC[e]
}

pred BinaryOC [ e : one BinaryExpr ] {
  e in BinaryUnionExpr ⇒ no e.lhs.bdenotes & e.rhs.bdenotes
  e in BinaryDiffExpr ⇒ no e.lhs.bdenotes & e.rhs.bdenotes
  e in BinaryJoinExpr ⇒ (no a, b : JObject | {
    (a→b) in e.bdenotes
    some disjoint x, y : JObject | {
      ((a→x) +(a→y)) in e.lhs.bdenotes
      ((x→b) +(y→b)) in e.rhs.bdenotes }})
}

pred JoinClosureOC[ e : one AbstractJoinClosureExpr ] {
  (no o : JObject | { some disjoint x, y : JObject | {
    −− no object (o) reachable from two others (x, y)
    ((x→o)+(y→o)) in (e.rhs.bdenotes)
    (x+y) in (e.lhs.udenotes) +(e.lhs.udenotes).^(e.rhs.bdenotes)
  }}) and ( no o : JObject | {
    −− no cycle that is reachable from the LHS
    (o→o) in ^(e.rhs.bdenotes)
    some x : e.lhs.udenotes | (x→o) in ^(e.rhs.bdenotes)
  }) and (
    −− can't enter the tree at two hierarchically−related positions
    no disjoint p, q : e.lhs.udenotes | (p→q) in ^(e.rhs.bdenotes)
  )
}
```

Figure 3-16 Multiple navigation entry points into a tree data structure



3.7 Soundness of Optimization Condition Generation

This section establishes the soundness of the optimization condition generation described above: that is, if the optimization conditions described above hold, then streamlined navigation will produce results that are consistent with the semantics. We establish this soundness result with a number of Alloy models:

- the syntax of navigable expressions (seen above, in Haskell, in Figure 3-2);
- the semantics of navigable expressions (regular Alloy semantics; not shown);
- the optimization condition generation (seen above in Figure 3-15);
- a *direct* model of the streamlined computations (discussed below);
- an *indirect* model of the streamlined computations (discussed below).

The analysis searches for a field abstraction function and a program heap (constrained by the optimization conditions for the given field abstraction function) for which streamlined navigation is inconsistent with the semantics. No counterexample is found, giving support to the claim that streamlined navigation produces semantically correct results.

We have expressed the streamlined computations in four different ways during the course of this work: our running implementation is written as higher-order functions in Java; above they were described using the Haskell programming language in Figures 3-11, 3-13, 3-12, and 3-14; in this section we have both the *indirect* and *direct* Alloy models. Our objective in this section is to confirm that our running implementation is consistent with the semantics of navigable expressions. Each different form of the streamlined computations moves one small step along this continuum:

- The direct model is conceptually similar to the semantics because both explicitly represent the results associated with each node of the field abstraction function AST. We have mechanically verified the consistency of the direct model and the semantics of navigable expressions using the Alloy Analyzer.
- The indirect model is closer to the code because it uses a unary representation for the output of AST nodes that semantically have a binary output. We have mechanically verified the consistency of the indirect model with the direct model and the semantics using the Alloy Analyzer.
- The Haskell code is the most clear and concise expression of the streamlined computations. We used it as a conceptual specification both for the Alloy models and for the Java implementation. The consistency of the Haskell code with the Alloy models and the Java implementation was checked manually.
- The Java implementation has been tested with a variety of abstraction functions and program states, including hand-written tests by us and by independent third parties, and machine-generated tests from the Randoop tool [93, 94].

Through the course of these verification steps we have rectified a variety of issues with our understanding and expression of the streamlined computations. The most interesting such issue was the multiple-entry scenario depicted in Figure 3-16 above. Without the benefit of the Alloy Analyzer's mechanical analysis we would have failed to include the third optimization condition for join-closure that prevents this scenario (Figure 3-15).

We now explore the formalizations of the direct and indirect models and their consistency with each other and with the semantics. Figure 3-17 shows the Alloy predicate that claims the direct model is consistent with the semantics if the optimization conditions hold. This predicate checks that, for each node in the abstraction function AST, the denotation and the computation are equivalent if the optimization conditions hold. This check is broken down into two separate helper predicates (the text of which is not shown here), one for unary nodes (unaryComputesEqualsDenotes), and one for binary nodes (binaryComputesEqualsDenotes).

Figure 3-17 Theorem: the optimization conditions imply that the computation is equal to the denotation for the direct model of streamlined navigation

```
pred ComputationIsOk {
  OptimizationConditions ⇒ (
    (all e : UnaryExpr | unaryComputesEqualsDenotes[e]) and
    (all e : BinaryExpr | binaryComputesEqualsDenotes[e]))
}
```

Figure 3-18 shows the Alloy predicate that claims the indirect model is consistent with the semantics if the optimization conditions hold. In this case the check is only for the unary nodes of the abstraction function AST because the computed results for the binary nodes are not explicitly represented in the indirect model. Instead, the partial results at each time-step are modelled for the binary nodes (these time-steps are not shown in the listings here). These partial results cannot (easily) be compared against the denotations of the nodes. Note, however, that the root node of a navigable expression is always unary, and so the check in Figure 3-18 always includes the final output of the navigation.

Figure 3-18 Theorem: the optimization conditions imply that the computation is equal to the denotation for the indirect model of streamlined navigation

```
pred ComputationIsOk { OptimizationConditions ⇒ (all e : UnaryExpr | unaryComputesEqualsDenotes[e]) }
```

Figure 3-19 shows the Alloy predicate that checks that the direct and indirect models of streamlined computation produce the same results. This cross-check only examines the root node of the AST: *i.e.*, it only checks that the two models produce the same ultimate result (it does not check the partial results at each sub-node of the abstraction function AST).

Figure 3-19 Theorem: direct and indirect models of streamlined navigation are equivalent

```
assert crosscheck { OptimizationConditions ⇒ (GoalExpr.ucomputes = GoalExpr.ustream) }
```

Let us stand back for a minute to look at what the Alloy Analyzer is verifying here. First, the Analyzer constructs every possible navigable abstraction function AST (within some finite bound) according to (an Alloy version of) the grammar shown above in Figure 3-2. Second, it constructs every possible Java program heap (according to our model of the Java heap, and within some finite bound). Then it evaluates (1) the denotation of this abstraction function AST with respect to this Java program heap according to the semantics; and (2) the streamlined computation of this abstraction function AST with respect to this Java program heap according to either the direct or indirect model. Finally, it checks whether the denotation and the computation are equivalent.

We have run these checks with a variety of different finite bounds. The smallest bound was for abstraction function ASTs with up to five nodes and Java program heaps with up to five objects. This bound can be verified in one or two seconds for both the direct and indirect models. We have also verified for abstraction function ASTs up to eight nodes and Java program heaps up to four objects. These bounds take time between twenty seconds and twenty minutes depending on how the allowance of eight AST nodes is divided between different kinds of AST nodes and whether the direct (faster) or indirect (slower) model is being verified.

3.8 Predicate Substitution

The *predicate substitution* execution strategy involves substituting pre-defined imperative code for the library of predicates introduced above in Chapter 2. These are well-understood sequence operations and tree-traversals.

Recall our ongoing example of an integer set represented as a binary search tree. In Figure 2-9 we saw an example of a specification field nodeseq defined in terms of the inorder predicate, which we reproduce here in Figure 3-20. At runtime, the system uses the iterative definition of an in-order binary tree traversal to evaluate the inorder predicate; pseudo-code for this imperative in-order traversal is listed in Figure 3-21. Our actual implementation of in-order traversal is structured as a Java iterator, and looks somewhat more like the listing given above in Figure 1-8.

Figure 3-20 Ongoing example: use of inorder predicate to define nodeseq specification field

```
@SpecField("nodeseq : seq Node |
            inorder(this.root, Node@left, Node@right, this.nodeseq)")
class BinarySearchTree { Node root; }
class Node { Node left, right; int value; }
```

Figure 3-21 Iterative definition of in-order traversal predicate [115]

```
inorder(root, left, right, result)
    stack := empty Stack
    node = root
    while (!(node == null && stack.isEmpty()))
        while (node != null)
            stack.push(node)
            node = node.left
        assert !stack.isEmpty()
        n = stack.pop()
        node = n.right
        result.add(n)
```

The arguments passed to the inorder predicate in Figure 3-20 are handled in different ways. The last argument, **this**.nodeseq, names the specification field that is being constrained by this predicate. The middle two arguments, Node@left and Node@right, name the fields that are to be used for fetching the left and right children, respectively, of a node (this is the syntax used by JFSL [32, 119] for what might be written in Alloy as a domain restriction such as Node <: left or Node <: right). The first argument, **this**.root, is a navigable expression that is evaluated using the navigation techniques discussed above.

Substitution for the other predicates described in above in Chapter 2 works in much the same way: a well-known imperative implementation is used in place of the logical definition; arguments are either the specification field being constrained, expressions that name fields to be used in the evaluation, or navigable expressions that name particular objects to be used as starting points.

3.9 Constraint Solving

Constraint solving is the execution strategy of last resort, to be used when the field abstraction function does not conform to the syntactic restrictions of predicate substitution nor navigable expressions. It supports the entire syntax of the JForge Specification Language, and in almost all cases computes the results that the programmer expects (an exceptional case is described below). The steps involved in constraint solving are as follows:

1. **Serialize.** The visible heap is serialized into a (large) logical formula. The 'visible heap' is the objects reachable by field references starting from the receiver object and the method parameters. Following Jackson [51], each field in the program is modelled as a binary relation, and each individual field reference is modelled as a tuple in the appropriate such relation.

2. **Solve.** Combine the formula representing the visible heap with the formula of the field abstraction function, and invoke a constraint solver to search for a solution.

3. **Return.** The logical solution returned by the constraint solver names a set of program objects by their representative logical variables. From these logical variables, the corresponding program objects must be located in the heap and returned to the running program.

The technology stack that performs these operations was described above in §1.7.

Figure 3-22 gives an example of an abstraction function that requires constraint solving. The class ArrayPointSet uses an array to store a set of Point objects. The specification field points is the conceptual set of points that are stored in the array, and is computed via a navigable abstraction function. The specification field brightest refers to the brightest point in the set (there may be multiple brightest points that all have the same brightness value). This specification field is computed via a set comprehension, and that requires constraint solving in our system.

Figure 3-22 Example abstraction function that requires constraint solving (ArrayPointSet)

```
@SpecField({
    "points : set Point | this.points = int.(this.a.elems)-null",
    "brightest : set Point | { p : this.points |
        no q : this.points | q!=p => q.brightness > p.brightness}"})
@Invariant("no disjoint p, q : this.points | p.x == q.x && p.y == q.y")
class ArrayPointSet { Point[] a; }
class Point { int x, y, brightness; }
```

Our approach to constraint solving evaluates formulæ with respect to the portion of the heap that is visible from the receiver object. This approach may not always produce the results the programmer intends: in some circumstances the programmer may expect the formula to be evaluated against the entire heap or even against potential objects that have not been instantiated. In such cases our approach to constraint solving will produce results that may surprise the programmer.

Consider the example RangePointSet class in Figure 3-23. This class describes a rectangular area in a Cartesian coordinate space. Its concrete representation is just the minimim (bottom left) and maximum (top right) points of the rectangle. This example was inspired by the RangeSequence class written by Emina Torlak as part of the Kodkod [110, 111] relational model finder. The specification field points is computed via a set comprehension, which requires constraint solving in our system. However, the programmer clearly intends to this specification field to include every integer point in the rectangle — even if those objects have never been instantiated by the program. In our system the constraint solver will only see the minimum and maximum points, and so the abstraction function might as well be this.points = this.min + this.max.

Figure 3-23 Example abstraction function for which constraint solving would not produce the expected result (RangePointSet)

```
@SpecField("points : set Point | this.points = { p : Point |
            this.min.x <= p.x && this.min.y <= p.y &&
            this.max.x >= p.x && this.max.y >= p.y }")
@Invariant("this.min.x <= this.max.x && this.min.y <= this.max.y")
class RangePointSet { Point min, max; }
class Point { int x, y; }
```

3.10 Performance Characterization

This section characterizes the performance of our field abstraction function execution strategies on our ongoing binary search tree example. This characterization has two purposes: first, to show that it is worth the effort to apply more specialized execution strategies when possible; second, to show the circumstances under which these strategies are competitive with hand-written code.

We compare our execution strategies against two hand-written implementations, one using normal field access and the other using reflective field access. Our current implementations of the execution strategies use reflection for field access. This choice gives us greater flexibility and modularity in our implementation, at the cost of runtime performance. The hand-written implementations use the common recursive form of in-order traversal, which was listed above for reference in Figure 2-15.

We use two different syntactic forms of effectively the same abstraction function because the predicate substitution strategy and the other strategies operate on different subsets of the JForge Specification Language. For most strategies we use the navigable expression this.root.*(left+right) that was originally introduced in Figure 1-7. For the predicate substitution strategy we use the predicate inorder(this.root, Node@left, Node@right, this.nodeseq), which was introduced in Figure 2-9. These two abstraction functions are not logically equivalent because the navigable expression does not specify the traversal order. However, in practice our implementation uses in-order traversal for this example, and so the results at runtime are the same.

Figure 3-24 plots the performance times of these strategies on red/black trees of varying sizes. Red/black trees are a kind of balanced binary search tree. Figure 3-24a examines trees of up to five thousand nodes and presents the individual data points. Trees of this small size are traversed extremely quickly, and so the times are quantized at millisecond boundaries (which is the smallest unit of time we can measure inside a Java virtual machine). Figure 3-24b examines trees of up to one million nodes and uses lines to connect the individual data points so that the trend is clear.

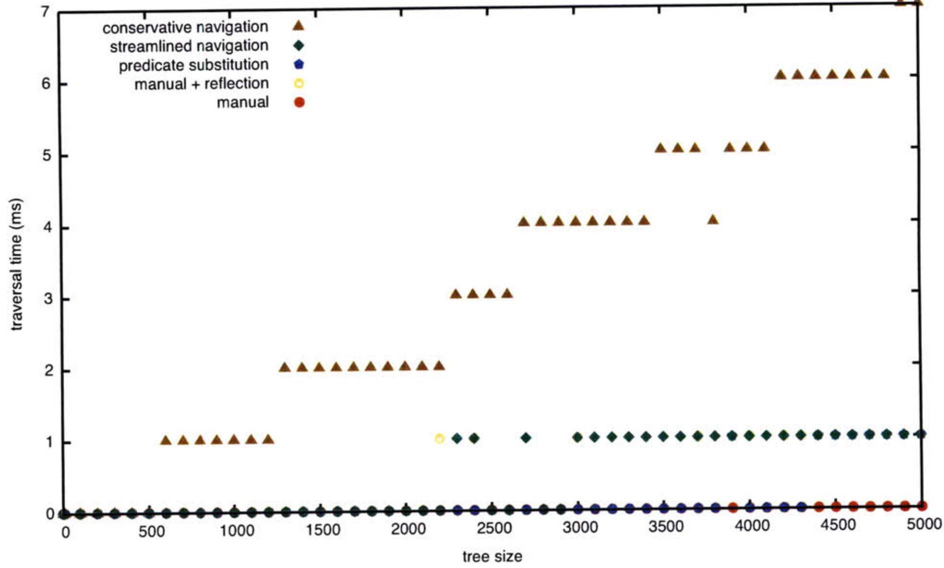
The constraint solving strategy has been omitted from the data in Figure 3-24 because that strategy is orders of magnitude slower than the others and does not scale to even medium-sized trees. For example, the maximum time recorded on Figure 3-24a for the other strate-

gies on trees of up to 5000 nodes is 7ms; the maximum time recorded on Figure 3-24b for the other strategies on trees of up to 1 million nodes is less than 2s. The constraint solving strategy incurs about 2s of overhead for an empty tree, and its run-time grows exponentially from there. Even the worst of the other strategies can traverse a tree of a million nodes in the time it takes the constraint solving strategy to process an empty tree.

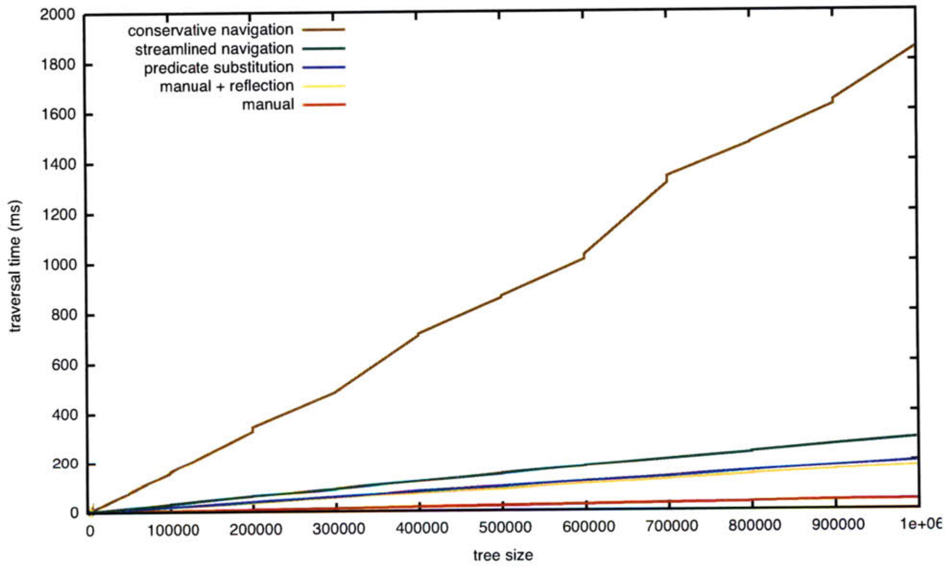
From the plots of Figure 3-24 we observe the following:

- All of the measured strategies scale to trees of one million nodes.
- The measured strategies are indistinguishable on small trees up to five hundred nodes. On such trees every strategy measures at zero milliseconds: *i.e.*, we cannot measure the performance differences. Hand-written code offers no measurable advantage over automatically synthesized code that uses reflective field access.
- On trees of over five hundred nodes the conservative navigation strategy is measurably more expensive than the others.
- On trees of over two thousand nodes the streamlined navigation strategy is measurably more expensive than the special predicate strategy and normal hand-written code. However, it appears that this extra time is consumed by the reflective field access and not the streamlined navigation strategy *per se*: the hand-written code using reflective field access becomes similarly more expensive with trees of this size.
- On trees of over four thousand nodes the predicate substitution strategy becomes measurably more expensive than the normal hand-written code. Again, it appears that this extra expense is due to the reflective field access more than the strategy choice.
- All examined strategies scale linearly. The conservative navigation strategy has a significantly steeper slope than do the rest of the examined strategies because of the cost of dynamically ensuring that duplicates are eliminated at each step of the computation.
- Predicate substitution performs very nearly as well as hand-written code using reflective field access on trees of up to one million nodes.

Figure 3-24 Traversal times for binary search tree abstraction function this.root.*(left+right)



(a) Tree sizes up to 5,000 nodes



(b) Tree sizes up to 1 million nodes

Based on these observations we make the following conclusions:

- It is worth the effort to apply a more specialized execution strategy, if possible. In particular, the streamlined navigation strategy shows very significant performance improvements over the conservative navigation strategy with medium to large trees. The predicate substitution strategy also shows some improvement over streamlined navigation, but this difference is not as large as the difference between conservative and streamlined navigation.

- Much of the cost of our implementations of these execution strategies is due to the engineering choice to use reflective field access. A future implementation of these ideas using bytecode generation for field access is likely to bring the performance of streamlined navigation and predicate substitution very close to normal hand-written traversal code.

- All of the measured execution strategies are likely to perform acceptably on small to medium-sized object structures in many contexts — even in their current realization with reflective field access.

- The streamlined navigation and predicate substitution strategies are likely to perform acceptably on even large object structures in many contexts.

This performance characterization provides evidence that synthesizing code for abstraction function specifications may be a reasonable alternative to hand-written implementations.

3.11 Annotating the JDK Collections v1.4

We evaluate the expressiveness of our approach on a subset of the standard Java libraries, the JDK Collections v1.4. This data-structure library was written primarily by Joshua Bloch and Doug Lea. Figure 3-25 characterizes the JDK Collections v1.4. We consider that there are five important abstract types: List, Set, Map, Map.Entry, and Iterator. The number of concrete subtypes of each of these abstract types may surprise the reader who has used the JDK Collections but not studied their implementation. A JDK user might think, for example, that there are only two concrete subclasses of List: ArrayList and LinkedList. However, there are in fact 14 concrete subclasses of List. Most of them are inner classes that are not directly visible to clients. They largely serve special utility purposes. For example, the Collections utility class has four inner classes that implement the List interface: EmptyList, SingletonList, SynchronizedList, UnmodifiableList.

Figure 3-25 Count of interesting classes in JDK Collections v1.4

Abstract Type	**Subtypes**			**Specification Fields**
	Abstract	**Concrete**	**Total**	
java.util.List	2	14	16	elts
java.util.Set	2	22	24	elts
java.util.Map	2	13	15	entries, keys, values
java.util.Map.Entry	0	9	9	key, value
java.util.Iterator	5	29	34	n/a
Total	11	87	98	

Figure 3-25 also names what we consider to be the specification fields for these abstract types. List has one specification field that we name 'elts' (short for 'elements'), which is a sequence of objects. Similarly, Set has one specification field, also named elts, but that is a set of objects rather than a sequence. Map has three specification fields: entries, keys, and values. The Map.Entry interface has two specification fields: key and value.

The abstraction functions for the concrete subclasses of Map.Entry are trivial, and always of the form **this**.specField = **this**.concreteField. These can all be easily expressed in JFSL.

The abstraction functions needed for concrete subclasses of List, Set, and Map are more interesting. Furthermore, they are all manifested as iterators in the code. So one way to assess the expressiveness of JFSL and our automatic synthesis is to see how many of these manually written iterators can be replaced by ones automatically synthesized from abstraction functions written in JFSL.

We were able to automatically synthesize 25 out of the 29 concrete iterators in the JDK Collections v1.4 from abstraction functions written in JFSL. All of these 25 abstraction functions can be evaluated with navigation or predicate substitution: constraint solving was not necessary. Some interesting examples of these abstraction functions are listed below. The remaining four cases that our approach did not cover are:

- Collections.UnmodifiableMap.UnmodifiableEntrySet.elts: The UnmodifiableMap class is a wrapper that provides an immutable view of some other Map object. Its UnmodifiableEntrySet inner class provides an immutable view of the Map.Entry objects wrapped by the UnmodifiableMap. We can easily express in JFSL that the value of elts specification field is the same as the value of the wrapped map's entries field. However, we cannot automatically synthesize an iterator that is equivalent to the manually written one because the manually written one wraps each returned Map.Entry object in an Immutable.Entry object.

- Collections.UnmodifiableCollection.iterator(): UnmodifiableCollection is a wrapper class that provides an immutable view of some other Collection object. Collection is a supertype of both Set and List. While Collection defines an iterator() method, it has no specification field: sets, lists, and bags are all expected to implement the Collection interface, and to provide an implementation of the iterator() method that iterates over their contents. Because Collection has no clear specification field it is impossible to write an abstraction function for it.

- IdentityHashMap.entries: IdentityHashMap is a hash-table implementation that uses linear-probing to resolve hashing conflicts. In such a strategy the table is one large array with keys stored at even indices and values stored at odd indices. There is no explicit instantiation of Map.Entry objects as there is in, for example, HashMap, which uses separate-chaining to resolve hashing conflicts. Consequently, we cannot write a JFSL abstraction function for the entries specification field. The value of the entries specification field is a set of Map.Entry objects. A JFSL abstraction function cannot cause objects to be created: it can only query objects that exist in the heap. Since the IdentityHashMap does not create Map.Entry objects as part of its internal storage, they aren't in the heap to be queried. The IdentityHashMap.EntryIterator instantiates Map.Entry objects when the client calls the next() method.

- TreeMap.SubMap.entries: The TreeMap class requires that its keys be drawn from a totally ordered domain, such as strings or integers. Consequently, it supports an operation to create a 'submap' comprising only entries whose keys are within some lower and upper bound in the key domain. Making the determination that a key falls within the selected range requires calling a Comparator object. Based on personal discussions with Dennis and Yessenov, it is not entirely clear how to specify these Comparator objects in a way that a static verifier can work with. Moreover, even if we had a good way to specify them, we would still need to be able to automatically synthesize their implementation in order to synthesize an iterator for the SubMap.entries field.

We verified that our 25 automatically synthesized iterators for these specification fields worked correctly by running 20,661 regression tests that were generated with the Randoop tool [93, 94]. We used Randoop to generate these tests against the original JDK Collections v1.4 code, and then ran the tests against our modified version of the code. The modified code past all of tests except those that expected mutating functionality from the iterators. The Iterator interface defines a number of methods that mutate the underlying data structure, such as add and remove. Our synthesized iterators only read the underlying data structure

and do not mutate it: they simply throw UnsupportedOperationException when these mutating methods are invoked. Randoop generated 1,757 tests (out of the 20,661) that expected to be able to mutate the underlying data structure via an iterator. Our modified code failed these tests, as expected.

3.11.1 Interesting Abstraction Functions from the JDK Collections

The most interesting abstraction functions in the JDK Collections v1.4 are those for TreeMap (Figure 3-26) and HashMap (Figure 3-27). The abstraction function for TreeMap.entries is like our ongoing example, since TreeMap implements a Red/Black binary search tree.

Figure 3-26 Abstraction functions for java.util.TreeMap

```
@SpecField({
        "public entries : set Map.Entry from this.root | " +
                "this.entries = this.root.*(left+right) - null",
        "public keys : set Object from this.root | " +
                "this.keys = this.entries.key",
        "public values : set Object from this.root | " +
                "this.values = this.entries.value" })
public class TreeMap extends AbstractMap
implements SortedMap, Cloneable, Serializable
```

The abstraction functions for HashMap are listed in Figure 3-27. The abstraction function for the entries specification field is the most interesting of the three. The sub-expression **this**.table.elems evaluates to the elements of the array (table) used for storage: it is a binary relation from **int** (array indices) to Map.Entry objects. Joining this sub-expression to the set of all 32-bit integers, denoted **int**, produces just the set of Map.Entry objects that are stored in the table array. These entry objects are not the all of the entries in the HashMap: they are just the entries that head the collision chains. Joining the set of these head entries to the reflexive transitive closure of the next field gives all of the entries in the HashMap.

Figure 3-27 Abstraction functions for java.util.HashMap

```
@SpecField({
        "public entries : set Map.Entry from this.table | " +
                "this.entries = (int.(this.table.elems) - null).*next - null",
        "public keys : set Object from this.table | " +
                "this.keys = this.entries.key - null",
        "public values : set Object from this.table | " +
                "this.values = this.entries.value" })
public class HashMap extends AbstractMap
implements Map, Cloneable, Serializable
```

Equality cannot be implemented automatically because two abstract objects can be equal even if their reps are not.

Liskov & Guttag [67, p.93]

Chapter 4

Equality

Until now programmers have been obliged to implement object equality comparison procedures manually. As is widely remarked in the literature (*e.g.*, [17, 49, 91, 93, 94, 97, 114]), this is both tedious and error-prone. Sources of error range from the normal minutiae of programming, to subtle issues around subclassing and delegation, to the practical challenge of cyclic references, and the conceptual challenge of differing concrete representations for the same abstract value.

By automating abstraction functions we can also automate object equality comparisons. The key idea is to define object equality in terms of the abstract state of the object. Two objects are equal if they have the same public specification fields and those fields have equal abstract values. We can perform this comparison at runtime thanks to techniques for automating abstraction function described above in Chapter 3.

Automation obviously removes many of the mechanical difficulties of implementing object equality comparisons. Techniques for computing equality in the presence of cyclic references are known in the research literature, but they are not easy to add to manually implemented equality comparisons, and so are usually not used by programmers. The key missing step has been automating the computation of the abstract state. With this piece we can now fully automate object equality comparisons.

Allowing objects of different types to be considered equal poses a variety of challenges. For example, a LinkedList and an ArrayList have completely different concrete representations. Another thorny case that has been the source of much discussion in the literature is when one type is a subtype of the other and adds some extra state: the classic example is ColourPoint extends Point. Computing the abstract state of the object is one important component of these comparisons; explicitly declaring which classes are comparable with each other is another.

We introduce the notion of *equality types* as a mechanism for the programmer to clearly express their intent about which types are to be considered potentially equal. Equality types ensure that the equals methods in the program, collectively, define a partition that aligns with the subtype structure of the program in a manner explicitly specified by the programmer.

To reduce the programmer annotation burden we have designed an equality type inference procedure. In our case study this procedure was able to infer the equality type for all but one class. It turns out that this one class had a design peculiarity that defeated the inference procedure, and that this design peculiarity has been independently classified as a bug in the vendor's bug-tracking system.

We evaluate our approach in terms of expressiveness and correctness on three real programs: the collections classes in the standard Java libraries (also studied above in Chapter 3), the Apache Commons Collections library, and JFreeChart (an open source plotting program). We find that almost all of the hand-written equals and hashCode methods in these programs can be replaced by our system. The methods that could not be replaced either intentionally violated the object contract or used weak references to control interaction with the garbage collector. Using both tests hand-written by the program authors and tests generated by the Randoop [93, 94] tool we show that our automatically generated implementations have fewer faults than the original hand-written versions.

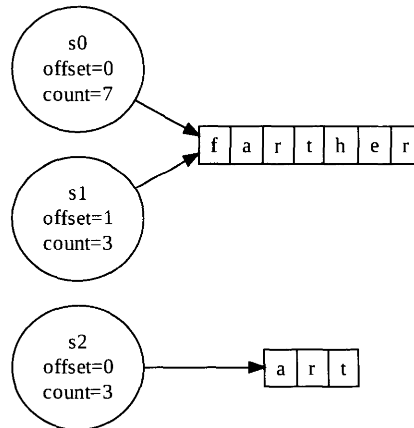
4.1 A Simple Example

An example of two objects with different concrete states representing the same abstract value is given in Figure 4-1, where the String objects s1 and s2 both represent the sequence of characters 'art'. Strings s0 and s2 are both created by calling a constructor of the String class that allocates new storage for the underlying character array. String s1, by contrast, is created by calling the substring method. Instead of allocating new storage, s1 shares the underlying array with s0 and just uses a different offset and length.

Figure 4-1 Example of two equal string objects

```
void main(String[] args) {
    String s0 = new String("farther");
    String s1 = s0.substring(1, 4);
    String s2 = new String("art");
    assert s1.equals(s2); // true
}

@SpecField("chars : seq char |
    subseq(this.value.elems,
        this.offset,
        this.count, this.chars)")
class java.lang.String {
    char[] value;
    int offset, count;
}
```

(a) Code snippet with two equal string objects. Abbreviated definition of String class included for reference.



(b) Heap diagram corresponding to code snippet in part **a**.

4.2 Object-Contract Compliance is Difficult

Previous researchers have documented that object-contract compliance is difficult to achieve for students [97], difficult for regular programmers [49, 114]; difficult for expert programmers [93, 94]; and difficult to write about for language designers and implementers [17, 91]. This section discusses five areas of difficulty, all of which are resolved by our systematic technique.

4.2.1 Difficulty: Simple Errors

Programming requires organizing many details. Programmers sometimes overlook some details, even if the they understand the issues at hand. In the programs we studied, simple errors (such as implementing equals but forgetting to implement hashCode) were common. Additionally, as programs evolve, maintenance of equality and hashing methods is frequently neglected. When a class is modified by a change to its fields, its equals and hashCode methods (as well as the equals and hashCode methods of its subclasses [80]) are likely to need modification as well.

If these methods are generated automatically, as they are in our technique, programmers are no longer burdened with the repetitive task of maintaining them, and their correctness is guaranteed.

4.2.2 Difficulty: Subclassing

The primary conceptual difficulty caused by subclassing, with respect to equality, is that all of the implementations of the equals method are supposed to, collectively, define a symmetric relation — but the programming language mechanism for implementing them, single dynamic dispatch, is asymmetrical. Looked at from another angle, the type hierarchy has the programmer thinking in terms of subsets but, an equivalence relation is a partition and there are no subsets in a partition. Bloch [17] illustrates this conundrum with the classic Point/ColourPoint example. Figure 4-2 lists a simple two-dimensional Point class and its equals method.

Figure 4-2 Simple Cartesian Point class and equals method (Bloch [17])

```
class Point {
    int x, y;
    Point(int x, int y) { this.x = x; this.y = y; }
    public boolean equals(Object o) {
        if (!(o instanceof Point)) return false;
        Point p = (Point)o;
        return p.x == x && p.y == y;
    }
}
```

As the classic example goes, suppose we want to extend this Point class and add a colour field to it in the new ColourPoint subclass. How shall we write the equals method of this subclass? There is no pleasant, commonly agreed upon solution to this problem. Bloch [17] gives two examples of equals methods that programmers in practice often write and shows that they are faulty. His first example uses the instanceof test the way it is popularly used and violates symmetry (Figure 4-3).

In Figure 4-4 Bloch [17] lists an example of how a clever programmer might try to modify the listing of Figure 4-3 to regain symmetry. However, this 'fix' comes at the cost of introducing a transitivity fault.

Roulo [102] suggests using the getClass() method instead of the instanceof test. While the suggestion is phrased in terms of the programming language features used, the essence of the idea is that points and colour points will never be considered equal to each other. Aligning the boundaries of the equality partition with the type structure of the program is a sensible and easily understood approach that we will discuss in more detail below. Following Roulo's suggestion in practice requires applying it uniformly, and that opens the door to the kinds of simple errors discussed above.

Finally, the superclass state may not be visible in the subclass. While this issue does not have the same intellectual depth as the one just discussed, it is a serious practical impediment for hand-written implementations. Moreover, it also points to the fact that implementations of equals are subject to the fragile base class problem [80] (changes in the superclass may cause problems in the subclass).

Figure 4-3 ColourPoint1 extends Point, violates symmetry (Bloch [17])

```
class ColourPoint1 extends Point {
    int colour;
    ColourPoint1(int x, int y, int colour) { super(x,y); this.colour = colour; }
    // Broken - violates symmetry.
    public boolean equals(Object o) {
        if (!(o instanceof ColourPoint1)) return false;
        ColourPoint1 cp = (ColourPoint1)o;
        return super.equals(o) && cp.colour == colour;
    }
    public static void violateSymmetry() {
        Point p = new Point(1, 2);
        ColourPoint1 cp = new ColourPoint1(1, 2, 3);
        assert p.equals(cp); // returns true
        assert cp.equals(p) : "symmetry violation";
    }
}
```

Figure 4-4 ColourPoint2 extends Point, violates transitivity (Bloch [17])

```
class ColourPoint2 extends Point {
    int colour;
    ColourPoint2(int x, int y, int colour) { super(x,y); this.colour = colour; }
    // Broken - violates transitivity.
    public boolean equals(Object o) {
        if (!(o instanceof Point)) return false;
        // If o is a normal Point, do a colour-blind comparison
        if (!(o instanceof ColourPoint2)) return o.equals(this);
        // o is a ColourPoint2; do a full comparison
        ColourPoint2 cp = (ColourPoint2)o;
        return super.equals(o) && cp.colour == colour;
    }
    public static void violateTransitivity() {
        Point p2 = new Point(1, 2);
        ColourPoint2 p1 = new ColourPoint2(1, 2, 3);
        ColourPoint2 p3 = new ColourPoint2(1, 2, 4);
        assert p1.equals(p2); // returns true
        assert p2.equals(p3); // returns true
        assert p1.equals(p3) : "transitivity violation";
    }
}
```

4.2.3 Difficulty: Delegation

Decorator/wrapper classes are used to add features to an existing class. This pattern is often used to make a thread-safe class from a thread-unsafe one, or to prevent mutation of the wrapped object. For example, a common implementation of equals in such a class is listed in Figure 4-5.

Figure 4-5 A simple wrapper class and its equals implementation

```
class Wrapper {
    Object x;
    Wrapper (Object o) { this.x = o; }
    public boolean equals(Object that) {
        synchronized (this) { return x.equals(that); }
    }
}
```

However, this common implementation has reflexivity, symmetry, and termination faults, as illustrated by Figures 4-6, 4-7, and 4-8. In Figure 4-6 the test makes use of a class named Contrarian that has an equals method with a fault: it always returns false. This fault in turn causes a reflexivity failure in our Wrapper class, which is itself faulty.

Figure 4-6 Test revealing reflexivity fault in common wrapper implementation of equals

```
static void reflexivityViolation() {
    Contrarian c = new Contrarian();
    Wrapper w = new Wrapper(c);
    assert w.equals(w) : "reflexivity violation";
}

class Contrarian { public boolean equals(Object o) {return false;}}
```

Figure 4-7 lists a test that reveals a symmetry fault in our wrapper: we see that w.equals(o) is true but that the converse o.equals(w) is false. Note that the fact that w.equals(o) evaluates to true is probably inconsistent with the programmer's intentions, but that is not *per se* an object-contract violation. The object-contract violation here is the fact that w.equals(o) and o.equals(w) do not return the same result. Pacheco et al. [94] found exactly this problem in the JDK Collections v1.4.

Figure 4-7 Test revealing symmetry fault in common wrapper implementation of equals

```
static void symmetryViolation() {
    Object o = new Object();
    Wrapper w = new Wrapper(o);
    assert w.equals(o);  // passes
    assert o.equals(w) : "symmetry violation";
}
```

Figure 4-8 lists a test that causes our common wrapper implementation of equals to throw a StackOverflowError: *i.e.*, to enter an infinite loop. This undesirable behaviour is provoked by having a wrapper object wrap itself.

Figure 4-8 Test revealing non-termination in common wrapper implementation of equals

```
static void stackOverflow() {
    Wrapper w = new Wrapper(null);
    w.x = w;
    assert w.equals(w) : "infinite loop";
}
```

The simple delegation code of Figure 4-5 contains three serious object-contract faults: reflexivity, symmetry, and termination. Yet this idiom is widely used, including in the standard Java libraries, as we discuss below. The essence of these problems with delegation is that equality of the wrapper is being defined in terms of the code of the wrapped object. In our approach, the abstract state of the wrapper object is defined in terms of the abstract state of the wrapped object, and equality is defined in terms of these abstract states.

4.2.4 Difficulty: Objects of Different Classes

Sometimes objects should be regarded as equal even when they belong to different classes. The JDK Collections contain many such cases: a LinkedList, for example, can be compared to an ArrayList, and the two will be considered equal if they contain the same set of elements, even though they use different representations internally.

4.2.5 Difficulty: Cyclic Object Graphs

Cyclic object graphs – which are not uncommon in object-oriented programs – are even more challenging for equality and hashing than they are for serialization. Programmers often try to sidestep the problem by defining equality only on fields that do not contain cycles, or by adding a precondition to equals and hashCode that the object's references should be acylic. In such circumstances it is usually the case that if this precondition is violated that the equals and hashCode methods will fail to terminate. For example, the Javadoc comment for java.util.List says the following:

> While it is permissible for lists to contain themselves as elements, extreme caution is advised: the equals and hashCode methods are no longer well defined on such a list.

Moreover, adding predconditions to overriding method implementations in subclasses is generally considered poor programming practice because it violates Liskov's substitutability principle [67].

Proper handling of equals and hashCode in the presence of cycles is, however, not impossible, as we discuss below.

4.3 Object Equality in Terms of Abstract State

In general outline, our approach is the following. According to the object contract, the equals method is supposed to implement a mathematical equivalence relation. It is well known that any mathematical equivalence relation, $\approx$, has a corresponding function f ($\approx$ is sometimes known as the 'equivalence kernel' of f), such that:

$$a \approx b \;\Leftrightarrow\; f(a) = f(b)$$

We define the equals equivalence relation in terms of the object abstraction function $\mathcal{A}$: in other words, our system considers two objects to be equal if they have the same abstract state. This approach is consistent with our observations of code written by expert programmers.

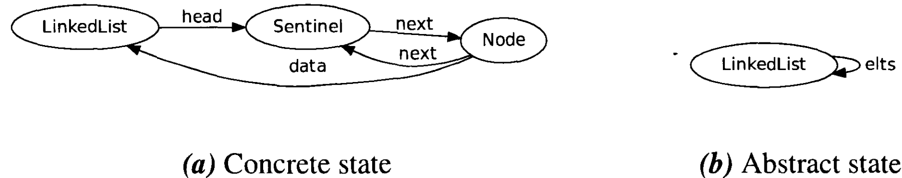
We define the object abstraction function $\mathcal{A}$ to be the combination of the field abstraction functions α_i for the public specification fields of each object. Two objects are equal if they

have the same public specification fields and if those fields have equal abstract values. This definition of equality is recursive: the abstract values of the objects referred to by each specification field must be computed, and so on. The main challenge in this approach is handling cycles in the abstract state, which we discuss next.

4.3.1 Cyclic References

There may be cycles in both the concrete state and in the abstract state. Consider the example of a linked list that contains itself, as illustrated in Figure 4-9. This linked list implementation is designed with a sentinel node, as is done in the JDK Collections that we study, and so the next relation is cyclic. This cycle in the concrete next relation is encountered during the evaluation of the abstraction function for the elts specification field (α_{elts}). Evaluating field abstraction functions in the presence of cycles in the concrete state was discussed in the previous chapters.

Figure 4-9 Concrete and abstract cyclic references



(a) Concrete state *(b)* Abstract state

However, as can be seen in Figure 4-9b, there is also a cycle in the abstract state, since the linked list contains itself as an element. Cycles in the abstract state may cause difficulty for operations such as equality and hashing that operate on the abstract state.

A common definition of equality in the presence of cycles is that two objects are equal if the (possibly infinite) tree unfoldings of their state are element-wise equal. This definition of equality is used (in terms of concrete state) in Scheme [107, §11.5] and Eiffel [78] and is seen elsewhere in the research literature [2, 38].

There is also a common technique used for computing equality according to this definition: build a table of assumptions and search for evidence to refute them; the algorithm terminates when either contrary evidence is found (the objects are not equal), or at some step no new evidence is found and no new assumptions are made (the objects are equal). Consider two linked lists, L1 and L2, that contain each other so the abstract state looks like so: L1 $\rightleftarrows$ L2. These two lists are equal to each other according to the definition. The computation of L1.equals(L2) proceeds as follows:

- add ⟨L1, L2⟩ and ⟨L2, L1⟩ to the assumption table;
- compare the contents of L1.elts (=L2) and L2.elts (=L1);
  - initiate comparison of L2 and L1;

    - discover ⟨L2, L1⟩ in assumption table;
    - return true;

- no more elements to examine and no contrary evidence found, so assumptions must be correct, therefore return true.

Chapter 5 explores a variety of approaches to handling cycles during hashing. This assumption-table technique does not work for hashing because hashing is a unary operation (*i.e.*, on a single object), whereas equality is a binary operation (*i.e.*, on a pair of objects).

4.3.2 A Syntactic Shorthand: @ConcreteEquality

We have observed that in many cases the programmer wishes to have trivial abstraction functions of the form this.abstractField = this.concreteField, and to have such a trivial abstraction function for every concrete field. We introduce the @ConcreteEquality as a syntactic shorthand for this common idiom.

4.3.3 Equality Types

We introduce the idea of *equality types* as a means to assist the programmer in ensuring that the equals method defines a partition while allowing objects of distinct types to be considered equal. The equality type is realized as a specification field that is common to all objects, as listed in Figure 4-10. Figure 4-10 also defines the default abstraction function for this specification field using the special syntax @ThisClass, which means that the equality type is the object's concrete class.

Figure 4-10 Declaration of equalityType specification field

```
@SpecField("equalityType : one Class | this.equalityType = @ThisClass")
public class Object {}
```

Subtypes may redefine the abstraction function for the equalityType specification field. For example, TreeSet might define it to be this.equalityType = Set@class, indicating that Set is the equality type for TreeSet. This kind of redefinition is only necessary in cases where different concrete classes are intended to implement the same abstract datatype.

For the purposes of equality comparison only the public specification fields of the equality type are used.

Recall the classic Point/ColourPoint example, given above in Figures 4-2, 4-3, and 4-4. The idea of equality types forces the programmer to make a clear design decision: the equality type for ColourPoint is either Point or ColourPoint. In the former case the colour field is not used for equality comparisons. In the latter case a Point and a ColourPoint will never be considered equal. In either case the object contract will be respected.

4.3.4 Inferring Equality Types

To reduce the programmer's annotation burden we attempt to infer the equality type abstraction function for concrete classes that do not have the @ConcreteEquality annotation. In the case of @ConcreteEquality, the default equality type abstraction function, described above, that names the concrete class of the object is usually what the programmer intends.

Consider a concrete class C that does not have (nor inherit) the @ConcreteEquality annotation. The basic idea of our inference is to find the first supertype of C, other than Object but including C itself, that declares new specification fields. 'First' here is defined in terms of proximity to Object. In a language where the subtype hierarchy is a strict tree and so each type has only one supertype this inference is simple. In a language where any given type may have multiple supertypes, such as Java, it requires a bit more sophistication.

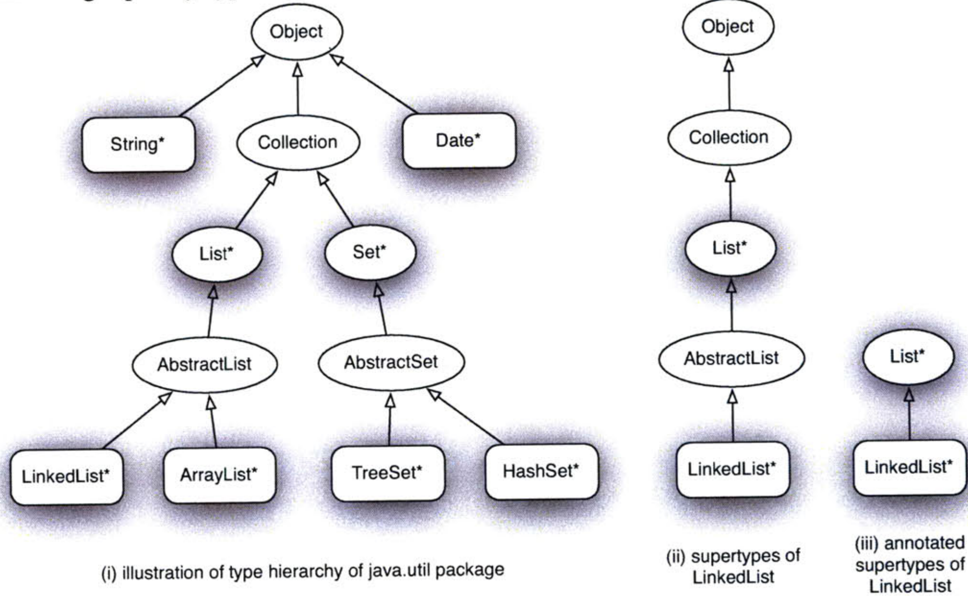
Consider, for example, C as java. util .LinkedList from the JDK Collections library, as depicted in Figure 4-11a. We are going to infer that the equality type of LinkedList is List. Our inference proceeds as follows. Consider the type structure of a Java program as a partially ordered set with a distinguished greatest element (java.lang.Object). Name this poset S_p; in our list example, S_P is illustrated in Figure 4-11a-i.

Now consider $S_C \subseteq S_p$ that consists of only C and all of C's supertypes. In our list example, S_C is illustrated in Figure 4-11a-ii. This subset S_C forms a lattice, with C as the least element and Object as the greatest element. Consider a further subset, $S_A \subseteq S_C$, that contains only C and its supertypes annotated with @SpecField, excluding Object. In our list example, S_A is illustrated in Figure 4-11a-iii. If S_A is a lattice, then the equality type of C is inferred to be the greatest element of S_A.

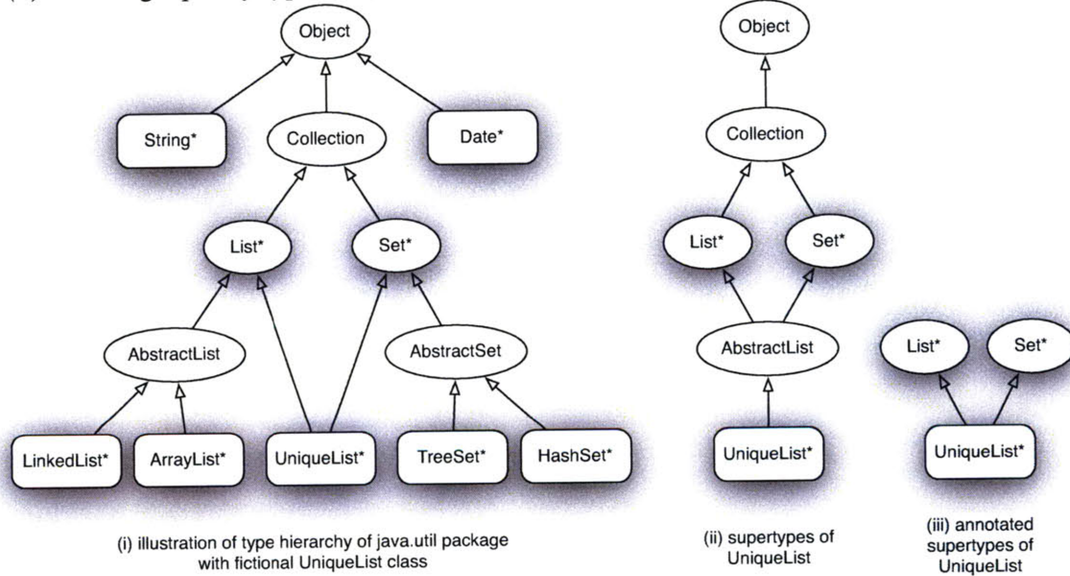
If S_A has no greatest element (*i.e.*, is not a lattice), then our inference returns no result and the program requires manual annotations. Such programs sometimes contain design flaws. For example, consider the fictional UniqueList class shown in Figure 4-11b that implements both the Set and List interfaces. Our inference procedure does not return a result for this fictional class. It turns out that this fictional class violates the specification of java.util.Collection.equals(), which states that instances of List may not be equal to instances of Set, and so no class may correctly implement both List and Set.

Figure 4-11 Illustrations of equality type inference

(a) Inferring equality type for LinkedList



(b) Inferring equality type for (fictional) UniqueList



Legend: Ellipses represent abstract types. Boxes represent concrete types. Asterisks indicates a type other than Object that is annotated with @SpecField.

Figure 4-12 Number of manually written equals and hashCode implementations in original program, and remaining after code was converted to use our technique.

Case Study		Before	After
JDK	equals	24	2
	hashCode	24	3
Apache	equals	56	6
	hashCode	57	9
JFreeChart	equals	263	0
	hashCode	114	0

4.4 Empirical Evaluation

We evaluate our approach to object equality from two software engineering perspectives:

1. Is it sufficiently expressive to replace the hand-written equals and hash code method implementations that programmers write in practice?
2. What effect does replacing hand-written implementations with automatically generated implementations have on the correctness of actual programs?

To answer these questions, we annotated three widely used programs: the JDK Collections library v1.4 (a subset of java.util), the Apache Commons Collections library v3.2, and JFreeChart 1.0.0 (an open source plotting program that is a member of the DaCapo Benchmark Suite [15]).

4.4.1 Expressiveness

Our technique was able to replace almost all the equals and hashCode methods of the three benchmark programs, as shown by Table 4-12. Those that we were not able to replace were within classes that either intentionally violated the object contract (such as IdentityHashMap) or used weak references to control interaction with the garbage collector.

Our equality type inference algorithm produced acceptable output in all but one case, where it did not produce any output at all. The class IdentityHashMap.Entry implements both the Map.Entry and Iterator interfaces, which have no common supertype that declares their specification fields, and so caused the inference to fail. While the JDK 1.4 code shipped in this configuration, this questionable design decision has resulted in bug #6312706 in the JDK bug database [53].

Figure 4-13 Test results

		Total	*Original* Errors	*Original* Failures	*Annotated* Errors	*Annotated* Failures
Machine-generated Tests by Randoop	JDK Collections	20661	1	184	0	0
	Apache Collections	3415	0	7	0	0
	JFreeChart	59	2	57	2	0
Hand-written Tests by Vendor	JDK Collections	1366	2	5	2	7
	Apache Collections	2443	0	0	0	0
	JFreeChart	1038	1	8	0	3

'Errors' are test cases that throw unexpected exceptions.
'Failures' are test cases that compute incorrect results.

4.4.2 Correctness

To evaluate correctness, we ran two sets of unit tests on each of the benchmark programs: a set generated by the Randoop tool [94] and the standard set provided by the program's author.

The annotated versions of these programs are more often correct than the originals. This is primarily shown in Table 4-13, where it can be seen that the annotated versions usually pass more test cases than the original versions. Where they do not, the test cases themselves either contain hard-coded constants tied to particular implementations or are faulty, as discussed below.

Table 4-14 classifies the faults we found according to the parts of the object contract violated (§1.4) and according to the reasons that writing correct implementations of equals and hashCode are difficult (§4.2). The application of our technique fixed all faults found.

JDK Collections Faults. We found the same faults in the JDK 1.4 collections that Pacheco et al. [94] found previously: namely that some of the wrapper sets and maps had reflexivity and symmetry faults.

There is an interesting case where our technique forced us to restructure some of the JDK code, from an original design that produced some surprising behavior to a more conventional design. Originally, the IdentityHashMap.Entry class implemented both the Iterator and Map.Entry interfaces. When one called next() on the iterator, it returned itself in the guise of a Map.Entry. A programmer retaining a reference to this entry from a previous iteration of the loop would be surprised to find that its key and value were now the key and value for the current iteration. This design saved allocating a Map.Entry object for each iteration of the loop, at the cost of surprising mutation behavior (which is documented as Bug #6312706 in the Sun/Java bug database).

Our technique forced us to split this class into two classes, because Iterator and Map.Entry

are different equality types, and so no single class may implement both of them.

Failing JDK Collections test cases. Our annotated JDK Collections code fails two vendor test cases that the original code passed: BitSet.hashCodeTests and Vector.ToStringTests. Both of these test cases have hard-coded values (such as hash codes and string encodings) that are computed differently (but still correctly) in our implementation.

Apache Collections Faults. Delegation issues led to symmetry faults in SynchronizedBuffer.equals() and ReferenceMap.equals(), and a reflexivity fault in MapBackedSet.equals().

PriorityBuffer.equals() was not implemented, even though it was clear from the usage that an implementation was expected.

JFreeChart Faults. We found 32 classes in JFreeChart that implement equals but not hashCode, which is a simple but common error.

AbstractObjectList.equals() was actually performing a prefix test, rather than an equality test, because it neglected to compare the size of the lists. This caused symmetry and transitivity failures. Furthermore, the equals and hashCode implementations in this class failed to terminate on cyclic inputs.

ShapeUtilities.equals(GeneralPath,GeneralPath) had a copy-paste error that resulted in it comparing the first argument to itself, and so it always returned true.

XYImageAnnotation had a 'todo' indicating that the image field should be serialized. After de-serialization, the image field was null, and so the equality test failed.

Range had a fault caused by the special primitive double value NaN (not-a-number). The programmer mistakenly expected == to be reflexive, when in fact it is not: all boolean operations involving NaN return false, including NaN==NaN. java.awt.geom.Point2D, which is used by JFreeChart, had a similar fault.

We found 14 classes in JFreeChart that neglected parts of their abstract state in their equals and hashCode implementations in order to avoid potential cycles.

Faulty JFreeChart test cases. We found faults in five of JFreeChart's vendor-supplied unit tests. Interestingly, these tests did not exhibit failures when run against the original JFreeChart code because of the fault in AbstractObjectList.equals(): two wrongs made a right. Once we fixed the fault in AbstractObjectList, we found that these test cases failed. Upon inspection, we discovered that the test cases had incorrect expectations about which objects are equals. The numbers in Table 4-13 reflect the corrected versions of these tests cases. The five test cases in question were:

- XYAreaRendererTests.testSerialization()
- StackedXYAreaRenderer.testSerialization()
- CombinedRangeCategoryPlotTests.testCloning()

- CombinedDomainCategoryPlotTests.testCloning()
- CategoryPlotTests.testCloning().

Failing JFreeChart test cases. As shown in Table 4-13, our annotated version of JFreeChart still fails three tests – which also failed when run with the original code.

XYImageAnnotationTests.testSerialization() fails because of the previously mentioned serialization/equality fault in XYImageAnnotation.

XYPlotTests.testGetDatasetCount() has expectations that are inconsistent with the behavior of the XYPlot constructor.

SegmentedTimelineTests.testMondayThroughFridayTranslations() imprecisely transforms one representation of time to another.

Observations. On the basis of this, albeit limited, sample of programs we might infer that:

1. When programmers wish to simply compare the concrete state of two objects, the kinds of mistakes they make amount to simple oversights, rather than the more subtle issues discussed in §4.2.
2. When an abstract view is needed in order to compare objects of different concrete classes, programmers tend to make mistakes due to the more subtle difficulties, most often delegation.
3. Cyclic object graphs are not handled well. The two most common approaches that we observed were: (1) allowing the stack to overflow, and (2) neglecting parts of the abstract state that may lead to a cycle.

Figure 4-14 Classification of faults found (and fixed)

	Object Contract Violations												Difficulties (§4.2)					
	equals									hash								
Fault Location / Description	*reflexivity*	*symmetry*	*transitivity*	*non-null*	*consistency*	*termination*	*well-foundedness*	*side-effect free*	*faithfulness*	*equals consistent*	*termination*	*side-effect free*	*oversight*	*evolution*	*subclassing*	*delegation*	*different classes*	*cycles*
JDK 1.4 Collections																		
SynchronizedSet.equals(Obj)	1	1														1	1	
UnmodifiableSet.equals(Obj)	1	1														1	1	
SynchronizedMap.equals(Obj)	1	1														1	1	
UnmodifiableMap.equals(Obj)	1	1														1	1	
Apache Commons Collections																		
MapBackedSet.equals(Obj)	1	1														1	1	
SynchronizedBuffer.equals(Obj)	1	1														1	1	
ReferenceMap.equals(Obj)	1	1				1	1				1						1	1
PriorityBuffer.equals(Obj)									1				1				1	
JFreeChart 1.0.0																		
equals but not hashCode[1]										32			32					
Ignoring abstract state[2]									14									14
AbstractObjectList.equals(Obj)		1	1			1	1		1		1		1					
ShapeUtilities.equals(GP,GP)									1				1					
XYImageAnnotation					1								1					
Range.equals(Obj)	1												1					
Totals	8	8	1		1	2	2		17	32	2		37			6	8	15

1. The following JFreeChart classes implement equals but not hashCode: BlockBorder, ChartRenderingInfo, Colorblock, CombinedDomainXYPlot, ContourPlot, DefaultBoxAndWhiskerCategoryDataset, DefaultDrawingSupplier, EntityCollection, FastScatterPlot, HistogramDataset, ItemLabelPosition, JFreeChart, KeyToGroupMap, LegendGraphic, LegendItemCollection, MeterPlot, MiddlePinNeedle, PiePlot3D, PieSectionLabelGenerator, PinNeedle, PlotRenderingInfo, SimpleHistogramBin, SpiderWebPlot, StackedBarRenderer, StandardXYToolTipGenerator, StandardXYZURLGenerator, SymbolicXYItemLabelGenerator, Task, TaskSeriesCollection, ThermometerPlot, TimeTableXYDataset, XYPlot.

2. The equals implementations in the following JFreeChart ignore some part of their abstract state in order to avoid potential cycles: CategoryPlot, CategoryTableXYDataset, ChartRenderingInfo, CombinedDomainCategoryPlot, CombinedDomainXYPlot, CombinedRangeCategoryPlot, CombinedRangeXYPlot, DefaultTableXYDataset, IntervalXYDelegate, PlotRenderingInfo, TableXYDataset, XYPlotTests, XYSeriesCollection.

4.5 Related Work

Automatic generation of concrete equality comparisons is not uncommon: there are tools (*e.g.*, Eclipse), libraries (*e.g.*, Apache Commons Lang), annotations (*e.g.*, [79]), built-in language features (*e.g.*, [78, 91, 105, 107]), and language extensions (*e.g.*, [10]) that accomplish this goal. Most of these approaches are for functional languages or work only for record-like structures within object-oriented languages.

Static code generation techniques can be fragile if the programmer needs to remember to regenerate the code each time a new field is added to a class. Approaches based on annotations, reflection, built-in language features, and language extensions are robust in this respect.

Many of these approaches are also fragile in the face of cyclic object references; notable exceptions include Eiffel [78] and Scheme [107]. We borrow the idea of treating the tree-unfolding of cyclic structures from Scheme and the strategy of using an assumption set to compute equality of cyclic structures from Eiffel.

Grogono and Sakkinen [38] and Vaziri et al. [114] both have a concept of abstract state. However, neither of these approaches allow objects of different concrete classes to be considered equals. The two crucial elements of our technique that facilitate these kinds of comparisons are explicit equality types and rich abstraction functions.

Grogono and Sakkinen [38] also distinguish fields as 'essential' or 'accidental' and use this distinction to specify object ownership properties that are taken into account when cloning objects.

Vaziri et al. [114] distinguish fields as either part of the object's identity or not. The identity of an object must be immutable, whereas the other fields may refer to mutable objects. Object construction is controlled so that there can never be two different objects with the same type and the same identity. In other words, Vaziri et al. [114] adopt the Liskov and Guttag [67] view of the consistency of equality, but provide some linguistic mechanisms to make this discipline easier for practicing programmers to follow.

Object mutability (or immutability) plays an important role in the ideas of Liskov, Guttag, Grogono, Sakkinen, and Vaziri et al. We provide a mechanism, and we think that programmers would be well-advised to use that mechanism in a manner consistent with the ideas of these researchers.

The object contract and difficulties in implementing it have been discussed extensively elsewhere. Abiteboul and Bussche [2], for example, present three formalizations of deep equality and prove that they are equivalent. Their main concern is deep equality in the presence of cyclic object graphs. Bloch's book [17], which is a standard reference for Java programmers, explains other common pitfalls (such as asymmetry due to subclassing) and how to avoid them. Odersky et al. [91] also discuss common pitfalls, and provide some recipes for implementing equals and hashCode manually.

Finally, we note that Grogono and Sakkinen [38] consider four different notions of equality, including a topological notion of equality that is not discussed elsewhere. They use the term

'structural equality' for this topological notion, and use the term 'deep equality' for what we refer to as 'concrete equality' and some other authors refer to as 'structural equality.'

4.6 Object Equality and Observational Equivalence

Liskov, Guttag, Wing, and Leavens [60, 67–70] recommend that object equality should imply *observational equivalence: i.e.*, that the two equal objects are indiscernible, and any computation would produce the same result with either one. The phrase 'observational equivalence' is also used in the formal semantics of programming languages with a similar meaning: any observationally equivalent terms compute the same result.

One could see this idea of observational equivalence as a modern incarnation of *Leibniz's Law* [64], which has two parts: the *Identity of Indiscernibles* and the *Indiscernibility of Identicals*. In the notation of modern symbolic logic one might write for the first part [36]:

$$\forall P(Px \leftrightarrow Py) \rightarrow x = y$$

meaning that if object y has every property P that object x has, and vice versa, then x is identical to y. The converse, the *Indiscernibility of Identicals*, might be written as [36]:

$$x = y \rightarrow \forall P(Px \leftrightarrow Py)$$

In the Haskell programming language object equality implies observational equivalence. The Haskell runtime system uses this property to reduce the memory footprint of the running program by sharing objects. However, Haskell has neither mutation nor subtyping.

In object-oriented programming languages with subtyping, aliasing, and mutable state it is much harder to ensure that object equality implies observational equivalence. Some of the challenges include:

- Mutation violates the Indiscernibility of Identicals: even if x and y have the same state now now and then x is mutated it may no longer be equal to y.
- In a program with subtyping there may be multiple implementations of some method m. In order to ensure that the various subtypes are *behavioural subtypes* [6, 60, 69, 70] we need to verify that the various implementations of method m are equivalent, which is undecidable in general.
- Even if the various implementations of method m have been proven to satisfy some specification s, s may be under-constrained and hence the client may actually be depending on a specification stronger than s. One might argue that this is the client's fault, but it nevertheless leads to a situation where the client may work with m_1 and not work with m_2.

In our experience we have observed that object-oriented programmers sometimes seem to expect object equality to imply observational equivalence and sometimes do not. When they do not expect object equality to imply observational equivalence, sometimes they seem to expect to violate the Identity of Indiscernibles, sometimes they seem to expect to violate the Indiscernibility of Identicals, and sometimes they seem to expect to violate both.

Our synthesized equals methods provide a stepping stone towards observational equivalence but do not guarantee it. If the programmer wants this stronger property then they will have to use other analysis tools, such as type-checkers (*e.g.*, [95, 125] or verification tools (*e.g.*, [32, 119], [59, 123], [21, 63]). Whatever the case, ensuring object-contract compliance is a first step, and having declarative abstraction functions is another step towards verification.

Chapter 5

Hashing

Object hashing needs to be consistent with object equality so that two equal objects hash to the same value. Since we can define object equality in terms of the abstract state of the object, we also define object hashing in terms of the abstract state.

In algorithms text books, discussions of hashing assume that a hash code is computed from a value represented as a string of bits – and indeed this is sufficiently general to address computational properties of hashing. But from a programming perspective, an object is not a string of bits, but is rather a graph structure on the heap. Hashing this structure therefore requires a traversal. We therefore distinguish the hashing *method* from the hash *function* and the traversal *strategy*. The method takes an object and returns an integer by composing the function (that computes an integer from a sequence of integers) and the strategy (that computes a sequence of integers from an object). Since hashing is an operation on the abstract state of the object, the traversal strategy makes use of the object abstraction function $\mathcal{A}$.

We describe six object hashing strategies. These strategies are situated at different points in a two-dimensional design space.

The first strategy design dimension is how much of the object state is examined to produce the hash: examining more of the state is likely to produce fewer hash collisions but takes more time. The textbook treatment of hashing strings assumes that many of them will be stored in a single hash table: for example, the number of strings in the hash table is usually assumed to be much larger than the average length of the strings stored in it. Under these circumstances it is worth the effort to examine the object in depth. However, the practical circumstances of many object-oriented programs differ from these textbook circumstances. In practice, many object-oriented programs often have small hash tables of large objects. For example, the average size of the objects in a hash structure could be larger than the number of objects in the structure. Under these practical circumstances it can make sense to examine less of the object state and tolerate the higher potential for hash collisions.

The other strategy design dimension is the kind of cycle detection the strategy uses. Cyclic references may confound both correctness and performance. We identify *distinct logical cycles* as a kind of cyclic reference that does not exactly align with a physical cycle. In order

to be consistent with equality a hashing strategy must handle the possibility of distinct logical cycles in some way. Explicitly checking for these kinds of cycles can be more expensive than checking for physical cycles. Moreover, the technique used for computing equality in the presence of cyclic references cannot work for hashing, since that technique takes advantage of the fact that equality is a binary operation, whereas hashing is a unary operation.

We measure the performance of these strategies on real programs, primarily from the DaCapo [15] benchmark suite, and on synthetic micro-benchmarks that we generated using the Korat [20, 82] tool. The empirical results are:

- Hashing strategy selection can affect overall program performance.
- Automatically generated hashing methods can have comparable performance with manually written ones.
- Returning a constant, while terrible in theory, is not always terrible in practice. In some real programs it even offers moderately better performance than hand-written hashing methods.
- Finite unrolling is a good general-purpose hashing strategy. It may not always be the best, but (at least in our experiments) it is never terrible.

5.1 Design Objectives

An object hashing strategy should be general, correct, and efficient. In this section we discuss these design objectives with respect to important corner cases that confound them.

5.1.1 Design Objective: Generality

The main challenge to generality of a hashing strategy is different concrete representations of the same abstract value. In most cases we resolve this challenge by applying the hashing strategy to the abstract state of the object, computed with the object abstraction function $\mathcal{A}$. An alternative approach that we also examine is to simply ignore the state of the object and always return a constant.

5.1.2 Design Objective: Correctness

An object hashing strategy is correct if it is consistent with equals and it terminates on any input. These are conditions from the object contract that we discussed above in §1.4. To ensure consistency with equals our traversal strategies (with the exception of returning a constant) use the object abstraction function $\mathcal{A}$. Like the computation of equality, the strategy will apply $\mathcal{A}$ to each object that it examines. Consider, for example, a java.util.HashSet

object s that contains two Cartesian Point objects p_1 and p_2. Using the field abstraction function α_{elts} (part of $\mathcal{A}_{\text{HashSet}}$) we compute that s.elts = $\{p_1, p_2\}$. Then using α_{x} and α_{y} from $\mathcal{A}_{\text{Point}}$ on objects p_1 and p_2 we compute the their x and y values. As discussed above in Figure 4-9, there may be cycles in the abstract state. The strategy is charged with ensuring termination in the presence of such cycles.

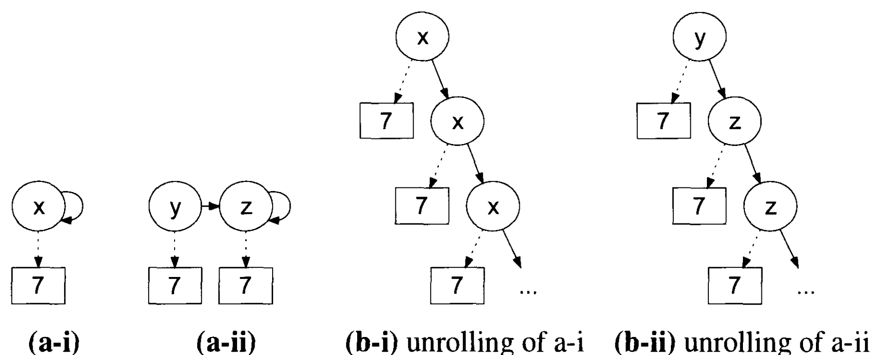
The main threat to termination of a hashing method is termination of its hashing strategy: the termination of our abstraction function execution strategies was discussed above, and classic hash functions are designed to terminate. Cyclic object structures are the primary reason that a hashing strategy may fail to terminate (we do not consider infinite object structures).

We distinguish between *physical* cycles and *logical* cycles (Adams and Dybvig [3] also have these two notions, but do not name them.) To guarantee termination it is sufficient for a hashing strategy to detect physical cycles. However, to maintain consistency with equality a hashing strategy must also be designed with the notion of logical cycles in mind. We now explain these two kinds of cycles and the challenges they create for hashing strategy correctness.

We say that object x is in a physical cycle if there is some path through the heap that begins at x and reaches x. Consider, for example, Figure 5-1a-i, which depicts an object x of type Node. The Node class defines two fields: next and data. In Figure 5-1 solid lines represent next references and dotted lines represent data references. So, according to Figure 5-1a-i, x.next = x and x.data = 7. The reference from x to x via the next field creates a physical cycle.

We say that an object structure contains a *logical cycle* if it contains two objects y and z (possibly the same) such that there is a path from y to z and y.equals(z). Every physical cycle is also a logical cycle, by the reflexivity of equality. Assuming finite object structures, every logical cycle also contains a physical cycle. The case we are concerned with is when a logical cycle is larger than the physical cycle it contains: we refer to such cycles as *distinct logical cycles*. In these cases techniques used to detect physical cycles may not be adequate to ensure correctness of the hashing strategy. Figure 5-1b-i shows an object structure with a distinct logical cycle.

Figure 5-1 Equivalent cyclic object structures and their unrollings



(a-i) **(a-ii)** **(b-i)** unrolling of a-i **(b-ii)** unrolling of a-ii

As discussed above in Chapter 4 we use the common tree-unfolding definition of equality for cyclic structures: two objects are equal if their (possibly infinite) tree unfoldings are equal [2, 3, 38, 78, 107]. According to this tree unfolding notion of equality, all of the objects in Figure 5-1 — x, y, and z — are equal to each other. Figure 5-1b shows the infinite tree unfoldings of objects x, y, and z originally depicted in Figure 5-1a; this visual representation makes it easier to see that they are all equal.

Distinct logical cycles do not present any special difficulty in computing equality. The technique we described above in Chapter 4 for computing the equality of cyclic structures still computes correct results for object structures that contain distinct logical cycles. Let's walk through the computation of x.equals(y) from Figure 5-1:

- add $\langle x,y \rangle$ and $\langle y,x \rangle$ to the assumption table;
- compare x.data and y.data;
  - return true, since 7=7;
- compare x.next and y.next;
  - add $\langle x,z \rangle$ and $\langle z,x \rangle$ to the assumption table;
  - compare x.data and z.data;
    - return true, since 7=7;
  - compare x.next and z.next;
    - return true, since $\langle x,z \rangle$ is already in the assumption table;
  - return true, since no more evidence to examine;
- return true, since no more evidence to examine.

Hashing methods, on the other hand, are unary operators, and so this pairwise comparison technique cannot be used: cycles must be either worked around or explicitly detected. We detect physical cycles by maintaining a set of previously visited memory addresses. We detect distinct logical cycles by maintaining a set of previously visited objects. This set may itself make use of the hashCode and equals methods, and so this can be more expensive than physical cycle detection.

5.1.3 Design Objective: Performance

Conventional wisdom holds that hashing methods should produce few hash collisions and run in time linear in the size of their input [58]. Consequently almost all classic hashing functions have at worst this linear performance. However, for a hashing method to have this linear performance it must use a hashing strategy that is linear.

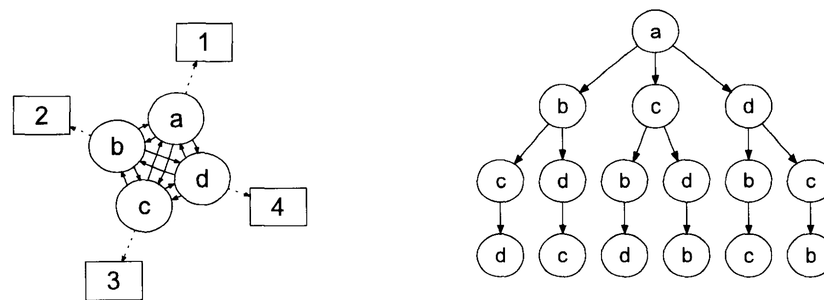
We are not aware of any strategy that examines the entire object state (to reduce the number of potential collisions), runs in linear time, is consistent with equality, and terminates on all inputs. Since we do not consider inconsistent nor non-terminating strategies to be acceptable, a trade-off must be made between the time taken to compute the hash code (*i.e.*, amount of the object's state examined) and the number of potential collisions.

Returning a constant value runs in constant time while maintaining consistency with equality, but does not examine the state of the object and hence produces many collisions.

A naïve hashing strategy would be linear for tree-like object structures and produce few collisions. However, such a strategy is likely to be super-linear for structures with undirected cycles (because subobjects will be visited for each incoming reference) and to fail to terminate on structures with directed cycles.

A strategy that examines the entire state of the object (to reduce potential hash collisions) and detects logical cycles (to maintain correctness) will run in super-linear time. At best, logical cycle detection will cause it to run in quadratic time. At worst it may potentially be exponential. Consider, for example, the fully connected structure shown in Figure 5-2a. The tree unrolling of this structure, starting from node *a*, and pruning each path containing a cycle, is shown in Figure 5-2b. The size of this tree unrolling is exponential in the number of objects in the original structure.

Figure 5-2 A fully connected object structure and its unrolling



(a) fully connected object structure ***(b)*** tree-unrolling of structure from *(a)*

Memoizing hash codes offers an important way to improve performance. Memoization of hash codes is an obvious choice when the object is immutable. Our system will automatically memoize hash codes for objects that it is told are immutable. However, techniques for ensuring that objects are immutable in imperative object-oriented programming languages are still maturing (*e.g.*, [95, 125]). Therefore, in the experiments described in this chapter we do not perform memoization. Combining our system with one for ensuring object immutability offers an obvious opportunity for improved performance in the future.

5.2 Design Space

We identify a two-dimensional design space for object hashing strategies. The first dimension is the amount of the object's state that is examined: *none*, *some*, or *all*. The second dimension is kind of cycles detected: *none*, *physical*, or *logical*. Table 5.1 gives an overview of this design space and the hashing strategies we discuss in this chapter; shaded cells indicate points for which we have designed and evaluated hash strategies.

The cells that are not shaded indicate points in the space for which it is impossible to design a strategy that meets our design goals. The cells marked 'n/a' are nonsensical: if none of the state is examined, then no cycle detection is needed. As discussed in the previous section, examining all of the object's state with anything less than logical cycle detection leads either to non-termination or results that are inconsistent with equals, making it impossible for such a strategy to meet our design goals.

Table 5.1 Design space for object hashing strategies

State	Cycle Detection		
Examined	None	Physical	Logical
None	Constant	n/a	n/a
Some	Finite Unrolling	• Constant When Cyclic • Ignore Cycles	Canonical Set
All	naïve (incorrect)	incorrect	Canonical Tree

There is one strategy that is commonly used by practicing programmers that is not included in Table 5.1: examining only a subset of an object's state that is guaranteed to be acyclic. Mechanizing a strategy such as this would require a process similar to the optimization condition generation and discharge described above in Chapter 3: at the least it would entail either a program verification system or a deep program analysis.

5.3 Object Hashing Strategies

This section describes the hashing strategies named in Table 5.1. We begin in the upper-left corner of the table, and move through the labelled strategies in order toward the bottom-right of the table; for each strategy, we provide a pseudocode listing and argue that the strategy satisfies the design goals. Figure 5-3 describes the notation that we use in the strategy pseudo-code listings.

Recall that a hash strategy has the type $R \rightarrow [\mathbb{Z}]$ (where R is the set of references, or objects, and $[\mathbb{Z}]$ is a list of ints). A classic hash function is then applied to the output of the hash strategy; so the type of a classic hash function is $[\mathbb{Z}] \rightarrow \mathbb{Z}$. A hash method is a function applied to the result of a strategy and hence has the type $R \rightarrow \mathbb{Z}$.

While the presentation of the strategies here is of type $R \rightarrow [\mathbb{Z}]$, and so involves the explicit construction of a list of integers, in an actual implementation (including ours) it is likely that the programmer would compose the classic hash function with the hashing strategy so as to avoid explicitly constructing this list. This composition is trivial if one uses the simple classic hash functions recommended by Bloch [17], for example.

Figure 5-3 Notation for hash strategy listings

Term	Description
[]	empty list
R	set of all references (objects)
P	set of all primitives
V	set of all values ($R \cup P$)
$\mathcal{A}(r)$	object abstraction function ($R \rightarrow [\alpha]$)
$\alpha(r)$	field abstraction function ($R \rightarrow [V]$)
++	list concatenation operator
$hashp(p)$	hash a primitive ($P \rightarrow \mathbb{Z}$)

5.3.1 Constant Value

Returning a constant value, such as 42, executes in constant time and trivially satisfies the termination and consistency correctness properties, since it assigns the same hash value to *all* objects. This strategy produces a high number of collisions.

5.3.2 Finite Unrolling

The Finite Unrolling hashing strategy (Figure 5-4) traverses the infinite tree unrolling of its argument only up to the finite depth d. The returned hash value is composed of the hash values of the primitives in that tree.

Finite unrolling does not loop on cyclic object structures because the bound on the computation is not a function of the object structure. Finite unrolling is consistent with equals: identical tree unrollings remain identical when pruned at a particular depth; the Finite Unrolling strategy prunes all unrollings at the same depth, and thus yields the same hash values for two identical tree unrollings.

Figure 5-4 Finite Unrolling Hash Strategy

```
finiteUnrolling(r, d) :
  if d ≤ 0 :
    return [42]
  else :
    h ← []
    ∀ α ∈ 𝒜(r) :
      ∀ v ∈ α(r) :
        if isPrimitive(v) :
          h ← h ++ [hashp(v)]
        else :
          h ← h ++ finiteUnrolling(v, d − 1)
    return h
```

5.3.3 Constant When Cyclic

The hashing strategy Constant When Cyclic (Figure 5-5) examines all of the state for tree-like object structures and returns a constant value for object structures that contain physical cycles.

This strategy always terminates on finite object structures, since it returns immediately on detecting a physical cycle. It is also consistent with equals: when no cycles are present, it examines the entire tree unrolling of its argument; when cycles are present it reduces to the Constant Value strategy, which is consistent with any equality method.

Figure 5-5 Constant When Cyclic Hash Strategy

```
constantWhenCyclic(r) :
  try h ← recursiveHashWithPhysicalCycleDetection(r, ∅)
  catch PhysicalCycleException
    return [42]
  else
    return h

recursiveHashWithPhysicalCycleDetection(r, S) :
  h ← []
  if r ∈ S :
    throw PhysicalCycleException
  else :
    ∀ α ∈ 𝒜(r) :
      ∀ v ∈ α(r) :
        if isPrimitive(v) :
          h ← h ++ [hashp(v)]
        else :
          h ← h ++ recursiveHashWithPhysicalCycleDetection(v, S ∪ {r})
    return h
```

5.3.4 Ignore Cycles

The Ignore Cycles hashing strategy (Figure 5-6) is a slight variation of the Constant When Cyclic strategy. The difference is in the handling of cycles. When Constant When Cyclic detects a physical cycle, it returns a constant for the entire object structure. When Ignore Cycles detects a physical cycle, on the other hand, it produces a constant only for the sub-structure with the cycle.

A cycle in the original object structure of an object appears in its tree unrolling as a repeated node. This hash strategy ignores parts that contain cycles, meaning that the strategy does not examine any branch of the unrolling that contains a repetition. That is, the strategy examines *only the finite-length branches* of the unrolling, ensuring both termination and examination of the same set of nodes for identical unrollings.

Figure 5-6 Ignore Cycles Hash Strategy

```
ignoreCycles(r) :
  h ← []
  ∀ α ∈ 𝒜(r) :
    ∀ v ∈ α(r) :
      if isPrimitive(v) :
        h ← h ++ [hashp(v)]
      else :
        try t ← recursiveHashWithPhysicalCycleDetection(v, {r})
        catch PhysicalCycleException :
          noop
        else :
          h ← h ++ t
  return h
```

5.3.5 Canonical Set

The Canonical Set hashing strategy (Figure 5-7) selects one object per equivalence class of the objects reachable from its argument and composes the hash values of their corresponding primitive parts. The implementation achieves this by mutating a global set S of logically distinct objects visited during the traversal.

Canonical Set is consistent with equality if it is paired with an associative and commutative hashing function. The other strategies discussed in this paper do not place restrictions on the hashing functions they are paired with. If two objects are equal, the set of equivalence classes of objects reachable from them must be equal. Note that this is a *set* of equivalence classes: sets are unordered; hence the requirement that this strategy be paired with a hash function that is oblivious to the ordering of its input sequence.

This strategy detects logical cycles and therefore terminates for finite object structures.

Figure 5-7 Canonical Set Hash Strategy

```
canonicalSet(r) :
  global S ← ∅
  buildSet(r)
  h ← []
  ∀ o ∈ S :
    ∀ α ∈ 𝒜(o) :
      ∀ v ∈ α(o) :
        if isPrimitive(v) :
          h ← h ++ [hashp(v)]
  return h
```

```
buildSet(r) :
  for each α ∈ 𝒜(r) :
    for each v ∈ α(r) :
      if isObject(v) :
        if (no x ∈ S | x.equals(v)) :
          S ← S ∪ {v}
          buildSet(v)
```

5.3.6 Canonical Tree

The Canonical Tree hashing strategy (Figure 5-8) transforms the object graph into a (finite) canonical representation of its (potentially infinite) tree unrolling, which we call its *canonical tree*. Finite paths are unrolled completely. Infinite paths are pruned when a logical cycle is detected. Thus the Canonical Tree strategy examines all of an object's state without the possibility of divergence.

The canonical tree of the example in Figure 5-1a is $x \rightarrow 7$, and the canonical tree of the example in Figure 5-1b is $y \rightarrow 7$. In the former case, the $x \rightarrow x$ edge is pruned, and in the latter case the $y \rightarrow z$ edge is pruned: both of these edges represent logical cycles. Similarly, Figure 5-2b depicts the canonical tree of the object structure shown in Figure 5-2a.

The Canonical Tree strategy prunes infinite branches of its argument's tree unrolling before the first repetition, making all branches of the unrolling finite. Since this pruning is done based on logical cycles, two identical unrollings will have identical (finite) prunings, and thus will yield the same hash code. Since the resulting tree is finite, termination is guaranteed.

Figure 5-8 Canonical Tree Hash Strategy

```
canonicalTree(r, S) :
  h ← []
  if (∃x ∈ S | x.equals(r)) :
    return []
  else :
    ∀ α ∈ 𝒜(r) :
      ∀ v ∈ α(r) :
        if isPrimitive(v) :
          h ← h ++ [hashp(v)]
        else :
          h ← h ++ canonicalTree(v, S ∪ {v})
    return h
```

5.4 Evaluation

In this section we present empirical evaluations of the hashing strategies described above. Our evaluations involve both specially constructed micro-benchmarks and experiments involving real-world programs. We also present some conclusions based on our results.

5.4.1 Micro-Benchmarks

To develop a systematic understanding of the empirical performance of these hashing strategies we designed a set of micro-benchmarks. Each benchmark adds a number of

objects to a HashSet (using the put operation) and then performs a containment check (using the contains operation) on each object that was added. We record the total time taken for these operations.

These benchmarks characterize the space of possible programs by varying both the number of objects placed in a collection and the size of those objects. We define an object's size to be the number of objects reachable from it, and a collection's size to be the maximum number of key objects it may contain in a given benchmark. We ran our benchmarks on a crafted set of structures containing integers.

We generated both cyclic and acyclic structures of sizes ranging from 1 to 1000 using Korat [20, 82]. Korat is particularly suited to this task: it exhaustively generates all valid non-isomorphic structures up to a given bound, ensuring that all shapes are represented. Since these experiments required only a small number of structures, however, we ran Korat only until it generated the desired number. Korat allowed us to easily express desired structure properties (*e.g.* cyclicity; number of nodes; number of children per node) and automatically generated non-isomorphic structures.

For each hashing strategy, our micro-benchmarks evaluate both hashing speed and effectiveness in avoiding collisions. Adding an object to a HashSet requires the computation of that object's hash value; the time taken in this operation is recorded as part of the overall time. The subsequent containment test, on the other hand, uses an object's hash value to find it; in the presence of collisions, this process degrades to linear search, penalizing each hashing strategy in proportion to the number of collisions it causes.

Figure 5-10 summarizes the results of our experiments with a collection of heat-map plots. Each heat map represents one experiment; rows correspond to hashing strategies, while columns correspond to the variables of our micro-benchmark. The columns labelled "acyclic" represent experiments in which our micro-benchmark was run over a set of object structures containing no cycles, while those labelled "cyclic" represent experiments in which all objects contained cycles. The remaining column labelings are as follows:

- **Varying Shapes, Random Values**
  Generated object structures varied both in their topologies and in the integer values they contained.
- **Varying Shapes, Same Values**
  Generated object structures varied in topology, but contained the same integer values.
- **50% Equals, Same Values**
  Generated objects contained the same integer values; half of them also shared a topology with another object, making them equals to that object.

Figure 5-9 contains an example heat-map plot with its associated axes labelled and is intended as a legend for the plots in Figure 5-10. All of the plots in Figure 5-10 use the same logarithmic axes as the example in Figure 5-9.

In general, the amount of state examined was not a good predictor of hashing strategy performance. In fact, in the majority of cases, Finite Unrolling to a depth of 1 was the most efficient strategy; the use of more object state in Finite Unrolling to depths 2 and 3 actually

hurt performance in most cases. Even ignoring object state entirely, as in the Constant strategy, was a reasonable choice when collections were small.

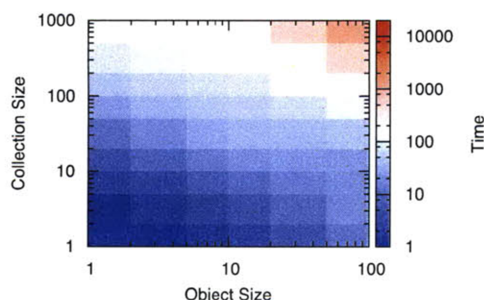
The Constant when Cyclic and Ignore Cycles strategies performed as badly (or worse, due to the overhead of cycle detection) as the Constant strategy in experiments involving cyclic objects; there was no significant difference between the two strategies.

The greater amount of object state used by the Canonical Tree strategy seems to have mitigated the overhead involved in cycle detection: Canonical Tree performed reasonably well in all experiments, though it was never as efficient as Finite Unrolling.

Most hashing strategies perform well when collection sizes are small. In fact, the hashing strategy seems to make very little difference for collections with fewer than 100 elements.

Finite Unrolling of depth 1 is the best general-purpose strategy on these micro-benchmarks. Potential future work could use these heatmaps for automatic customized strategy selection as follows: a program with an associated workload identifies specific points in this heatmap space (object size *vs.* collection size *vs.* cyclicity *etc.*); each point in this space has an optimal strategy; matching the program to the point in the heatmap space selects the strategy. In most, but not all, cases the selected strategy will be Finite Unrolling of depth 1.

Figure 5-9 Example Heat Map Plot



An example heat map representing one experiment. Object size is plotted along the x-axis, collection size along the y-axis, and time in milliseconds along the z-axis (colour). All scales are logarithmic; lower times are better. Red parts of the graph represent points where performance was poor, while blue areas represent good performance. The graphs in Figure 5-10 are rendered with the same scales and color schemes.

Figure 5-10 Heat maps plots of micro-benchmark experiments

	Acyclic			Cyclic		
	Varying Shapes Random Values	Varying Shapes Same Values	50% Equals Same Values	Varying Shapes Random Values	Varying Shapes Same Values	50% Equals Same Values
Constant						
Constant when Cyclic						
Ignore Cycles						
Canonical Tree						
Finite Unrolling (depth 3)						
Finite Unrolling (depth 2)						
Finite Unrolling (depth 1)						

5.4.2 Strategy Evaluation in The Wild

While the synthetic analysis above characterizes the space of possible programs and workloads, it does not tell us what common programs and workloads are actually like. To get an understanding of what real-world programs and workloads are like we evaluated the hashing strategies on the following programs and workloads:

- The JDK Collections Library 1.4 with workloads from the JCK (Java Compliance Kit) and the Randoop tool [93, 94] (Figure 5-11).
- The Apache Commons Collections Library 3.2 with hand-written vendor tests and tests generated by Randoop (Figure 5-11).
- The JFreeChart graphing program from the DaCapo benchmark suite 2006 with five different workloads: DaCapo small, medium, large; hand-written vendor tests; and Randoop-generated tests (Figure 5-11).
- The batik, fop, pmd, and xalan benchmarks from the DaCapo 2009.12 suite on the medium and large DaCapo workloads (Figure 5-12). DaCapo 2009.12 contains a number of other benchmarks that were excluded for reasons described below.

Figure 5-11 confirms the general conclusion that Finite Unrolling (depth 1) is a good general-purpose hashing strategy. However, for all of the programs and workloads presented in Figure 5-11, other strategies are also reasonable choices: Finite Unrolling of depths 2 or 3; Constant when Cyclic; and Canonical Tree. Ignore Cycles is often a reasonable choice, but for the JFreeChart vendor tests and DaCapo large workload is a terrible choice.

Figure 5-11 shows that, surprisingly, returning a constant is a reasonable choice in most circumstances. Only on the Apache Collections vendor tests is it a terrible choice. We explore this result in more detail below.

Figure 5-12 compares the original programmer-written hashing methods with the Finite Unrolling (depth 1) on the batik, fop, pmd, and xalan benchmarks from the DaCapo suite 2009.12. Finite unrolling always outperforms the programmer-written hashing methods on the batik benchmark, by a margin of about 10%. The programmer-written hashing methods perform better in all other cases except pmd on the large workload with no warmup run. This exceptional case, and the contrast between the times for no warmup and four warmup runs, highlights how important JIT compilation technology is.

The data in Figure 5-12 suggest that the programmer-written hashing methods are much better than Finite Unrolling on the xalan benchmark. We believe that this is a reflection of the overhead of our framework more than a valid comparison of a programmer-written hashing method and Finite Unrolling. There is only one programmer-written hashing method in xalan that is heavily exercised by the DaCapo workloads: CharInfo$CharKey from the org.apache.xml.serializer package. This class is just a wrapper for a primitive char, and its hash method simply returns the char widened to an int. Finite Unrolling, in principle,

does exactly this as well. However, our framework adds a substantial amount of overhead to what is essentially a widening conversion.

A production quality implementation of our framework should produce identical runtimes for xalan. The fact that our research quality implementation sometimes outperforms hand-written hashing methods supports the general point that programming language implementors are the ones who should be hashing objects, rather than application programmers.

Excluded Benchmarks from DaCapo 2009.12. The avrora, luindex, and lusearch benchmarks were excluded because of inconsistencies between their hashing and equality methods. Our abstraction function framework is not capable of expressing these inconsistencies. We suspect that these inconsistencies are programmer errors, but verifying that conjecture is beyond the scope of this chapter.

The jython benchmark was excluded because, as a programming language implementation, it often needs to work with objects at a low-level not supported by our abstraction function framework (which is designed for application programs).

The sunflow benchmark was excluded because its workloads do not exercise any user-defined hashing methods.

The eclipse, h2, tomcat, tradebeans, and tradesoap benchmarks were excluded for technical reasons. For example, in some cases perturbing the hashing methods at all causes the workloads to fail because they are hard-wired to expect certain results.

Figure 5-11 Performance evaluation

Times in milliseconds	Randoop Tests			Vendor Tests			DaCapo Input			*Averages*
	Before	*After*	*Change*	*Before*	*After*	*Change*	*Before*	*After*	*Change*	
JDK	7374	7699	+4%	4410	3837	-13%	.	.	.	-4%
Apache	2231	2514	+13%	2479	5049	+104%	.	.	.	+58%
JFreeChart	669	986	+47%	20382	21217	+4%	21355	21827	+2%	+18%
Averages			+21%			+32%			+2%	+23%

Figure 5-12 Programmer-written hashing methods versus Finite Unrolling (depth 1) comparison in DaCapo benchmarks

Times in Seconds	Medium WL (no warmup)		Medium WL (4 warmups)		Large WL (no warmup)		Large WL (4 warmups)	
	User	Unroll	User	Unroll	User	Unroll	User	Unroll
batik	5.91	**5.36**	1.69	**1.61**	8.78	**7.83**	3.58	**3.25**
fop	4.11	5.42	0.58	1.76	-	-	-	-
pmd	7.71	8.70	3.01	3.10	13.03	**11.95**	5.17	5.28
xalan	8.48	9.03	2.60	3.73	36.27	43.52	26.29	30.82

5.4.3 The Worst Possible Strategy

```
// The worst possible legal hash function - never use!
@Override public int hashCode() { return 42; }
```

— *Effective Java* [17]

Figure 5-11 above contains the surprising result that returning a constant is sometimes a reasonable hashing strategy. In Figure 5-13 we explore this result in more depth.

Conventional wisdom dictates that an object's hashcode should represent as much information about it as possible, since hash collisions may make containment tests and lookup operations more expensive. Consequently, returning a constant is sometimes referred to as "the worst possible legal hash function," and programmers are advised to "never use" it [17].

However, as discussed above, there is a trade-off between the time spent examining the object's state and the time saved by reducing hash collisions. It may be the case that, in some circumstances, returning a constant is actually a reasonable strategy. For example, as the micro-benchmark results in the heatmaps of Figure 5-10 show, returning a constant is a fine strategy when the objects are used only in collections with a single element. It takes at least two elements to have a hash collision.

Figure 5-13 evaluates returning a constant against the programmer-provided hashing methods for all but two of the benchmark programs in the DaCapo 2009.12 suite. (The excluded programs are eclipse and sunflow, which depend on specific results from certain hashing methods in order to complete the workloads.)

The instrumentation used to produce Figure 5-13 was done without using the framework that we employ in the rest of this dissertation. For Figure 5-13 we did a simple bytecode transformation to replace the hashCode method in every programmer-defined class with returning a constant. This instrumentation approach is applicable to any program and incurs no framework-related runtime overhead.

Surprisingly, in 15 out of 35 cases of Figure 5-13, returning a constant outperforms the programmer-provided hashing methods. The improvement is sometimes significant, as in the case for lusearch on the medium workload, which reduces 9.253s to 6.398s; or the jython small workload, which reduces 1.585 seconds to 0.505s.

Perhaps even more surprisingly, returning a constant is rarely terrible. Only for the h2 and jython benchmarks on medium and large workloads do we see the kind of exponential performance degradation that one might expect from having all hashing methods implemented as 'return 42'.

These results suggest that it may be reasonable guidance to encourage application programmers to use 'return 42' as their default hashing method implementation. Returning a constant has the great virtues that it is easy to implement and is always correct. Using this strategy is equivalent to replacing hash tables with linked lists. The data in Figure 5-13 suggest that the effort that application programmers are putting into writing customized hashing methods for their classes is misspent in most cases.

Figure 5-13 Performance of returning a constant (the "worst possible hash function")

Times in Seconds	Small Workload User	Small Workload Return1	Medium Workload User	Medium Workload Return1	Large Workload User	Large Workload Return1
avrora	2.609	2.613	5.457	**5.243**	27.242	28.378
batik	1.976	2.101	4.682	5.069	9.965	10.155
fop	1.817	**1.339**	3.970	**3.395**	-	-
h2	1.887	2.735	7.995	11.437	44.006	713.862
jython	1.585	**0.505**	13.675	360.945	43.977	2,574.207
luindex	0.587	**0.581**	2.775	**2.186**	2.265	**2.069**
lusearch	2.251	2.625	9.253	**6.398**	13.770	**12.788**
pmd	0.416	**0.380**	7.079	8.153	12.362	**11.217**
tomcat	5.686	5.741	10.275	**9.943**	34.354	36.950
tradebeans	3.678	3.725	13.386	**12.663**	33.851	**33.162**
tradesoap	21.246	24.579	43.416	**42.402**	103.946	104.875
xalan	2.737	3.174	7.356	7.787	31.547	33.519

[1] We didn't instrument "eclipse" and "sunflow" benchmarks because those two jars were signed, and so any modifications to their code causes the tests to fail.
[2] Large workload for "fop" benchmark doesn't exist

5.5 Conclusions

The textbook study of classic hashing *functions* has been concerned with hashing immutable bitstrings to an integer. Such classic hashing functions are an important component of an object hashing *method*, but they are not the only component. An object hashing method also needs an object hashing *strategy* to convert the (potentially cyclic) object graph into a bitstring suitable for input to a classic hashing function. This object hashing strategy may in turn require the use of an object abstraction function in order for the hashing method to be consistent with equality. The correctness and performance of an object hashing method are determined largely by its hashing strategy rather than by its hashing function.

The dominant design issues for correctness are consistency with equality and cyclic object structures. The former can be easily handled through the use of abstraction functions. The latter is an integral part of the design of the hashing strategy. The common technique of building up an assumption table that is used to handle cyclic structures during an equality comparison cannot be used during hashing because hashing is a unary operation. Alternative cycle management techniques must be employed. Moreover, to be correct an object hashing strategy must handle not only physical cycles but also logical cycles.

Practical performance of object hashing methods varies with both object size and collection size. The textbook study of classic hash functions usually focuses on large collections of small immutable objects. However, many object-oriented program executions contain small collections of large mutable objects. (The hashcode of a mutable object cannot be

cached and so must be recomputed each time it is needed.) Under these circumstances some of the conventional wisdom about hashing no longer holds: we have found that it is usually better to examine less of the object state, and thereby have a higher chance of collisions, than to focus on reducing potential collisions.

At the extreme we have found that replacing programmer-written hashing methods with returning a constant can actually improve program performance in some cases. In some cases, of course, returning a constant result in a dramatic decrease in program performance — but not in as many cases as one might have expected. Application programmers using a language that does not yet provide default hashing methods that are consistent with programmer-defined equality methods may reasonably consider returning a constant until profiling shows that this hashing strategy is actually degrading overall program performance.

We have found that the best default hashing strategy is Finite Unrolling of depth 1: it may not always be the best, but it is never terrible. This strategy examines enough of the object state to avoid extensive collisions in most cases and it avoids expensive cycle detection. For immutable objects where the hashcode can be cached then it may be worth using a strategy such as Canonical Tree that examines more of the object state. However, immutable classes are uncommon in object-oriented programs, and ways for application programmers to reliably communicate immutability to language implementors are still maturing [95, 125].

In general we find that application programmer effort is probably better spent on problem-domain specific aspects of their program, rather than attempting to implement clever hashing methods.

In future, once the language runtime has control over hashing strategy selection then performance improvements may be possible. For example, some analysis of the program could be used to determine which hashing strategy to use for each class. In this paper we only looked at scenarios where the same strategy was used for every class in a given execution. Even with this primitive approach we saw performance improvements of up to 10% in some cases. This idea is similar in approach to the complementary goal of automatic data structure selection [73, 103, 104], which has recently been shown to reduce the memory footprint of some Java programs by up to 30% [104]. There are a wide range of opportunities for tuning programs through high-level optimizations such as automatic data structure selection and automatic hashing strategy selection.

Chapter 6

Conclusions

This dissertation has argued that executable abstraction functions enable object-oriented programming languages to provide better support for object-contract compliance, resulting in clearer and more concise programs with fewer errors that still have acceptable performance. We have supported this thesis by developing:

- four approaches for executing abstraction functions written in the JForge [32, 119] variant of the Alloy [52] declarative logic;
- a library of logical predicates that make abstraction functions both more concise and more amenable to execution;
- a generic notion of object equality defined in terms of abstraction functions;
- a handful of object hashing strategies and their evaluation.

We have found empirically that our system is capable of synthesizing a replacements for a majority of hand-written object-contract methods and iterators from declarative abstraction functions. Were an industrial quality version of our system to be included as part of a programming language runtime, application programmers might spend less time and effort on object-contract compliance, and therefore a greater proportion of their time and effort on the domain-specific dimensions of their programs.

Providing a dynamic interpretation for abstraction functions may encourage programmers to write specifications, just as dynamic uses for invariants has. There is a long history of intellectual arguments for writing formal specifications for software. In practice, though, the benefits of specifications are not realized as often as they might be. We hope that his work provides another starting point for programmers to realize the broader benefits of specifications. We have elsewhere [100] contributed to this discussion by illustrating how executable specifications might be used together with code to find some methodological middle ground between agile and formal methods: for example, mock-objects [75] could be synthesized from executable specifications, and executable specifications could make it easier for the programmer to take a test-driven [13] approach to specification development.

6.1 Zave Evaluation Criteria

In reflecting on the PAISLey [120] system of executable specifications for concurrent systems, Pamela Zave proposed five questions to evaluate executable specification languages. She intended these questions for more general systems than ours — our system has a very particular focus — but nevertheless the questions are worth examining.

1. *Is the language expressive enough for the functional specification of the intended class of system?*

   We evaluated expressiveness from two different angles. First, in Chapter 3 we looked at the ability of our JFSL-based system to express the abstraction functions needed for the JDK Collections v1.4 data structure library. We found that 25 out of the 29 hand-written iterators could be synthesized from declarative abstraction functions by our system. The remaining cases generally involved the hand-written iterator returning objects that did not exist at the time the iteration began. Our system can be viewed as a query evaluator for the extant heap: it does not synthesize new objects during its traversal, and it is limited to return objects discovered during that traversal.

   In Chapter 4 we looked at how many of the equals and hashCode methods could be replaced with our generic, abstraction function based, versions. We found that our generic versions could replace the hand-written versions in almost all cases. The remaining cases involved classes that intentionally violated the object contract or used weak references to control interaction with the garbage collector.

   Our system is sufficiently expressive for the majority of classes that programmers in practice need to write. In the cases our system doesn't cover the programmer is still free to implement iterators and object contract methods by hand.

2. *Does the language support the formal reasoning necessary for validation of the intended class of system?*

   Use of our system improves the both the options for formal reasoning about the program and the correctness of the program versus hand-written implementation of object-contract methods. The abstraction function that our system requires the programmer to write can also be reused as part of the specifications of the invariants, pre-conditions, and post-conditions of the program.

   In Chapter 2 we developed a library of logical predicates that make it easier for programmers to write complex abstraction functions involving sequences and trees. Even without the executability aspect of our system these predicates make it easier for programmers to specify and hence verify their programs.

   In Chapter 4 we saw that shipping production code from expert programmers contains bugs in object contract methods, and that replacing these hand-written implementations with our system removed these bugs.

3. *Does the language preserve the implementation freedom needed to meet the nonfunctional requirements of the intended class of system?*

Our system does not prevent the programmer from implementing iterators and object-contract methods by hand, and so the programmer's freedom is not impinged by our system.

The programming language implementor, on the other hand, gains a new dimension of freedom to improve program performance from the use of our system. By bringing hashing strategy selection under control of the language runtime, as discussed in Chapter 5, the language implementor has a new opportunity to systematically improve program performance.

4. *Is there a method for construction, validation, and implementation of specifications in the language?*

   While we have not deeply explored the methodological dimensions of this work, our experience with our system suggests two things: first, simple hand-written unit tests are usually sufficient to validate that the abstraction function meets programmer intent; second, automated testing tools that specifically target object-contract compliance, such as Randoop [93, 94], are useful for more systematic validation.

5. *Is use of the language cost-effective for the intended class of system?*

   There are three costs to be considered: the cost of producing the program, the time required to run the program, and the space required to run the program.

   The space consumed by an industrial quality implementation of our system should be minimal: it would be a linear function of the number of classes in the system and therefore constant in the number of objects in a given execution.

   In Chapter 3 we saw that our synthesized iterators are slower than hand-written ones, but that there is also room for improvement. In Chapter 5 we saw that changing the hashing strategy can have a noticeable impact on overall program running time. Since our system brings the choice of hashing strategy under the control of the language runtime there is an opportunity here to systematically improve running time. This opportunity is not available when object-contract methods are implemented by hand.

   While we do not have quantitative data specifically on production cost, we argue that the evidence suggests that using our system is less expensive than implementing iterators and object contract methods by hand. It is certainly the case that a declarative abstraction function is more concise than the hand-written iterator, equality comparison, and hashing method that it replaces. If lines of programmer-written code are a measure of production cost, then our system, where applicable, would seem to be less expensive than implementing by hand. Furthermore, there is ample evidence in the literature [17, 49, 91, 93, 94, 97, 114], and in Chapter 4, that programmers of all skill levels have difficulty attaining object-contract compliance. In Chapter 4 we show that our synthesized code has fewer bugs than hand-written code — even industrial quality code hand-written by expert programmers. This evidence suggests that by using our system the programmer will save both development time and debugging time, producing a more concise program with fewer flaws.

6.2 The Programmer as Relational Navigator

Among the first approaches to computerised database management systems were the *network* and *hierarchical* models, epitomized by General Electric's Integrated Data Store (IDS) and IBM's Information Management System (IMS), respectively. These models have been referred to as *navigational* [7] because queries are written in imperative form as a sequence of operations navigating references between records. These database systems are important commercially and academically, in addition to historically: Charles Bachman won the 1973 Turing Award for his work on IDS [7], and despite being introduced in 1968 as part of the Apollo space program [16], IMS had its most successful sales year in 2003 [50].

In 1970 E.F. Codd introduced relational database management systems [25]. Codd's proposal had a number of dimensions. He argued that data should be conceptualized as sets of tuples (*i.e.*, relations), rather than a graph of records. He further argued that queries should be expressed in a declarative logic, and that the database system should automatically determine how to translate these declarative expressions into navigations of the underlying storage. Codd won the 1981 Turing Award for his pioneering work on relational databases [26].

The heap of an object-oriented program is like an IDS database: a graph of references between objects/records. And currently object-oriented programmers write their programs in much the same way that IDS programmers did almost fifty years ago, as a sequence of imperative navigations.

One way to look at this dissertation is that it brings the idea of declarative relational logic queries to data that is physically structured in a graph rather than in relations. In other words, our Alloy query engine for Java programs is like a (hypothetical) SQL query engine for IDS. Continuing in the vein of this comparison, the main optimization that we perform is to determine whether the Java program under consideration represents an IMS database (*i.e.*, hierarchical) or an IDS database (*i.e.*, unrestricted graph).

It so happens that most abstraction functions that we have observed in practice are essentially queries on the program heap. So while we focus on abstraction functions and uses for their dynamic interpretation, the fundamental technology is really a query engine. The technical issues in our query engine differ from traditional relational database query engines because: (a) our entire 'database' is in memory rather than on disk; (b) the data are stored as a graph rather than as row-store or column-store relations; and (c) the query always starts from the distinguished this object.

Both Bachman [7] and Codd [26] hoped to improve programmer productivity through two main principles: (1) separating the application code from the data management code; and (2) giving the programmer clearer, more concise, and more powerful tools for working with their data. For my part, I hope to replay their shared vision in miniature, with the same motivations and principles, and hopefully similar (albeit smaller) attendant benefits.

6.3 Future Directions

There are a variety of future directions that could be pursued from this work.

For programmers in the field, a more industrial grade implementation would help them realize the practical benefits of our system. Such an implementation would improve obvious deficiencies, such as performance and error message quality. It would also enhance some features that we have not discussed in this dissertation, such as automatic synthesis of toString and compareTo methods (from additional annotations). Enhancing debuggers to give programmers a view of the abstract state of their objects would also be useful. The Eclipse Java debugger already contains a rudimentary ad-hoc implementation of this feature, suggesting that there is demand for a more systematic solution.

We think practicing programmers would also appreciate some minor modifications to the JForge Specification Language, such as syntactically separating specification field declarations from abstraction function definitions, and perhaps adding explicit syntactic expression for equality types.

We also have some ideas for future research. For example, it would be interesting to see the performance gains that could be had by automatic selection of hashing strategies, and further by combining this with automatic data structure selection.

Our optimization condition generation process derives potential program invariants from an abstraction function. It would be interesting to know if these derived potential invariants are a super-set of the programmer's hand-written invariants. Such an experiment could be conducted with the manually specified and verified data structures from the Jahob project [59, 123]. If there are hand-written invariants that are not included in the automatically derived set, it would be interesting to learn what they are and why, and to see if they could also be automatically derived from an abstraction function through a modified version of our optimization condition generation technique.

Implementations of abstract data types often maintain a separate concrete field that records the size of their main specification field: for example, the number of elements in a set or the length of a linked list. Ensuring that this size field is accurate consumes programmer effort. It would be easier to simple define a specification field and abstraction function as follows: this.size = #(this.nodes). Programmers can do that today with our system. However, today our system would not evaluate this expression in the most efficient manner. What would be interesting is to take this specification field and abstraction function and automatically synthesize a concrete field and the code to maintain its correct value incrementally. For example, in a linked-list insertion procedure to automatically synthesize code to increment the concrete size field. Code synthesis techniques such as those developed by Solar-Lezama [106] might be applicable here since we have both a working but innefficient implementation of the desired functionality (our current system) and a 'sketch' of the efficient implementation in the form of the hand-written linked-list insertion procedure that does not maintain the value of the size field.

It is my conjecture, at the conclusion of this work, that data abstraction and object ownership [4, 19, 90] are closely related: the owned objects may be named by a (perhaps private)

specification field and its associated abstraction function. In our ongoing example of a binary search tree we might say that the tree owns its nodes: that is, the tree object owns all of the objects named by the abstraction function this.nodes = this.root.*(left+right). Using abstraction functions in this way would likely reduce the programmer's annotation burden as compared to ownership type systems: the number of specification fields in a program is a function of the number of classes, whereas the number of ownership annotations is a function of the lines of code; moreover, if the programmer is already writing the specification field and abstraction function for other reasons then ownership is an added benefit negligible marginal cost.

If this connection between ownership and data abstraction can be substantiated then perhaps it can also be used, in combination with the work of this dissertation, for object cloning. With today's technology object cloning tends to be either *shallow*, which clones just the object under consideration, or *deep*, which clones not only the object under consideration but also all objects reachable from it. In most cases the programmer desires something in between. For example, when cloning a linked list the programmer usually desires to have a new list that refers to the same external data objects as the original list. Grogono and Sakkinen [38] explored some potential solutions to this problem that involve annotations tailored to this task. The combination of our dynamic interpretation of abstraction functions along with deriving ownership from abstraction functions might give programmers the cloning operations they want for no additional programmer effort.

Bibliography

[1] Martín Abadi and Leslie Lamport. The existence of refinement mappings. In *Proceedings of the 3rdIEEE Symposium on Logic in Computer Science (LICS)*, pages 165–175, Edinburgh, Scotland, July 1988. Cited on page 15.

[2] Serge Abiteboul and Jan Van Bussche. Deep equality revisited. In T.W. Ling, A.O. Mendelzon, and L. Vieille, editors, *Deductive and Object-Oriented Databases*, volume 1013 of *Lecture Notes in Computer Science*, pages 213–228. Springer-Verlag, 1995. Cited on pages 73, 82, and 88.

[3] Michael D. Adams and Kent Dybvig. Efficient nondestructive equality checking for trees and graphs. In Peter Thiemann, editor, *Proceedings of the 13thACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 179–188, Victoria, British Columbia, Canada, September 2008. ACM Press, NYC, NY. Cited on pages 87 and 88.

[4] Jonathan Aldrich. *Using Types to Enforce Architectural Structure*. PhD thesis, University of Washington, August 2003. Cited on page 107.

[5] Saman Amarasinghe, editor. *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, Tucson, Arizona, USA, June 2008. Cited on page 119.

[6] Pierre America. Designing an object-oriented programming language with behavioural subtyping. In *REX Workshop on Foundations of Object-Oriented Languages*, volume 489 of *Lecture Notes in Computer Science*, pages 60–90, 1990. Cited on page 83.

[7] Charles W. Bachman. The programmer as navigator. *Communications of the ACM*, 6(11), 1973. Turing Award Lecture. Cited on page 106.

[8] Ralph-Johan Back. *On the Correctness of Refinement Steps in Program Development*. PhD thesis, University of Helsinki, 1978. Report A–1978–4. Cited on page 19.

[9] Ralph-Johan Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag, 1998. Cited on pages 15 and 19.

[10] DAVID F. BACON. Kava: A Java dialect with a uniform object model for lightweight classes. *Concurrency—Practice and Experience*, 15(3–5):185–206, March–April 2003. Cited on page 82.

[11] M. BARNETT, R. DELINE, M. FAHNDRICH, K. R. M. LEINO, AND W. SCHULTE. Verification of ob ject-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004. Cited on page 13.

[12] M. BARNETT, K. R. M. LEINO, AND W. SCHULTE. The Spec# programming system: An overview. In G. BARTHE, L. BURDY, M. HUISMAN, J.-L. LANET, AND T. MUNTEAN, editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart devices (CASSIS 2004)*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69, New York, NY, 2005. Springer-Verlag. Cited on page 13.

[13] KENT BECK. *Test-Driven Development*. Addison-Wesley, Reading, Mass., 2003. Cited on page 103.

[14] A.D. BIRRELL, J.V. GUTTAG, J.J. HORNING, AND R. LEVIN. Thread synchronization: A formal specification. In Nelson [88]. Cited on page 14.

[15] S. M. BLACKBURN, R. GARNER, C. HOFFMAN, A. M. KHAN, K. S. MCKINLEY, R. BENTZUR, A. DIWAN, D. FEINBERG, D. FRAMPTON, S. Z. GUYER, M. HIRZEL, A. HOSKING, M. JUMP, H. LEE, J. E. B. MOSS, A. PHANSALKAR, D. STEFANOVIĆ, T. VANDRUNEN, D. VON DINCKLAGE, AND B. WIEDERMANN. The DaCapo benchmarks: Java benchmarking development and analysis. In PERI TARR AND WILLIAM R. COOK, editors, *Proceedings of the 21st ACM/SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Portland, Oregon, October 2006. Cited on pages 77 and 86.

[16] K. R. BLACKMAN. IMS celebrates thirty years as an IBM product. *IBM Systems Journal*, 37(4), 1998. Cited on page 106.

[17] JOSHUA BLOCH. *Effective Java*. Addison-Wesley, Reading, Mass., 2001. Cited on pages 8, 20, 66, 68, 69, 70, 82, 90, 100, and 105.

[18] CHARLES BOUILLAGUET, VIKTOR KUNCAK, THOMAS WIES, KAREN ZEE, AND MARTIN RINARD. Using first-order theorem provers in the jahob data structure verification system. In BYRON COOK AND ANDREAS PODELSKI, editors, *Proceedings of the 8th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI)*, volume 4349 of *Lecture Notes in Computer Science*, Nice, France, January 2007. Springer-Verlag. ISBN 978-3-540-69735-0. Cited on page 26.

[19] CHANDRASEKHAR BOYAPATI. *SafeJava: A Unified Type System for Safe Programming*. PhD thesis, MIT Electrical Engineering and Computer Science, 2004. Cited on page 107.

[20] CHANDRASEKHAR BOYAPATI, SARFRAZ KHURSHID, AND DARKO MARINOV. Korat: Automated Testing Based on Java Predicates. In PHYLLIS FRANKL, editor, *Proceedings*

of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), Rome, Italy, July 2002. Cited on pages 86 and 95.

[21] LILIAN BURDY, YOONSIK CHEON, DAVID COK, MICHAEL D. ERNST, JOSEPH R. KINIRY, GARY T. LEAVENS, K. RUSTAN M. LEINO, AND ERIK POLL. An overview of JML tools and applications. *Software Tools for Technology Transfer*, 7(3):212–232, June 2005. Cited on pages 13, 26, and 84.

[22] FELIX CHANG. *Generation of Policy-rich Websites from Declarative Models*. PhD thesis, MIT Electrical Engineering and Computer Science, 2009. Cited on page 24.

[23] YOONSIK CHEON. *A Runtime Assertion Checker for the Java Modeling Language*. PhD thesis, Department of Computer Science, Iowa State University, April 2003. Technical report 03-09. Cited on pages 7, 19, 26, and 27.

[24] YOONSIK CHEON AND GARY T. LEAVENS. The Larch/Smalltalk interface specification language. *ACM Transactions on Software Engineering and Methodology*, 3(3), July 1994. Cited on page 14.

[25] EDGAR F. CODD. Relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970. Cited on page 106.

[26] ———. Relational database: A practical foundation for productivity. *Communications of the ACM*, 25(2):109–117, 1982. Cited on page 106.

[27] IVICA CRNKOVIC AND ANTONIA BERTOLINO, editors. *Proceedings of the 6thEuropean Software Engineering Conference (ESEC) and ACM/SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, Dubrovnik, Croatia, September 2007. Cited on pages 116 and 119.

[28] O.-J. DAHL, B. MYHRHAUG, AND K. NYGAARD. The Simula 67 Common Base Language. Technical Report S-22, Norwegian Computing Center, Oslo, Norway, 1970. Cited on page 11.

[29] W. P. DE ROEVER AND K. ENGELHARDT. *Data Refinement: Model-Oriented Proof Methods and their Comparison*, volume 47 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1998. Cited on page 15.

[30] BRIAN DEMSKY, CRISTIAN CADAR, DANIEL ROY, AND MARTIN C. RINARD. Efficient specification-assisted error localization. In DAVID EVANS AND RAIMONDAS LENCEVICIUS, editors, *Proceedings of the Workshop on Dynamic Analysis (WODA)*, pages 60–67, Edinburgh, Scotland, 2004. Cited on page 48.

[31] BRIAN C. DEMSKY. *Data Structure Repair Using Goal-Directed Reasoning*. PhD thesis, MIT Electrical Engineering and Computer Science, January 2006. Cited on page 19.

[32] Greg Dennis. *A Relational Framework for Bounded Program Verification.* PhD thesis, MIT Electrical Engineering and Computer Science, 2009. Cited on pages 13, 14, 15, 21, 24, 25, 26, 28, 42, 44, 50, 57, 84, and 103.

[33] Amer Diwan, editor. *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, Dublin, Ireland, June 2009. Cited on pages 117 and 119.

[34] Niklas Een and Niklas Sörensson. An Extensible SAT-solver. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing*, 2003. Cited on page 24.

[35] B. Elkarablieh, I. Garcia, Y. Suen, and S. Khurshid. Assertion-based repair of complex data structures. In Alexander Egyed and Bernd Fischer, editors, *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Atlanta, GA, November 2007. Cited on page 19.

[36] Peter Forrest. The identity of indiscernibles. *Stanford Encyclopedia of Philosophy*, 2006. Cited on page 83.

[37] Norbert E. Fuchs. Specifications are (preferably) executable. *Software Engineering Journal*, 7(5):323–334, September 1992. Cited on page 19.

[38] Peter Grogono and Markku Sakkinen. Copying and comparing: Problems and solutions. In Elisa Bertino, editor, *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP)*, volume 1850 of *Lecture Notes in Computer Science*, pages 226–250, Cannes, France, June 2000. Springer-Verlag. Cited on pages 73, 82, 88, and 108.

[39] John V. Guttag. *The Specification and Application to Programming of Abstract Data Types.* PhD thesis, University of Toronto, Department of Computer Science, 1975. Cited on page 11.

[40] John V. Guttag, James. J. Horning, Stephen J. Garland, K. D. Jones, A. Modet, and Jeanette M. Wing. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag, 1993. Cited on pages 11, 13, 14, and 20.

[41] John Hatcliff, Gary T. Leavens, K. Rustan M. Leino, Peter Müller, and Matthew Parkinson. Behavioral interface specification languages. Technical Report CS-TR-09-01, University of Central Florida, March 2009. Cited on page 14.

[42] Ian Hayes and Cliff B. Jones. Specifications are not (necessarily) executable. *Software Engineering Journal*, 4(6):330–338, 1989. ISSN 0268-6961. Cited on page 19.

[43] J. He, C.A.R. Hoare, and J.W. Sanders. Data refinement refined. In *Proceedings of the European Symposium on Programming*, volume 213 of *Lecture Notes in Computer Science*, pages 187–196. Springer-Verlag, 1986. Cited on page 14.

[44] Eric C. R. Hehner. Do considered od: a contribution to the programming calculus. *Acta Informatica*, 11:287–304, 1979. Cited on page 19.

[45] Johannes Henkel, Christoph Reichenbach, and Amer Diwan. Developing and debugging algebraic specifications for java classes. *ACM Transactions on Software Engineering and Methodology*, 17(3), 2008. Cited on page 20.

[46] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1 (4):271–281, December 1972. Cited on pages 7, 11, 12, 13, 14, 15, and 16.

[47] ———. An overview of some formal methods for program design. *IEEE Computer*, 20(9):85–91, 1987. Cited on page 10.

[48] James J. Horning. The Larch Shared Language: Some Open Problems. In Magne Haveraaen, Olaf Owe, and Ole-Johan Dahl, editors, *Recent Trends in Data Type Specification: 11th Workshop on Specification of Abstract Data Types*, volume 1130 of *Lecture Notes in Computer Science*, Oslo, Norway, September 1995. Cited on pages 7, 11, and 16.

[49] David Hovemeyer and William Pugh. Finding bugs is easy. In Geoff Cohen, editor, *Proceedings of the ACM/SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) Onward!*, Vancouver, British Columbia, Canada, October 2004. Cited on pages 20, 66, 68, and 105.

[50] IBM. Is IMS still strategic for customers and IBM? IMS Version 11 Documentation, 2004. URL `http://publib.boulder.ibm.com/infocenter/dzichelp/v2r2/index.jsp?topic=/com.ibm.imsintro.doc.intro/isimsstrat.htm`. Cited on page 106.

[51] Daniel Jackson. Object models as heap invariants. In Annabelle McIver and Carroll Morgan, editors, *Programming Methodology*. Springer-Verlag, 2002. Cited on pages 14 and 57.

[52] ———. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, Cambridge, Mass., April 2006. ISBN 978-0-262-10114-1. Cited on pages 14, 19, 26, 28, 29, 44, and 103.

[53] Java Bug Parade. 6312706, 2005. URL `http://bugs.sun.com/view_bug.do?bug_id=6312706`. Map entrySet iterators should return different entries on each call to next(). Cited on page 77.

[54] Kevin D. Jones. LM3: a Larch interface language for Modula-3. Technical Report SRC-RR-72, DEC Systems Research Center, June 1991. Version 1.0. Cited on page 14.

[55] ———. A Semantics for a Larch/Modula-3 Interface Language. In Martin and Wing [77], pages 142–157. Cited on pages 4 and 14.

[56] Eric Jul, editor. *Proceedings of the 12thEuropean Conference on Object-Oriented Programming (ECOOP)*, volume 1445 of *Lecture Notes in Computer Science*, Brussels, Belgium, July 1998. Springer-Verlag. ISBN 3-540-64737-6. Cited on pages 115 and 116.

[57] Sarfraz Khurshid. *Generating Structurally Complex Tests from Declarative Constraints*. PhD thesis, MIT Electrical Engineering and Computer Science, 2003. Cited on pages 3, 19, and 33.

[58] Donald E. Knuth. *The Art of Computer Programming v3: Sorting and Searching*. Addison-Wesley, Reading, Mass., 1973. Cited on page 88.

[59] Viktor Kuncak. *Modular Data Structure Verification*. PhD thesis, MIT Electrical Engineering and Computer Science, 2007. Cited on pages 13, 26, 84, and 107.

[60] Gary T. Leavens. *Verifying Object-Oriented Programs that use Subtypes*. PhD thesis, MIT Electrical Engineering and Computer Science, February 1989. Cited on page 83.

[61] ———. Preliminary Design of Larch/C++. In Martin and Wing [77]. Cited on page 14.

[62] ———. Larch/C++ Reference Manual, April 1999. URL `http://www.cs.ucf.edu/~leavens/larchc++manual/lcpp_toc.html`. Cited on pages 13 and 14.

[63] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06u, Iowa State University, April 2003. URL `http://www.jmlspecs.org`. Cited on pages 13, 26, and 84.

[64] Wilhelm Gottfeid Leibniz. *Philosophical Papers and Letters*. D. Reidel, Publisher, 2 edition, 1969. Cited on page 83.

[65] K. R. M. Leino. *Toward reliable modular programs*. PhD thesis, California Institute of Technology, 1995. Cited on page 14.

[66] K. Rustan M. Leino. Data groups: specifying the modification of extended state. In Craig Chambers, editor, *Proceedings of the 13thACM/SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Vancouver, British Columbia, Canada, October 1998. Cited on page 29.

[67] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development*. The MIT Press, Cambridge, Mass., 1986. Cited on pages 66, 72, 82, and 83.

[68] ———. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley, Reading, Mass., 2001. No cited.

[69] Barbara H. Liskov. Data abstraction and hierarchy. In Norman K. Meyrowitz, editor, *Proceedings of the ACM/SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Orlando, FL, October 1987. URL http://doi.acm.org/10.1145/62138.62141. Keynote address. Cited on page 83.

[70] Barbara H. Liskov and Jeanette Wing. A behavioural notion of subtyping. *Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994. Cited on page 83.

[71] Barbara H. Liskov and Jeanette M. Wing. A new definition of the subtype relation. In Oscar Nierstrasz, editor, *Proceedings of the 7thEuropean Conference on Object-Oriented Programming (ECOOP)*, volume 707 of *Lecture Notes in Computer Science*, pages 118–141, Kaiserslautern, Germany, July 1993. Springer-Verlag. ISBN 3-540-57120-5. Cited on page 115.

[72] ———. Corrigenda to ECOOP'93 Paper. *ACM SIGPLAN Notices*, 29(4):4, April 1994. Notes on [71]. Cited on page 15.

[73] J. R. Low. Automatic data structure selection: an example and overview. *Communications of the ACM*, 21(5):376–385, 1978. Cited on page 102.

[74] P. Lucas. Two constructive relations of the block concept and their equivalence. Technical Report 25.085, IBM Laboratory, Vienna, June 1968. Cited on page 15.

[75] Tim Mackinnon, Steve Freeman, and Philip Craig. Endo-testing: Unit testing with mock objects. In *eXtreme Programming and Flexible Processes in Software Engineering (XP2000)*, 2000. Cited on page 103.

[76] Darko Marinov. *Automatic Testing of Software with Structurally Complex Inputs.* PhD thesis, MIT Electrical Engineering and Computer Science, 2004. Cited on page 19.

[77] Ursula Martin and Jeannette Wing, editors. *Proceedings of the 1stWorkshop on Larch*, Dedham, MA, USA, July 1992. Springer-Verlag. Cited on pages 14, 113, and 114.

[78] Bertrand Meyer. *Eiffel: the language.* Prentice-Hall, Inc., 1992. Cited on pages 73, 82, and 88.

[79] Microsoft Research. The F# 1.9.6 Draft Language Specification, September 2008. Cited on page 82.

[80] Leonid Mikhajlov and Emil Sekerinski. A study of the fragile base class problem. In Jul [56], pages 355–382. ISBN 3-540-64737-6. Cited on pages 68 and 69.

[81] Robin Milner. An algebraic definition of simulation between programs. Technical Report CS-TR-71-205, Stanford University, 1971. Cited on page 12.

[82] SASA MISAILOVIC, ALEKSANDAR MILICEVIC, NEMANJA PETROVIC, SARFRAZ KHURSHID, AND DARKO MARINOV. Parallel test generation and execution with Korat. In Crnkovic and Bertolino [27], pages 135–144. Cited on pages 86 and 95.

[83] CARROLL MORGAN. The specification statement. *Transactions on Programming Languages and Systems*, 10(3), 1988. Cited on page 19.

[84] ———. *Programming from Specifications*. Prentice-Hall, Inc., 1st edition, 1990. Cited on page 15.

[85] ———. *Programming from Specifications*. Prentice-Hall, Inc., 2nd edition, 1998. First edition 1990. Cited on pages 15 and 19.

[86] J. MORRIS. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9(3), December 1987. Cited on page 19.

[87] J.M. MORRIS. Laws of data refinement. *Acta Informatica*, 26(4):287–307, February 1989. Cited on page 15.

[88] GREG NELSON, editor. *Systems Programming with Modula-3*. Prentice-Hall, Inc., 1991. Cited on page 110.

[89] TOBIAS NIPKOW, L. C. PAULSON, AND M. WENZEL. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002. Cited on page 26.

[90] JAMES NOBLE, JAN VITEK, AND JOHN POTTER. Flexible alias protection. In Jul [56]. ISBN 3-540-64737-6. Cited on page 107.

[91] MARTIN ODERSKY, LEX SPOON, AND BILL VENNERS. *Programming in Scala*. Artima, November 2008. Cited on pages 20, 66, 68, 82, and 105.

[92] S. OWICKI AND D. GRIES. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6(4):319–340, 1976. Cited on page 15.

[93] CARLOS PACHECO. *Directed Random Testing*. PhD thesis, MIT Electrical Engineering and Computer Science, 2009. Cited on pages 20, 54, 64, 66, 67, 68, 98, and 105.

[94] CARLOS PACHECO, SHUVENDU K. LAHIRI, MICHAEL D. ERNST, AND THOMAS BALL. Feedback-directed random test generation. In WOLFGANG EMMERICH AND GREGG ROTHERMEL, editors, *Proceedings of the 29thACM/IEEE International Conference on Software Engineering (ICSE)*, Minneapolis, MN, 2007. Cited on pages 20, 54, 64, 66, 67, 68, 71, 78, 98, and 105.

[95] MATTHEW M. PAPI, MAHMOOD ALI, TELMO LUIS CORREA JR., JEFF H. PERKINS, AND MICHAEL D. ERNST. Practical pluggable types for Java. In BARBARA G. RYDER AND ANDREAS ZELLER, editors, *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, Seattle, WA, July 2008. ISBN 978-1-60558-050-0. Cited on pages 84, 89, and 102.

[96] TERRENCE PARR. *The Definitive ANTLR Reference*. The Pragmatic Bookshelf, May 2007. ISBN 978-0-9787-3925-6. Cited on page 24.

[97] VIERA K. PROULX AND WESTON JOSSEY. Unit test support for java via reflection and annotations. In *Proceedings of the Principles and Practice of Programming in Java*, Calgary, Alberta, Canada, 2009. Cited on pages 20, 66, 68, and 105.

[98] DEREK RAYSIDE, ZEV BENJAMIN, RISHABH SINGH, JOSEPH P. NEAR, ALEKSANDAR MILICEVIC, AND DANIEL JACKSON. Equality and hashing for (almost) free: Generating implementations from abstraction functions. In JOANNE ATLEE AND PAOLA INVERARDI, editors, *Proceedings of the 31st ACM/IEEE International Conference on Software Engineering (ICSE)*, Vancouver, British Columbia, Canada, 2009. Cited on page 26.

[99] DEREK RAYSIDE, H.-CHRISTIAN ESTLER, AND DANIEL JACKSON. A Guided Improvement Algorithm for Exact, General Purpose, Many-Objective Combinatorial Optimization. Technical Report MIT-CSAIL-TR-2009-033, MIT Computer Science and Artificial Intelligence Laboratory, 2009. URL `http://hdl.handle.net/1721.1/46322`. Cited on page 22.

[100] DEREK RAYSIDE, ALEKSANDAR MILICEVIC, KUAT YESSENOV, GREG DENNIS, AND DANIEL JACKSON. Agile specifications. In YVONNE COADY, editor, *Proceedings of Onward'09*, Orlando, FL, October 2009. Cited on pages 19 and 103.

[101] ROSETTACODE. Tree traversal, Retrieved January 2010. URL `http://rosettacode.org/wiki/Tree_traversal`. Cited on pages 7 and 34.

[102] MARK ROULO. How to avoid traps and correctly override methods from java.lang.Object. *Java World*, 1999. URL `http://www.javaworld.com/javaworld/jw-01-1999/jw-01-object.html`. Cited on page 69.

[103] E. SCHONBERG, J. T. SCHWARTZ, AND M. SHARIR. Automatic data structure selection in SETL. In *Conference Record of the 6th ACM Symposium on the Principles of Programming Languages (POPL)*, pages 197–210, San Antonio, Texas, January 1979. Cited on page 102.

[104] OHAD SHACHAM, MARTIN VECHEV, AND ERAN YAHAV. Chameleon: adaptive selection of collections. In Diwan [33], pages 408–418. Cited on page 102.

[105] SIMON PEYTON JONES. Haskell 98 Language and Libraries, Revised Report, December 2002. URL `http://www.haskell.org/onlinereport/`. Cited on pages 34, 43, 46, 50, and 82.

[106] ARMANDO SOLAR-LEZAMA. *Program Synthesis by Sketching*. PhD thesis, University of California, Berkeley, 2008. Cited on page 107.

[107] MICHAEL SPERBER, R. KENT DYBVIG, MATTHEW FLATT, AND ANTON VAN STRAATEN. Revised^6 Report on the Algorithmic Language Scheme, September 2007. URL `http://r6rs.org`. Cited on pages 73, 82, and 88.

[108] MANDAYAM K. SRIVAS. *Automatic synthesis of implementations for abstract data types from algebraic specifications*. PhD thesis, MIT Electrical Engineering and Computer Science, 1982. Cited on page 19.

[109] YANG MENG TAN. *Formal specification techniques for promoting software modularity, enhancing documentation, and testing specifications*. PhD thesis, MIT Electrical Engineering and Computer Science, 1994. URL `http://hdl.handle.net/1721.1/35391`. Cited on page 14.

[110] EMINA TORLAK. *A Constraint Solver for Software Engineering: Finding Models and Cores of Large Relational Specifications*. PhD thesis, MIT Electrical Engineering and Computer Science, 2008. Cited on pages 22, 24, and 58.

[111] EMINA TORLAK AND DANIEL JACKSON. Kodkod: A relational model finder. In ORNA GRUMBERG AND MICHAEL HUTH, editors, *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 4424 of *Lecture Notes in Computer Science*, pages 632–647, Braga, Portugal, March 2007. Springer-Verlag. Cited on pages 22 and 58.

[112] FRITS VAANDRAGER. Verification of a distributed summation algorithm. In *CONCUR'95*, volume 962 of *Lecture Notes in Computer Science*, pages 190–203, 1995. Invited paper. Cited on page 15.

[113] RAJA VALLÉE-RAI, ÉTIENNE M. GAGNON, LAURIE J. HENDREN, PATRICK LAM, PATRICE POMINVILLE, AND VIJAY SUNDARESAN. Optimizing java bytecode using the soot framework: Is it feasible? In DAVID A. WATT, editor, *Proceedings of the 9th International Conference on Compiler Construction (CC)*, volume 1781 of *Lecture Notes in Computer Science*, pages 18–34, Berlin, March 2000. Springer-Verlag. ISBN 3-540-67263-X. Cited on page 24.

[114] MANDANA VAZIRI, FRANK TIP, STEPHEN FINK, AND JULIAN DOLBY. Declarative object identity using relation types. In ERIK ERNST, editor, *Proceedings of the 21st European Conference on Object-Oriented Programming (ECOOP)*, volume 4609 of *Lecture Notes in Computer Science*, pages 54–78, Berlin, Germany, July 2007. Springer-Verlag. Cited on pages 20, 66, 68, 82, and 105.

[115] WIKIPEDIA. Tree traversal, Retrieved January 2010. URL `http://en.wikipedia.org/wiki/Tree_traversal`. Cited on pages 8, 34, and 57.

[116] JEANETTE M. WING. *A two-tiered approach to specifying programs*. PhD thesis, MIT Electrical Engineering and Computer Science, 1983. Cited on pages 11 and 14.

[117] ———. Writing Larch interface language specifications. *Transactions on Programming Languages and Systems*, 9(1):1–24, January 1987. Cited on page 14.

[118] JIM WOODCOCK AND JIM DAVIES. *Using Z: Specification, Refinement, and Proof.* Prentice-Hall, Inc., 1996. Cited on page 12.

[119] Kuat Yessenov. A light-weight specification language for bounded program verification. Master's thesis, MIT Electrical Engineering and Computer Science, May 2009. Cited on pages 13, 14, 21, 24, 25, 26, 28, 29, 42, 44, 50, 57, 84, and 103.

[120] Pamela Zave. An Insider's Evaluation of PAISLey. *IEEE Transactions on Software Engineering*, 17(3):212–225, March 1991. Cited on pages 19 and 104.

[121] Karen Zee, Viktor Kuncak, and Martin Rinard. Full functional verification of linked data structures. In Amarasinghe [5]. Cited on page 26.

[122] ———. An integrated proof language for imperative programs. In Diwan [33]. Cited on page 26.

[123] Karen K. Zee. *Program Verification [title forthcoming]*. PhD thesis, MIT Electrical Engineering and Computer Science, 2010. Cited on pages 13, 26, 84, and 107.

[124] Karen K. Zee and Viktor Kuncak. Jahob website, November 2009. URL `http://lara.epfl.ch/dokuwiki/data_structure_examples.html`. Retrieved March 2010. Cited on pages 7 and 27.

[125] Yoav Zibin, Alex Potanin, Mahmood Ali, Shay Artzi, Adam KieÅun, and Michael D. Ernst. Object and reference immutability using Java generics. In Crnkovic and Bertolino [27]. Cited on pages 84, 89, and 102.