Modular Data Structure Verification

by

Viktor Kuncak

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

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2007

Author . . . .
Department of Electrical Engineering and Computer Science
February 2007

Certified by . . . .
Martin C. Rinard
Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Modular Data Structure Verification

by

Viktor Kuncak

Submitted to the Department of Electrical Engineering and Computer Science
on February 2007, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

This dissertation describes an approach for automatically verifying data structures, focusing on techniques for automatically proving formulas that arise in such verification. I have implemented this approach with my colleagues in a verification system called Jahob. Jahob verifies properties of Java programs with dynamically allocated data structures.

Developers write Jahob specifications in classical higher-order logic (HOL); Jahob reduces the verification problem to deciding the validity of HOL formulas. I present a new method for proving HOL formulas by combining automated reasoning techniques. My method consists of 1) splitting formulas into individual HOL conjuncts, 2) soundly approximating each HOL conjunct with a formula in a more tractable fragment and 3) proving the resulting approximation using a decision procedure or a theorem prover. I present three concrete logics; for each logic I show how to use it to approximate HOL formulas, and how to decide the validity of formulas in this logic.

First, I present an approximation of HOL based on a translation to first-order logic, which enables the use of existing resolution-based theorem provers. Second, I present an approximation of HOL based on field constraint analysis, a new technique that enables decision procedures for special classes of graphs (such as monadic second-order logic over trees) to be applied to arbitrary graphs.

Third, I present an approximation using Boolean Algebra with Presburger Arithmetic (BAPA), a logic that combines reasoning about sets of elements with reasoning about cardinalities of sets. BAPA can express relationships between sizes of data structures and invariants that correlate data structure size with integer variables. I present the first implementation of a BAPA decision procedure, and establish the exact complexity bounds for BAPA and quantifier-free BAPA.

Together, these techniques enabled Jahob to modularly and automatically verify data structure implementations based on singly and doubly-linked lists, trees with parent pointers, priority queues, and hash tables. In particular, Jahob was able to prove that data structure implementations satisfy their specifications, maintain key data structure invariants expressed in a rich logical notation, and never produce run-time errors such as null dereferences or out of bounds accesses.

Thesis Supervisor: Martin C. Rinard
Title: Professor of Computer Science and Engineering

Acknowledgments

I would like to thank my advisor Martin Rinard for his guidance in all the years of graduate school, for providing me with an environment where I was able to dedicate myself to research as never before, for teaching me to think about the implications of my research, and for his patience to work with me on papers and talks. I thank my thesis committee member Daniel Jackson for numerous discussions on logic and its use in software analysis, for a very enjoyable collaboration experience, and for his excellent questions about my technique for combining reasoning procedures described in Chapter 4. I thank my thesis committee member Professor Arvind for great discussions about programing languages, analysis, and methodology.

I thank my collaborators, who greatly enriched my research and life experience and made the sleepless nights before each deadline so entertaining. I thank Andreas Podelski for discussions about program analysis that started a fruitful collaboration, for his wise advice, and for hosting me at the Max-Planck Institute for Computer Science. I thank Charles Bouillaguet for his work on the translation to first-order logic described in Chapter 5. I thank Thomas Wies for developing a sophisticated symbolic shape analysis whose one part is the field constraint analysis described in Chapter 6, and implementing parts of Jahob infrastructure. I thank Huu Hai Nguyen for implementing a first translation from higher-order logic to BAPA and an interface to the Omega solver, on which the experience of Section 7.7 is based. I thank Karen Zee for her contributions to Hob and Jahob infrastructure, for many proved formulas and many discussions. I thank Patrick Lam for his role in designing and building the Hob system. Without our experience with Hob, I would have never started building Jahob. I thank Bruno Marnette for exciting discussions on polynomial-time logics for sets with cardinality constraints. I thank Rustan Leino for a great internship experience at Microsoft Research and many insightful discussions on program verification methodology. I thank Konstantine Arkoudas for introducing me to his Athena interactive theorem proving environment. I thank Peter Schmitt and Suhabe Bugrara who used Jahob in its early days to specify and verify Java programs.

For useful discussions I also thank: current and previous members of our research group C. Scott Ananian, Chandrasekhar Boyapati, Michael Carbin, Brian Demsky, Zoran Džunić, Maria-Cristina Marinescu, Darko Marinov, Radu Rugina, Alexandru Salcianu, and Amy Williams, my MIT colleagues including Greg Dennis, Michał Karczmarek, Sarfraz Khurshid, Angelina Lee, Karola Meszaros, Sayan Mitra, Derek Rayside, Gregory Shakhnarovich, Mana Taghdiri, Bill Thies, Michael Taylor, Emina Torlak, Mandana Vaziri, David Wentzlaff, and to my colleagues around the world, including Sumit Gulwani, Mayur Naik, Ružica Piskač, Andrey Rybalchenko, and Greta Yorsh.

I thank my parents, Katarina Kunčak and Jaroslav Kunčak, and to my brother, Miro Kunčak, for their continuous support, for numerous emails, and for coming to visit me in Boston this October—it was a wonderful week.

Finally, I thank you, the reader who came across this document. While no part of a dissertation can be as entertaining as the acknowledgments, I hope you find something of interest on the pages that follow.

Viktor Kuncak
, Cambridge, Massachusetts
November 2006

Contents

Chapter 1

Introduction

Does a program written in some programming language behave in a desirable way? Continuously answering such questions in a constructive way is the essence of programming activity. As the size and complexity of programs grows beyond the intellectual capacity of any single individual, it becomes increasingly interesting to apply software tools themselves to help programmers answer such questions. The techniques for building such verification tools are the subject of this dissertation. These questions are immediately relevant to programming practice and are likely to have impact for the future of programming languages and software engineering. They are also a source of interesting problems in mathematical logic, algorithms, and theory of computation.

1.1 Program Verification Today

Decades after its formulation, the verification of correctness properties of software systems remains an open problem. There are many reasons for revisiting the challenge of verifying software properties today.

1. We are witnessing algorithmic advances in specialized techniques such as abstract interpretation and data-flow analysis [35, 75, 226, 225], model checking [24, 40, 136], and advanced type systems [257, 78, 6, 115, 39]. These specialized techniques have proven useful for improving the reliability and performance of software, and there is a clear potential for their use as part of more general verification systems. Moreover, techniques that establish specialized properties have demonstrated that it is profitable to focus the verification effort on partial correctness properties; it is therefore interesting to explore the consequences of applying the same focus to systems that support partial but more heterogeneous properties.

2. Advances in automated theorem proving [248, 244], constraint solving [132, 80, 268], decision procedures [143, 215], and combinations of these techniques [30, 162, 94] enable us to solve new classes of problems in an automated way. The basic idea of this approach is to formulate the verification problem as a set of logical constraints and use solvers for these constraints to check the properties of interest.

3. Increases in hardware resources make the computational power available for verification problems larger, making previously infeasible tasks possible and allowing easier evaluation of expensive techniques.

4. As software becomes ubiquitous, society is less willing to tolerate software errors. The software industry is therefore beginning to use specification and verification as a standard part of their software development process [74].

Several research groups have created program verification infrastructures that exploit some of these recent developments [61, 27, 80, 96, 178], and their creation has led to important steps towards a principled and effective programming methodology for reliable software. Currently, however, these tools are still limited in the scope of properties they can handle in an automated way. The limitations apply in particular to the problem of verifying that complex linked data structures and arrays satisfy given properties in all possible program executions. I have developed a variety of new techniques for automatically proving the validity of constraints that arise in the verification of such complex data structures. To evaluate these techniques, my colleagues and I designed and implemented a concrete verification system that transforms programs and specifications into logical constraints and then proves the validity of the generated constraints. Our results indicate that our system can modularly and automatically verify data structure implementations based on singly and doubly-linked lists, trees with parent pointers, priority queues, and hash tables. In particular, the system was able to prove that data structure implementations satisfy their specifications, maintain key data structure invariants expressed in a rich logical notation, and never produce run-time errors such as null dereferences or out of bounds accesses.

1.2 Verification of Data Structures

Software with dynamically allocated linked data structures is both important and difficult to reason about. Dynamically allocated data structures allow applications to adapt to the sizes of their inputs and to use many efficient algorithms. As a result, such data structures are common in modern software. Unfortunately, this same flexibility makes it easy for programmers to introduce errors, ultimately causing software to crash or produce unexpected results. Automated reasoning about such software is therefore desirable, and many techniques have emerged to tackle this problem, often under the name of "shape analysis" [106, 226, 224, 190, 167, 101] because data structure properties determine the shapes of these data structures in program memory.

One of the main challenges in this area was obtaining a modular approach for automated analysis of data structure properties. Modular analysis appears to be the only long-term solution for designing scalable analyses for data structures, because the precision needed for data structure analyses prevents them from being deployed as whole-program analyses. While type systems and program verification environments have always supported the notion of modular analysis, they have traditionally lacked support for reasoning about precise reachability properties within linked data structures. On the other hand, shape analyses have traditionally lacked good specification formalisms that would allow developers to specify the desired properties and decompose the overall analysis task into smaller subtasks.

Our starting point is several recent approaches for modular analysis of data structure properties. These approaches include role analysis [151], which combines typestate ideas [233] with shape analysis, and the Hob system [152] for the analysis of data structure implementations [253, 254] and data structure uses [164].

Jahob system. Building on these ideas, my collaborators and I have implemented a new verification system, named Jahob. The current focus of Jahob is data structure verification.

However, given the range of properties that it is currently able to verify, I believe that Jahob is also promising as a more general-purpose verification system. The primary contribution of this dissertation is a set of reasoning techniques within Jahob. I also summarize the design of Jahob and present our experience using Jahob to verify sophisticated correctness properties of a variety of data structure implementations.

1.3 The Design of the Jahob Verification System

There are several design decisions involved in creating a usable verification system. I have evaluated one such set of design decisions by building a working Jahob prototype. My primary goal was to create a system that produces sufficiently realistic proof obligations to demonstrate that Jahob's reasoning techniques are practically relevant. I note that Jahob currently does not have a fully developed methodology for object encapsulation. Certain aspects of the Jahob design are therefore likely to change in the future, but most of the aspects I discuss are independent of this issue.

Java subset as the implementation language. My guiding principles in designing Jahob were generality, simplicity and familiarity. Therefore, the Jahob implementation language is a sequential, imperative, memory-safe language defined as a subset of the popular memory-safe language Java [112]. The developer provides Jahob specifications as special comments. This approach enables the developer to compile and test the software using existing Java tools. I found it counterproductive to support the full Java language, so the current Jahob implementation language is a simple subset of Java sufficient for writing imperative and functional data structures.

Isabelle subset as the specification language. Familiarity and generality were also factors in choosing the specification language for Jahob. Specification formulas in Jahob are written in classical higher-order logic—a notation that, in my view, fits well with current informal reasoning in computer science and mathematics. Syntactically, the specification language for formulas is a subset of the language of the Isabelle interactive theorem prover [201]. This design decision allows Jahob to build on the popularity of an existing and freely available interactive theorem proving framework to help developers understand Jahob annotations. It also enables the use of Isabelle to interactively prove formulas that Jahob fails to prove automatically. In particular, this design enables the developers to map parts of proof obligations to program annotations when they perform interactive proofs.

Modular verification. Jahob uses modular reasoning with procedures and modules (classes) as basic units of modularity. This approach allows the developer to separately 1) verify that each procedure satisfies its specification, and 2) verify the rest of the program assuming the specification of each procedure. This separation is key to Jahob's scalability: verifying expressive data structure properties would not be feasible without a mechanism for restricting reasoning to a small program fragment at a time. Following the standard approach [175], [96, Section 4], procedures have preconditions (describing program state at the entry to the procedure), postconditions (correlating initial and final states), and frame conditions (approximating the region of the state that the procedure may modify). Classes can specify representation invariants, which indicate properties that should be true initially and that each procedure (method) should preserve.

Specification variables for abstraction. When verifying a complex piece of software, it is often useful to abstract the behavior of parts of the system using data abstraction [77].

Jahob uses specification variables to support data abstraction. Specification variables are analogous to variables in Java but exist only for the purpose of reasoning. In addition to individual objects and integers, specification variables can also denote sets, functions, and relations. Specification variables often lead to simpler specifications and easier proof obligations.

Modular architecture. The architecture of the current Jahob implementation separates the verification problem into two main subproblems: 1) the generation of logical constraints (proof obligations) in higher-order logic (HOL); and 2) proving the validity of the generated proof obligations. The generation of the proof obligations proceeds through several stages; if needed, a Jahob user can inspect the results of these stages to understand why Jahob generated a particular proof obligation. To prove the validity of the generated proof obligations, Jahob combines multiple automated reasoning procedures. Each reasoning procedure can prove formulas in a specific subset of HOL. To apply a reasoning procedure to a specific proof obligation, Jahob first uses a sound approximation to convert the proof obligation into the procedure's subset of HOL, then uses the procedure to prove the converted proof obligation. The techniques for approximating expressive constraints with more tractable constraints and the techniques for reasoning about these more tractable constraints are the core technical contribution of this dissertation.

1.4 Reasoning about Expressive Constraints in Jahob

Jahob represents proof obligations in a subset of Isabelle's higher-order logic. This approach enables Jahob to pretty print proof obligations and use Isabelle (or other interactive theorem provers) to prove the validity of proof obligations. Unfortunately, the use of interactive theorem provers often requires manually provided proof scripts (specifying, for example, quantifier instantiations, case analysis, and lemmas). Constructing proof scripts can be time consuming and requires the knowledge of proof rules and tactics of the interactive theorem prover. To make the proposed verification approach practical, it is therefore essential to develop techniques that increase the automation of reasoning about higher-order logic formulas.

A simple combination method for expressive constraints. Jahob's approach to automate reasoning about higher-order logic formulas arising in data structure verification is the following. Jahob first splits formulas into an equivalent conjunction of independent smaller formulas. Jahob then attempts to prove each of the resulting conjuncts using a (potentially different) specialized reasoning procedure. Each specialized reasoning procedure in Jahob decides a *subset* of higher-order logic formulas. Such a procedure therefore first approximates a higher-order logic formula using a formula in the subset, and then proves the resulting formula using a specialized algorithm. The core contributions of this dissertation are three specialized reasoning techniques: translation to first-order logic, field constraint analysis with monadic second-order logic of trees, and Boolean Algebra with Presburger Arithmetic.

First-order logic approximation. Translation to first-order logic enables the use of first-order theorem provers to prove certain higher-order logic formulas. This technique is promising because 1) proof obligations for correct programs often have short proofs and 2) decades of research in resolution-based automated theorem proving have produced implementations that can effectively search the space of proofs. A first-order theorem prover can act as a decision procedure for certain classes of problems, while being applicable even outside these fragments. Note that it is possible to axiomatize higher-order logic and lambda calculus by translating it into combinatory logic [71]. Such approaches enable theorem provers to prove a wide range of mathematical formulas [126, 188, 187]. However, such translations also potentially obscure the original formulas. Instead, we use a translation that attempts to preserve the structure of the original formula, typically translating function applications into predicates applied to arguments (instead of encoding function application as an operator in logic). As a result, some constructs are simply not translatable and our translation conservatively approximates them. Nevertheless, this approach enabled Jahob to prove many formulas arising in data structure verification by leveraging theorem provers such as SPASS [248] and E [228]. We have successfully used this technique to verify properties of recursive linked data structures and array-based data structures.

Field constraint analysis. We have developed field constraint analysis, a new technique that enables decision procedures for data structures of a restricted shape to be applied to a broader class of data structures. Jahob uses field constraint analysis to extend the range of applicability of monadic-second order logic over trees. Namely, a direct application of monadic second-order logic over trees would only enable the analysis of simple tree and singly-linked list data structures. The use of field constraint analysis on top of monadic second-order logic enables Jahob to verify a larger set of data structures, including doubly-linked lists, trees with parent pointers, two-level skip lists, and linked combinations of multiple data structures. We have established the completeness of this technique for an interesting class of formulas, which makes it possible to predict when the technique is applicable. The overall observation of this approach is that it is often possible to further extend expressive decidable logics (such as monadic second-order logic over trees) to make them more useful in practice.

Boolean Algebra with Presburger Arithmetic. Boolean Algebra with Presburger Arithmetic (BAPA) is a logic that combines reasoning about sets of elements with reasoning about cardinalities of sets. BAPA supports arbitrary integer and set quantifiers. A specialized reasoning technique based on BAPA makes it possible to reason about the relationships between sizes of data structures as well as to reason about invariants that correlate data structure size with integer variables. I present the first implementation and the first and optimal complexity bounds for deciding BAPA. I also characterize the computational complexity of quantifier-free BAPA. These results show that data structure verification can lead to challenging and easy-to-describe decision problems that can be addressed by designing new decision procedures; these decision procedures then may also be of independent interest.

1.5 Summary of Contributions

In summary, this dissertation makes the following contributions.

- An approach for reasoning about expressive logical constraints by splitting formulas into conjuncts and approximating each conjunct using a different specialized reasoning procedure;

- A specialized reasoning procedure based on a translation to first-order logic that enables the use of resolution-based theorem provers;

- A specialized reasoning procedure based on field constraint analysis, which extends the applicability of existing decision procedures such as monadic second-order logic over trees;
- A specialized reasoning procedure based on Boolean Algebra with Presburger Arithmetic, which enables reasoning about sizes of data structures.

My colleagues and I have implemented these techniques in the context of the Jahob verification system. This dissertation outlines the current design of Jahob. Jahob accepts programs in a subset of Java with annotations written in an expressive subset of Isabelle. Jahob supports modular reasoning and data abstraction, enabling focused application of potentially expensive reasoning techniques. We used Jahob to verify several data structure implementations, including, for the first-time, automated verification that data structures such as hash tables correctly implement finite maps and satisfy their internal representation invariants. I present examples of these data structures and their specifications throughout the dissertation.

These results suggest that the routine verification of data structure implementations is realistic and feasible. We will soon have repositories of verified data structures that provide most of the functionality required in practice to effectively represent and manipulate information in computer programs. The focus will then shift from the verification of data structure implementations to the verification of application-specific properties across the entire program.

The reasoning techniques described in this dissertation are likely to be directly relevant for such high-level verification because they address the verification of certain fundamental properties of sets and relations that arise not only within data structures, but also at the application level. For example, the constraint "a set of objects and a relation form a tree" arises not only in container implementations (where the tree structure ensures efficient access to data but is not externally observable) but also when modelling application data, as in a family tree modelling ancestors in a set of individuals. Note that in the latter case, the ancestor relationship is externally observable, so the tree property captures a constraint visible to the users of the system. Such externally observable properties are likely to be of interest regardless of software implementation details.

There is another way in which the techniques that this dissertation presents are important for high-level verification. Because they address the verification of data structure implementations, these techniques allow other application-specific verification procedures to ignore data structure implementation details. The ability to ignore these internal details simplifies high-level verification, leading to a wider range of automatically verifiable properties.

In the near future, I expect developers to adopt tools that automatically enforce increasingly sophisticated correctness properties of software systems. By enforcing the key correctness properties, these tools narrow down the subspace of acceptable programs, effectively decreasing the cognitive burden of developing software. Left with fewer choices, each of which is more likely to be correct, developers will be able to focus less on formalizing their intuition in terms of programming language constructs and more on the creative and higher-level aspects of producing useful and reliable software systems.

Chapter 2

An Example of Data Structure Verification in Jahob

This chapter introduces the main ideas of data structure verification in Jahob using a verified container implementation as an example. Although simple, this example illustrates how Jahob reduces the verification problem to the validity of logical formulas, and how it combines multiple reasoning procedures to prove the resulting formulas. I illustrate the notation in the example along the way, deferring more precise descriptions to later sections: the details of the Jahob specification language are in Section 3.2, whereas the details of Jahob's formula notation are in Section 4.1 (Figure 4-3).

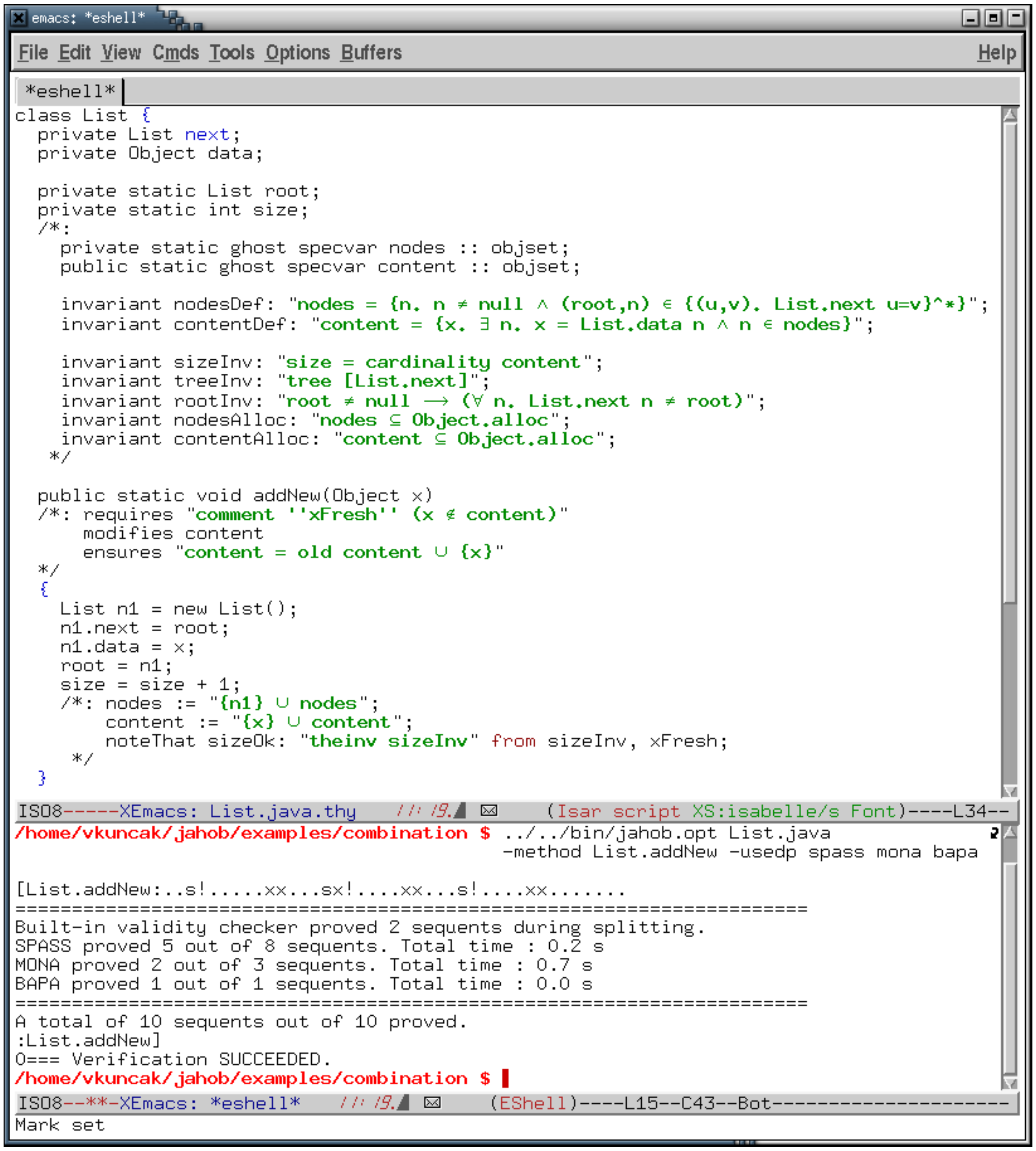
2.1 A Jahob Session

Consider the verification of a global singly-linked list in Jahob. Figure 2-1 shows an example verification session. The editor window consists of two sections. The upper section shows the `List` class with the `addNew` method. The lower section shows the command used to invoke Jahob to verify `addNew`, as well as Jahob's output indicating a successful verification. A Jahob invocation specifies the name of the source file `List.java`, the method `addNew` to be verified, and a list of three *provers*: `spass`, `mona`, and `bapa`, used to establish proof obligations during verification. Jahob's output indicates a successful verification and shows that all three provers took part in the verification.

2.2 Specifying Java Programs in Jahob

Figure 2-3 shows the `List` class whose verification is shown in Figure 2-1. Figure 2-3 uses the ASCII character sequences that produce the mathematical symbols shown in Figure 2-1. Just like in Isabelle/HOL [201], a more concise ASCII notation is also available for symbols in Jahob formulas. For example, the the ASCII sequence `-->` can replace the keyword \⟨longrightarrow⟩.

As Figures 2-1 and 2-3 illustrate, users write Jahob specifications as special comments that start with the colon sign ":". Therefore, it is possible to compile and run Jahob programs using the existing Java compilers and virtual machines. Jahob specifications use the standard concepts of preconditions, postconditions, invariants, and specification variables [96, Section 4]. The specification of the `addNew` method is in the comment that follows the header of `addNew`. It contains a `requires` clause (precondition), a `modifies`



```
class List {
  private List next;
  private Object data;

  private static List root;
  private static int size;
  /*:
    private static ghost specvar nodes :: objset;
    public static ghost specvar content :: objset;

    invariant nodesDef: "nodes = {n. n ≠ null ∧ (root,n) ∈ {(u,v). List.next u=v}^*}";
    invariant contentDef: "content = {x. ∃ n. x = List.data n ∧ n ∈ nodes}";

    invariant sizeInv: "size = cardinality content";
    invariant treeInv: "tree [List.next]";
    invariant rootInv: "root ≠ null ⟶ (∀ n. List.next n ≠ root)";
    invariant nodesAlloc: "nodes ⊆ Object.alloc";
    invariant contentAlloc: "content ⊆ Object.alloc";
   */

  public static void addNew(Object x)
  /*: requires "comment ''xFresh'' (x ∉ content)"
      modifies content
      ensures "content = old content ∪ {x}"
  */
  {
    List n1 = new List();
    n1.next = root;
    n1.data = x;
    root = n1;
    size = size + 1;
    /*: nodes := "{n1} ∪ nodes";
        content := "{x} ∪ content";
        noteThat sizeOk: "theinv sizeInv" from sizeInv, xFresh;
      */
  }
ISO8-----XEmacs: List.java.thy    //: 19.      (Isar script XS:isabelle/s Font)----L34--
/home/vkuncak/jahob/examples/combination $ ../../bin/jahob.opt List.java
                                           -method List.addNew -usedp spass mona bapa

[List.addNew:..s!.....xx...sx!....xx...s!....xx.......
======================================================================
Built-in validity checker proved 2 sequents during splitting.
SPASS proved 5 out of 8 sequents. Total time : 0.2 s
MONA proved 2 out of 3 sequents. Total time : 0.7 s
BAPA proved 1 out of 1 sequents. Total time : 0.0 s
======================================================================
A total of 10 sequents out of 10 proved.
:List.addNew]
0=== Verification SUCCEEDED.
/home/vkuncak/jahob/examples/combination $
ISO8--**-XEmacs: *eshell*    //: 19.      (EShell)----L15--C43--Bot----------------------
Mark set
```

Figure 2-1: Screen shot of verifying a linked list implementation

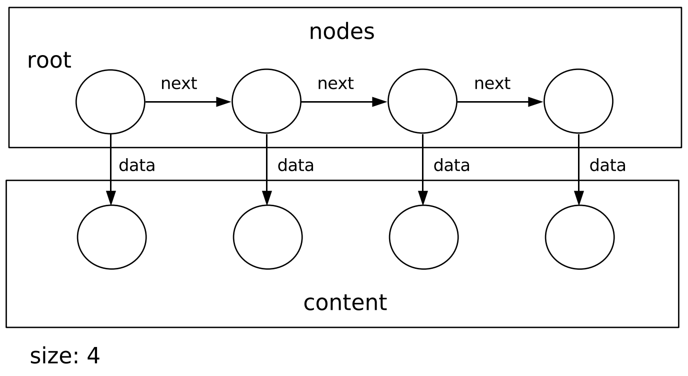


Figure 2-2: Example list data structure corresponding to Figure 2-3

clause (frame condition), and an `ensures` clause (postcondition). These clauses are the public interface of `addNew`. Public interfaces enable the clients of `List` to reason about `List` operations without having access to the details of the `List` implementation. Such data abstraction is possible because `addNew` specification is stated only in terms of the public `content` specification variable, without exposing variables `next`, `data`, `root`, and `size`. The `addNew` specification therefore hides the fact that the container is implemented as an acyclic singly-linked list, as opposed to, for example, a doubly-linked list, circular list, tree, or an array. Researchers have identified the importance of data abstraction in the context of manual reasoning about programs [77, 189, 123]; Jahob takes advantage of this approach to improve the scalability of verification. In this particular example, the postcondition of `addNew` allows clients to conclude (among other facts) that the argument `x` of a call to `addNew` becomes part of `content` after the procedure finishes the execution. In general, data structure interfaces that use sets and relations as specification variables allow clients to abstractly reason about the membership of objects in data structures.

2.3 Details of a Container Implementation and Specification

We next examine in more detail the `List` class from Figure 2-3. Figure 2-2 illustrates the values of variables in the `List` class when the list stores four elements.

Concrete state. The `List` class implements a global list, which has only one instance per program. The static `root` reference points to an acyclic singly-linked list of nodes of the `List` type. The `List` nodes are linked using the `next` field; the useful external data is stored in the list using the `data` field. The `size` field maintains the number of elements of the data structure.

Specification variables. Specification variables are variables that do not affect program execution and exist only for the purpose of reasoning about the program. The `List` class contains two ghost specification variables. A ghost specification variable changes only in response to explicit specification assignments and is independent of other variables (see Section 3.2.2). As Figure 2-2 shows, the `nodes` specification variable denotes the set of all auxiliary `List` objects reachable from list root, whereas `content` stores the actual external objects pointed to by `data` fields of `nodes` objects. Note that `content` is a public variable used in contracts of public methods, whereas `nodes` is an auxiliary private variable that helps the developer express the class invariants and helps Jahob prove them.

```
class List {
  private List next;
  private Object data;
  private static List root;
  private static int size;
  /*:
    private static ghost specvar nodes :: objset = "{}";
    public static ghost specvar content :: objset = "{}";

    invariant nodesDef: "nodes = {n. n \<noteq> null \<and> (root,n) \<in> {(u,v). List.next u=v}^*}";
    invariant contentDef: "content = {x. \<exists> n. x = List.data n \<and> n \<in> nodes}";

    invariant sizeInv: "size = cardinality content";
    invariant treeInv: "tree [List.next]";
    invariant rootInv: "root \<noteq> null \<longrightarrow> (\<forall> n. List.next n \<noteq> root)";
    invariant nodesAlloc: "nodes \<subseteq> Object.alloc";
    invariant contentAlloc: "content \<subseteq> Object.alloc";  */

  public static void addNew(Object x)
  /*: requires "comment ''xFresh'' (x \<notin> content)"
      modifies content
      ensures "content = old content \<union> {x}" */
  {
    List n1 = new List();
    n1.next = root;
    n1.data = x;
    root = n1;
    size = size + 1;
    /*: nodes := "{n1} \<union> nodes";
        content := "{x} \<union> content";
        noteThat sizeOk: "theinv sizeInv" from sizeInv, xFresh;
     */
  }

  public static boolean member(Object x)
  //: ensures "result = (x \<in> content)"
  {
      List current = root;
      //: ghost specvar seen :: objset = "{}"
      while /*: inv "(current = null \<or> current \<in> nodes)
                \<and> seen = {n. n \<in> nodes \<and> (current,n) \<notin> {(u,v). List.next u=v}^*}
                \<and> (\<forall> n. n \<in> seen --> List.data n \<noteq> x)" */
            (current != null) {
        if (current.data==x) {
          return true;
        }
        //: seen := "seen \<union> {current}"
        current = current.next;
      }
      //: noteThat seenAll: "seen = nodes";
      return false;
  }

  public static int getSize()
  //: ensures "result = cardinality content"
  {  return size; }
}
```

Figure 2-3: Linked list implementation

Class invariants. Class invariants denote properties of the private state of the class that are true in all reachable program states. Following the standard approach [247], Jahob proves that class invariants hold in reachable states by proving that they hold in the initial state, and conjoining them to preconditions and postconditions of public methods such as `addNew`. Invariants in Jahob are expressed using a subset of the notation of Isabelle/HOL [201]. The `List` class contains seven invariants.

The *nodesDef* invariant characterizes the value of the `nodes` specification variable as the set of all objects `n` that are not null and are reachable from the `root` variable along the `next` field. Note that the notation $\{x.\ P(x)\}$ in Isabelle/HOL denotes the set of all elements x that satisfy property $P(x)$. Reachability along the `next` field is denoted by the membership of the pair `(root,n)` in the transitive closure of the binary relation corresponding to the function `List.next`. The function `List.next` maps each object `x` to the object referenced by its `next` field. In general, Jahob models fields as total functions from all objects to objects; if the field is inapplicable to a given object, the function is assumed to have the value `null`. In Jahob's semantic model an object essentially corresponds to a Java reference, it is simply an identifier because its content is given entirely by the values of the functions modelling the fields.

The *contentDef* invariant characterizes the value of the `content` specification variable as the image of the `nodes` set under the `List.data` function corresponding to the `data` field. The *sizeInv* invariant uses the `cardinality` operator to state that the `size` field is equal to the number of objects in the `content` set. The *treeInv* invariant uses the `tree` shorthand to state that the structure given by the `List.next` fields of objects is a union of directed trees. In other words, there are no cycles of `next` fields and no two distinct objects have the `next` field pointing to the same object. The *rootInv* invariant states that no `next` field points to the first node of the list by requiring that, if `root` is not `null`, then the `next` field of no other object points to it. The last two invariants, *nodesAlloc* and *contentAlloc*, simply state that the sets `nodes` and `content` contain only allocated objects.

The `addNew` method. The `addNew` method expects a fresh element that is not already in the list. The developer indicates this requirement using the `requires` clause $x \notin \text{content}$. The construct comment "xFresh" (...) labels the precondition formula with the identifier xFresh, so that it can be referred to later. The `modifies` clause indicates that `addNew` may modify the `content` specification variable. If this clause was absent (as is the case for the `member` method), the method would not be allowed to modify the public `content` variable. Note that private variables (such as `root` and `next`) need not be declared in the `modifies` clause of a public method. The `ensures` clause of `addNew` indicates the relationship between the values of `content` at procedure beginning and procedure end. The value of `content` at the end of the procedure is denoted simply `content`, whereas the initial value of `content` at procedure entry is denoted `old content`. The operator $\backslash\langle\text{union}\rangle$ denotes set union, so the postcondition indicates that the object `x` is inserted into the list, that all existing elements are preserved, and that no unwanted elements are inserted. Therefore, the contract of `addNew` gives a complete characterization of the behavior of the method under the set view of the list.

The body of the `addNew` method consists of two parts. The first part is a sequence of the usual Java statements that create a fresh `List` node `n1`, insert `n1` in the front of the list, store the given object `x` into `n1`, and increment the size variable. The second part of the body is a special comment containing three specification statements. The first two statements are specification assignments, which assign the value denoted by the right-hand side formula to the variable on the left hand side. The first assignment updates the `nodes` specification variable to reflect the newly inserted element `n1`, and the second assignment updates the `content` variable to reflect the insertion of `x`. The final specification statement is a `noteThat` statement, which establishes a lemma about the current program state to help Jahob prove the preservation of the *sizeInv* invariant. In general, a `noteThat` statement has an optional label, a formula, and an optional justification indicating the labels of assumptions from which the formula should follow. In this case, the formula itself is an invariant labelled `sizeInv`; the shorthand `theinv L` expands into the class invariant given by the label `L`. The `from` keyword of the `noteThat` statement in `addNew` indicates that the preservation of the *sizeInv* invariant follows from the fact that *sizeInv* was true in the initial state of the procedure, and from the procedure precondition labelled `xFresh`. When the `from` keyword is omitted, the system attempts to use all available assumptions, which can become unfeasible when the number of invariants is large. Note that omitting the precondition ($x \notin \text{content}$) from the contract of `addNew` would cause the verification to fail because the method would increment `size` even when `x` is already in the data structure, violating the `sizeInv` invariant.

The `member` and `getSize` methods. The `member` method tests whether a given object is stored in the list, whereas the `getSize` method returns the size of the data structure. The `member` and `getSize` methods do not modify the data structure, but only traverse it and return the appropriate values. The fact that these methods do not modify the `content` variable is reflected in the absence of a `modifies` clause from the contracts of these methods. These methods also happen to have no precondition, so their precondition is implicitly assumed to be the formula `True`.

Because `member` and `getSize` return values depending on the data structure content, these methods illustrate why the approach based on ghost variables is sound. Namely, to prove the postconditions of `member` and `getSize` methods, it is necessary to assume the class invariants. As the simplest example, proving the postcondition of `getSize` relies on *sizeInv*; in the absence of *sizeInv*, the `size` field could be an arbitrary integer field unrelated to the data structure content. Similarly, in the absence of *contentDef* it would be impossible to prove any relationship between the result of the `member` method and the value of `content`. Once the user introduces these class invariants, the `addNew` method must preserve them. This means that `addNew` must maintain `content` according to *contentDef* when `nodes` or `data` change, and maintain `nodes` according to *nodesDef* when `root` or `next` change.

The implementation of the `member` method also illustrates the use of loop invariants and local ghost variables. In general, I assume that the developer has supplied loop invariants for all loops of the methods being analyzed. (Jahob does have techniques for loop invariant inference [254], but they are outside the scope of this dissertation.) A loop invariant is stated before the condition of the `while` loop. The purpose of the local ghost variable `seen` is to denote the nodes that have been already traversed in the loop. This purpose is encoded in the second conjunct of the loop invariant. In addition to the definition of `seen`, the loop invariant for the `member` method states that the local variable `current` belongs to the nodes of the list (and is therefore reachable from the root), unless it is null. The last conjunct of the loop invariant states that the parameter object `x` is not in the portion of the list traversed so far. The key observation, stated by the `noteThat` statement after the loop, is that, when the loop terminates, `seen = nodes`, that is, `seen` contains all nodes in the list. From this observation and the final conjunct of the loop invariant Jahob proves that returning `false` after the end of the loop implies that the element is not in the list.

Proving that the result is correct when the element is found follows from the class invariants and the first conjunct of the loop invariant.

2.4 Generating Verification Conditions in Jahob

I next illustrate how Jahob generates a formula stating that a method to be verified 1) conforms to its explicitly supplied contract, 2) preserves the class invariants, and 3) never produces run-time errors such as a null dereference.

Translation to guarded command language. Like ESC/Java [96], Jahob first transforms the annotated method into a guarded command language. Figure 2-4 shows the sequence of guarded commands resulting from the translation of the `addNew` procedure. Note that preconditions and class invariants become part of the `assume` statement at procedure entry (Lines 3–10). Similarly, the postcondition and the class invariants become part of an `assert` statement at the end of the procedure (Lines 35-42). The `assume` statements in Lines 11–13 encode the fact that parameters and local variables 1) have the correct type (for example, n1 \⟨in⟩ List), and 2) point to allocated objects.

Jahob models allocated objects using the variable `Object.alloc`, which stores currently allocated objects. (Figure 2-4 denotes this variable as `Object_alloc`, replacing the occurrences of "`.`" in all identifiers with "`_`" for compatibility with Isabelle/HOL.). Jahob assumes that allocated objects include `null`, but not the objects that will be allocated in the future. Lines 15–23 are the result of translation of the statement `n1 = new List()` of the `addNew` procedure. The `havoc` statement non-deterministically changes the value of the temporary variable `tmp_1`; the subsequent `assume` statement assumes that the object is not referenced by any other object and that its fields are null. Together, these two statements have the effect of picking a fresh unallocated object. The assignment statement in Line 22 then extends the set of currently allocated objects with the fresh object, and Line 23 assigns the fresh object to the variable `n1`.

Lines 24–25 are the translation of the field assignment `n1.next=root`. Line 24 is an assertion that checks that `n1` is not null, whereas Line 25 encodes the change to the `next` field using the function update operator that changes the function at a given argument. The translation of `n1.data=x` is analogous. Finally, Jahob translates the `noteThat` statement into an `assert` statement followed by an `assume` statement with the identical formula.

Verification condition generation. Figure 2-5 shows the verification condition corresponding to the guarded commands in Figure 2-4. Jahob computes the verification condition as the weakest precondition of the guarded command translation with respect to the predicate `True`. The computation uses standard rules for weakest precondition computation where `assume` becomes an assumption in an implication, `assert` becomes a conjunction, assignment becomes substitution, and `havoc` introduces a fresh variable.

The resulting verification condition therefore mimics the guarded commands themselves, but is expressed entirely in terms of the program state at procedure entry. When comparing Figure 2-5 to Figure 2-4, note that Jahob replaces the transitive closure of a binary relation with the transitive closure of a binary predicate, denoted by the rtrancl_pt operator. Furthermore, to allow references to old versions of variables, Jahob replaces each identifier `old_id` with the identifier `id` in the final verification condition. The result is identical to saving the values of all variables using a sequence of assignments of the form `old_id := id` at the beginning of the translated procedure.

```
public proc List.addNew(x : obj) : unit
{
  assume ProcedurePrecondition: "comment ''xFresh'' (x \<notin> List_content)
    \<and> comment ''List_PrivateInvnodesDef'' (List_nodes = ...)
    \<and> comment ''List_PrivateInvcontentDef'' (List_content = ...)
    \<and> comment ''List_PrivateInvsizeInv'' ( List_size = cardinality List_content)
    \<and> comment ''List_PrivateInvtreeInv'' (tree [List_next])
    \<and> comment ''List_PrivateInvrootInv'' ((List_root \<noteq> null) --> ...)
    \<and> comment ''List_PrivateInvnodesAlloc'' (List_nodes \<subseteq> Object_alloc)
    \<and> comment ''List_PrivateInvcontentAlloc'' (List_content \<subseteq> Object_alloc)";
  assume x_type: "(x \<in> Object) \<and> (x \<in> Object_alloc)";
  assume tmp_1_type: "(tmp_1 \<in> List) \<and> (tmp_1 \<in> Object_alloc)";
  assume n1_type: "(n1 \<in> List) \<and> (n1 \<in> Object_alloc)";
  havoc tmp_1;
  assume AllocatedObject: "(tmp_1 \<noteq> null)
              \<and> (tmp_1 \<notin> Object_alloc)
              \<and> (tmp_1 \<in> List)
              \<and> (\<forall> y. ((List_next y) \<noteq> tmp_1))
              \<and> (\<forall> y. ((List_data y) \<noteq> tmp_1))
              \<and> (List_next tmp_1 = null)
              \<and> (List_data tmp_1 = null)";
  Object_alloc := "Object_alloc \<union> {tmp_1}";
  n1 := "tmp_1";
  assert ObjNullCheck: "n1 \<noteq> null";
  List_next := "List_next(n1 := List_root)";
  assert ObjNullCheck: "n1 \<noteq> null";
  List_data := "List_data(n1 := x)";
  List_root := "n1";
  tmp_2 := "List_size + 1";
  List_size := "tmp_2";
  List_nodes := "{n1} \<union> List_nodes";
  List_content := "{x} \<union> List_content";
  assert sizeOk: "comment ''sizeInv'' ( List_size = cardinality List_content)" from sizeInv, xFresh;
  assume sizeOk: "comment ''sizeInv'' (List_size = cardinality List_content)";
  assert ProcedureEndPostcondition: "List_content = (old_List_content \<union> {x})
   \<and> comment ''List_PrivateInvnodesDef'' (List_nodes = ...)
   \<and> comment ''List_PrivateInvcontentDef'' (List_content = ...)
   \<and> comment ''List_PrivateInvsizeInv'' ( List_size = cardinality List_content)
   \<and> comment ''List_PrivateInvtreeInv'' (tree [List_next])
   \<and> comment ''List_PrivateInvrootInv'' ((List_root \<noteq> null) --> ...)
   \<and> comment ''List_PrivateInvnodesAlloc'' (List_nodes \<subseteq> Object_alloc)
   \<and> comment ''List_PrivateInvcontentAlloc'' (List_content \<subseteq> Object_alloc)";
}
```

Figure 2-4: Guarded command version of `addNew` from Figure 2-3

Note that the resulting verification condition includes quantifiers, reachability properties, as well as cardinality constraints on sets. The main contribution of this dissertation are techniques for proving the validity of such formulas.

2.5 Proving Formulas using Multiple Reasoning Procedures

Jahob's approach to proving the validity of complex verification conditions is to split the verification condition into multiple conjuncts and prove each conjunct using a potentially different automated reasoning procedure. I illustrate this approach on the verification condition from Figure 2-5.

Splitting into conjuncts. Splitting transforms a formula into multiple conjuncts using a set of simple equivalence-preserving rules. One such rule transforms $A \rightarrow (B \land C)$ into the conjunction of $A \rightarrow B$ and $A \rightarrow C$. Jahob uses such rules to split the verification condition of Figure 2-5 into 10 different conjuncts. It is possible to identify these conjuncts from the guarded commands of Figure 2-4 already:

- The two "ObjNullCheck" `assert` statements generate identical subformulas in the verification condition, which Jahob detects during the construction of the verification condition and generates only one subformula in Figure 2-5. This subformula leads to one conjunct during splitting;
- The `sizeOk` assertion generates the second conjunct;
- The remaining 8 conjuncts are in the `assert` statement corresponding to the end of the procedure: one is for the explicit postcondition of the procedure, and one is for each of the 7 class invariants.

Each of the conjuncts generated by splitting has the form $A_1 \land \ldots A_n \rightarrow B$ where $A_1, \ldots, A_n$ are assumptions and B is the goal of the conjunct. I call the generated conjuncts "sequents" by analogy with the sequent calculus expressions $A_1, \ldots, A_n \vdash B$ (but I do not claim any deeper connections with the sequent calculus).

Proving individual conjuncts. As Figure 2-1 indicates, Jahob takes advantage of four techniques to prove the 10 sequents generated by splitting.

1. Jahob's built-in validity checker uses simple syntactic matching to prove 2 sequents: the absence of null dereference and the fact that the `noteThat` statement implies that *sizeInv* invariant holds in the postcondition. In both of these sequents, the goal occurs as one of the assumptions, so a simple syntactic check is sufficient. Figure 2-6 shows the sequent corresponding to the check for absence of null dereference.

2. Jahob uses an approximation using monadic second-order logic (Chapter 6) and the MONA decision procedure [143] to prove two sequents that require reasoning about transitive closure: the preservation of the *nodesDef* invariant and the preservation of the *treeInv* invariant. Figure 2-7 shows the sequent for the preservation of *nodesDef* invariant.

3. Jahob uses a translation into Boolean Algebra with Presburger Arithmetic (Chapter 7) to prove the `noteThat` statement `sizeOk` from the precondition *xFresh* and the fact that *sizeInv* holds in the initial state. Figure 2-8 shows the corresponding sequent.

$$
\begin{array}{l}
\mathsf{List_root} \in \mathsf{List} \wedge \\
\mathsf{comment}\text{“Precondition”} \\
\quad (\ \mathsf{comment}\text{“xFresh”}\ (x \notin \mathsf{List_content}) \wedge \\
\qquad \mathsf{comment}\text{“List_PrivateInvnodesDef”} \\
\qquad\qquad \mathsf{List_nodes} = \{n.\, n \neq \mathsf{null} \wedge (\mathsf{rtrancl_pt}(\lambda uv.\mathsf{List_next}\, u = v)\, \mathsf{List_root}\, n)\} \wedge \\
\qquad \mathsf{comment}\text{“List_PrivateInvcontentDef”} \\
\qquad\qquad \mathsf{List_content} = \{x.\, \exists n.\, x = \mathsf{List_data}\, n \ \wedge \ n \in \mathsf{List_nodes}\} \wedge \\
\qquad \mathsf{comment}\text{“List_PrivateInvsizeInv”}(\mathsf{List_size} = \mathsf{cardinality}\ \mathsf{List_content}) \wedge \\
\qquad \mathsf{comment}\text{“List_PrivateInvtreeInv”}(\mathsf{tree}\ [\mathsf{List_next}]) \wedge \\
\qquad \mathsf{comment}\text{“List_PrivateInvrootInv”} \\
\qquad\qquad (\mathsf{List_root} \neq \mathsf{null} \ \rightarrow \ (\forall n.\ \mathsf{List_next}\, n \neq \mathsf{List_root})) \wedge \\
\qquad \mathsf{comment}\text{“List_PrivateInvnodesAlloc”}(\mathsf{List_nodes} \subseteq \mathsf{Object_alloc}) \wedge \\
\qquad \mathsf{comment}\text{“List_PrivateInvcontentAlloc”}(\mathsf{List_content} \subseteq \mathsf{Object_alloc})) \wedge \\
\quad \mathsf{comment}\text{“x_type”}(x \in \mathsf{Object} \wedge x \in \mathsf{Object_alloc}) \wedge \\
\quad \mathsf{comment}\text{“AllocatedObject”}(\mathsf{tmp_1_10} \neq \mathsf{null} \wedge \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{tmp_1_10} \notin \mathsf{Object_alloc} \wedge \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{tmp_1_10} \in \mathsf{List} \wedge \\
\qquad\qquad\qquad\qquad\qquad\qquad (\forall y.\mathsf{List_next}\, y \neq \mathsf{tmp_1_10}) \wedge \\
\qquad\qquad\qquad\qquad\qquad\qquad (\forall y.\mathsf{List_data}\, y \neq \mathsf{tmp_1_10}) \wedge \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{List_next}\ \mathsf{tmp_1_10} = \mathsf{null} \wedge \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{List_data}\ \mathsf{tmp_1_10} = \mathsf{null}) \\
\longrightarrow \\
(\ \mathsf{comment}\text{“ObjNullCheck”}(\mathsf{tmp_1_10} \neq \mathsf{null}) \wedge \\
\quad \mathsf{comment}\text{“sizeOk FROM:sizeInv,xFresh”}(\mathsf{comment}\text{“sizeInv”} \\
\qquad\qquad \mathsf{List_size} + 1 = \mathsf{cardinality}\ (\{x\} \cup \mathsf{List_content})) \wedge \\
\quad (\mathsf{comment}\text{“sizeOk”}(\mathsf{comment}\text{“sizeInv”}(\mathsf{List_size} + 1 = \mathsf{cardinality}\ (\{x\} \cup \mathsf{List_content})) \\
\quad \longrightarrow \\
\quad \mathsf{comment}\text{“ProcedureEndPostcondition”} \\
\quad (\ \{x\} \cup \mathsf{List_content} = \mathsf{List_content} \cup \{x\} \wedge \\
\qquad \mathsf{comment}\text{“List_PrivateInvnodesDef”}(\{\mathsf{tmp_1_10}\} \cup \mathsf{List_nodes} = \{n.n \neq \mathsf{null} \wedge \\
\qquad\qquad (\mathsf{rtrancl_pt}(\lambda uv.((\mathsf{List_next}(\mathsf{tmp_1_10} := \mathsf{List_root}))u = v))\ \mathsf{tmp_1_10}\ n)\}) \wedge \\
\qquad \mathsf{comment}\text{“List_PrivateInvcontentDef”}(\{x\} \cup \mathsf{List_content} \ = \\
\qquad\qquad \{x__6.\, \exists n.\, x__6 = (\mathsf{List_data}(\mathsf{tmp_1_10} := x))\, n \ \wedge n \in (\{\mathsf{tmp_1_10}\} \cup \mathsf{List_nodes})\}) \wedge \\
\qquad \mathsf{comment}\text{“List_PrivateInvsizeInv”}(\mathsf{List_size} + 1 = \mathsf{cardinality}\,(\{x\} \cup \mathsf{List_content})) \wedge \\
\qquad \mathsf{comment}\text{“List_PrivateInvtreeInv”}\ (\mathsf{tree}\ [(\mathsf{List_next}(\mathsf{tmp_1_10} := \mathsf{List_root}))]) \wedge \\
\qquad \mathsf{comment}\text{“List_PrivateInvrootInv”} \\
\qquad\qquad \mathsf{tmp_1_10} \neq \mathsf{null} \rightarrow (\forall n.(\mathsf{List_next}(\mathsf{tmp_1_10} := \mathsf{List_root}))\, n \neq \mathsf{tmp_1_10}) \wedge \\
\qquad \mathsf{comment}\text{“List_PrivateInvnodesAlloc”} \\
\qquad\qquad (\{\mathsf{tmp_1_10}\} \cup \mathsf{List_nodes} \ \subseteq \ \mathsf{Object_alloc} \cup \{tmp_1_10\}) \wedge \\
\qquad \mathsf{comment}\text{“List_PrivateInvcontentAlloc”} \\
\qquad\qquad (\{x\} \cup \mathsf{List_content} \subseteq \mathsf{Object_alloc} \cup \{\mathsf{tmp_1_10}\})))))
\end{array}
$$

Figure 2-5: Verification condition for guarded commands in Figure 2-4

$$\begin{array}{l}
\mathsf{List\_root} \in \mathsf{List}\ \wedge \\
\mathsf{comment}\,\text{``xFresh''}\ (x \notin \mathsf{List\_content})\ \wedge \\
\mathsf{comment}\,\text{``List\_PrivateInvnodesDef''} \\
\qquad \mathsf{List\_nodes} = \{n.\, n \neq \mathsf{null} \wedge (\mathsf{rtrancl\_pt}(\lambda uv.\mathsf{List\_next}\ u = v)\ \mathsf{List\_root}\ n)\}\ \wedge \\
\ldots \\
\mathsf{comment}\,\text{``AllocatedObject''}\,(\mathsf{tmp\_1\_10} \neq \mathsf{null})\ \wedge \\
\ldots \\
\mathsf{comment}\,\text{``AllocatedObject''}\,(\mathsf{List\_data}\ \mathsf{tmp\_1\_10} = \mathsf{null}) \\
\longrightarrow \quad \mathsf{comment}\,\text{``ObjNullCheck''}\,(\mathsf{tmp\_1\_10} \neq \mathsf{null})
\end{array}$$

Figure 2-6: Null dereference proved using Jahob's built-in validity checker

$$\begin{array}{l}
\ldots \\
\mathsf{comment}\,\text{``List\_PrivateInvnodesDefPrecondition''} \\
\qquad \mathsf{List\_nodes} = \{n.\, n \neq \mathsf{null} \wedge (\mathsf{rtrancl\_pt}(\lambda uv.\mathsf{List\_next}\ u = v)\ \mathsf{List\_root}\ n)\}\ \wedge \\
\ldots \\
\mathsf{comment}\,\text{``List\_PrivateInvtreeInvPrecondition''}\,(\mathsf{tree}\ [\mathsf{List\_next}])\ \wedge \\
\ldots \\
\mathsf{comment}\,\text{``AllocatedObject''}\,(\forall y.\mathsf{List\_next}\ y \neq \mathsf{tmp\_1\_10})\ \wedge \\
\mathsf{comment}\,\text{``AllocatedObject''}\,(\mathsf{List\_next}\ \mathsf{tmp\_1\_10} = \mathsf{null}) \\
\ldots \\
\longrightarrow \mathsf{comment}\,\text{``List\_PrivateInvnodesDef''}\,(\{\mathsf{tmp\_1\_10}\} \cup \mathsf{List\_nodes} = \\
\quad \{n.\ n \neq \mathsf{null}\ \wedge\ (\mathsf{rtrancl\_pt}(\lambda uv.((\mathsf{List\_next}(\mathsf{tmp\_1\_10} := \mathsf{List\_root}))u = v))\ \mathsf{tmp\_1\_10}\ n)\})
\end{array}$$

Figure 2-7: Preservation of *nodesDef* invariant proved using MONA (Chapter 6)

$$\begin{array}{l}
\mathsf{comment}\,\text{``xFreshPrecondition''}\ (x \notin \mathsf{List\_content})\ \wedge \\
\mathsf{comment}\,\text{``List\_PrivateInvsizeInv''}\,(\mathsf{List\_size} = \mathsf{cardinality}\ \mathsf{List\_content}) \\
\longrightarrow \mathsf{comment}\,\text{``sizeOk\_sizeInv''}\,(\mathsf{List\_size} + 1 = \mathsf{cardinality}\ (\{x\} \cup \mathsf{List\_content}))
\end{array}$$

Figure 2-8: Proving `noteThat` statement using BAPA decision procedure (Chapter 7)

$$\begin{array}{l}
\ldots \\
\mathsf{comment}\,\text{``List\_PrivateInvcontentDef''} \\
\qquad \mathsf{List\_content} = \{x.\, \exists n.\, x = \mathsf{List\_data}\ n\ \wedge\ n \in \mathsf{List\_nodes}\}\ \wedge \\
\ldots \\
\mathsf{comment}\,\text{``List\_PrivateInvnodesAlloc''}\,(\mathsf{List\_nodes} \subseteq \mathsf{Object\_alloc})\ \wedge \\
\ldots \\
\mathsf{comment}\,\text{``AllocatedObject''}\,(\mathsf{tmp\_1\_10} \notin \mathsf{Object\_alloc}) \\
\ldots \\
\longrightarrow \mathsf{comment}\,\text{``List\_PrivateInvcontentDef''}\,(\{x\} \cup \mathsf{List\_content}\ = \\
\qquad \{x\_\_6.\ \exists n.\ x\_\_6 = (\mathsf{List\_data}(\mathsf{tmp\_1\_10} := x))\ n\ \wedge n \in (\{\mathsf{tmp\_1\_10}\} \cup \mathsf{List\_nodes})\})
\end{array}$$

Figure 2-9: Preservation of *contentDef* invariant proved using SPASS (Chapter 5)

The effect of the `noteThat` statement is to indicate which assumptions to use to prove that *sizeInv* continues to hold. After proving the `noteThat` statement, the formula becomes an assumption and the built-in validity checker proves the fact that *sizeInv* holds at the end of the procedure using simple syntactic matching, as noted above.

4. Jahob proves the remaining 5 sequents by approximating them with first-order logic formulas (Chapter 5) and using the first-order theorem prover SPASS [248]; these sequents do not require reasoning about transitive closure or reasoning about the relationship between sets and their cardinalities. Figure 2-9 displays one of these sequents: the preservation of the *contentDef* invariant.

In the following chapters I first give an overview of Jahob's specification language and the process of generating verification conditions from annotated Java programs. I then focus on the techniques for proving the validity of verification conditions.

Chapter 3

An Overview of the Jahob Verification System

In this chapter I give an overview of the front end of the Jahob verification system. The goal of this chapter is to set the stage for the results in subsequent chapters that deal with reasoning about the validity of formulas arising from verification. Section 3.1 outlines the Java subset that Jahob uses as the implementation language. This subset supports the creation of an unbounded number of data structures containing mutable references and arrays, allowing Jahob users to naturally write sequential imperative data structures. Section 3.2 presents Jahob's specification constructs that allow the users to specify desired properties of data structures using an expressive language containing sets, relations, quantifiers, set comprehensions, and a cardinality operator. I describe the meaning of annotations such as preconditions, postconditions, and invariants, leaving the description of formulas to Chapter 4. Section 3.3 summarizes the process of generating verification conditions from Jahob annotations and implementations. Section 3.4 reviews some previous program verification systems. I conclude in Section 3.5 with an outline of the architecture of the current Jahob implementation and its relationship to different chapters of this dissertation.

3.1 Implementation Language Supported by Jahob

Jahob's current implementation language is a a sequential, imperative, and memory-safe language that supports references, integers, and arrays. Syntactically, the language is a subset of Java [112]. It does not support reflection, dynamic class loading, multi-threading, exceptions, packages, subclassing, or any new Java 1.5 features. Modulo these restrictions, the semantics of Jahob's implementation language follows Java semantics. In fact, because all Jahob specification constructs are written as Java comments, the developers can use both Jahob and existing compilers, virtual machines, and testing infrastructure on the same source code.

Apart from multi-threading, the absence of Java features in Jahob's implementation language does not prevent writing key data structures and exploring their verification. Support for concurrent programming is the subject for future work, but would be possible using current techniques if the language is extended with a mechanism for ensuring atomicity of data structure operations [8, 9, 121].

```
    annotation  ::=  "/*:" specifications "*/"
                 |   "//:" specifications EOL
specifications  ::=  (specification[;] )*
 specification  ::=  contract | specvarDecl | vardefs | invariant
                 |   assert | assume | noteThat | specAssign | claimedby | "hidden"
      contract  ::=  [precondition] [frameCondition] postcondition
  precondition  ::=  "requires" formula
frameCondition  ::=  "modifies" formulaList
 postcondition  ::=  "ensures" formula
   specvarDecl  ::=  ("public" | "private")["static"] ["ghost"]
                     "specvar" ident "::" typeFormula [initialValue]
  initialValue  ::=  "=" formula
       vardefs  ::=  "vardefs" defFormula(defFormula)*
     invariant  ::=  ["public" ["ensured"] | "private"] "invariant" labelFormula
        assert  ::=  "assert" labelFormula ["from" identList]
        assume  ::=  "assume" labelFormula
      noteThat  ::=  "noteThat" labelFormula ["from" identList]
    specAssign  ::=  formula ":=" formula
     claimedby  ::=  "claimedby" ident

   formulaList  ::=  formula ("," formula)*
  labelFormula  ::=  [label] formula
         label  ::=  ident ":" | "(" string ")"
       formula  ::=  formula as described in Section 4.1, in quotes
   typeFormula  ::=  formula denoting a type
    defFormula  ::=  formula of the form v==e
     identList  ::=  ident ("," ident)*
```

Figure 3-1: Syntax of Jahob Specification Constructs

3.2 Specification Constructs in Jahob

This section presents the specification constructs in Jahob. These constructs allow the developer to specify procedure contracts and data structure invariants. The specifications in Jahob can use developer-introduced specification variables, which enables data abstraction and is essential for making the specification and verification tractable.

Figure 3-1 summarizes the syntax of Jahob specification constructs. Note that the developer specifies these constructs in special comments that start with a colon sign. The special comments can only appear at appropriate places in a program, and Jahob expects different constructs at different places of the program. For example, Jahob expects a procedure contract in a special comment after procedure header and before the opening curly brace of the procedure body.

Many specification constructs contain formulas denoting a property of a program state or a relationship between the current and a previous program state. I defer the description of the syntax and semantics of formulas to Section 4.1. Following the convention in Isabelle/Isar [200], formulas are written in quotes, which enables tools such as editors with syntax coloring to treat formulas as atomic entities, ignoring the complexity of formula syntax. For brevity, Jahob allows omitting quotes when the formula is a single identifier.

3.2.1 Procedure Contracts

A procedure contract in Jahob contains three parts:

- a precondition, written as a `requires` clause, stating the property of program state and the values of parameters that should be true before a procedure is invoked (if the developer states no precondition, Jahob assumes the default precondition `True`);
- a frame condition, written as a `modifies` clause, listing the components of state that may be modified by the procedure, meaning that the remaining state components remain unchanged (if the developer specifies no frame condition, Jahob assumes an empty frame condition);
- a postcondition, written as an `ensures` clause, describing the state of the procedure at the end of its invocation, possibly in relationship to parameters and the state at the entry of the procedure.

Jahob uses procedure contracts for assume/guarantee reasoning in a standard way. When analyzing a procedure p, Jahob assumes p's precondition and checks that p satisfies its postcondition and the frame condition. Dually, when analyzing a call to procedure p, Jahob checks that the precondition of p holds and assumes that the values of state components from the frame condition of p change subject only to the postcondition of p, and that the state components not in the frame condition of p remain unchanged. (More precisely, Jahob translates frame conditions into certain conjuncts of the postcondition, according to the rules in Section 3.2.8, allowing a procedure to modify fields of newly created objects without declaring them in the frame condition.)

A Jahob procedure is either public or private. The contract of a public procedure can only mention publicly visible variables.

3.2.2 Specification Variables

The state of a Jahob program is given by the values of program's variables. We call the standard Java variables (static and instance fields) in a Jahob program *concrete variables*. In addition to concrete variables, Jahob supports *specification variables* [96, Section 4], which do not exist during program execution (except potentially for debugging or run-time analysis purposes, which is outside the scope of this dissertation).

Declaring a specification variable. The `specvarDecl` non-terminal of the context-free grammar in Figure 3-1 shows the concrete syntax for specification variables. The developer uses `specvar` keyword to introduce a specification variable and specifies its type, which must be a valid HOL formula type. Optionally, the developer can specify the initial value of the variable, which is an HOL formula. The developer declares each variable, including specification variables, as public or private.

Static versus non-static variables. Similarly as for concrete Java variables, the developer can declare a specification variable as `static`, otherwise the variable is considered non-static. If a variable is non-static, Jahob augments the variable's type with an implicit argument of type `obj`. Therefore, non-static fields in Jahob have the type of the form `obj => t` for some type expression `t`.

Ghost versus non-ghost variables. There are two kinds of specification variables in Jahob according to their relationship to other variables: *ghost variables* and *non-ghost variables.*

A ghost specification variable is independent from other previously introduced variables. The developer introduces a ghost variable using the `ghost` keyword in specification variable declaration. The only way in which the value of a ghost variable `v` changes is in response to a *specification assignment* statement of the form `v := e` within a procedure body. This statement assigns to `v` the value of an HOL formula `e`. The type of `e` must be same as the type of `v`.

A non-ghost specification variable is a function of previously introduced concrete and specification variables. The developer introduces a non-ghost variable `v` using a `specvar` declaration (without the `ghost` modifier) and then defines its value using a `vardefs` keyword followed by a definition of the form `v == e`. Here `e` is an HOL formula that may contain occurrences of other variables. The meaning of such variable is that, in every program state, the value of `v` is always equal to the value of the expression `e`. To make sure that the values of non-ghost variables are well-defined, Jahob requires their definitions to form an acyclic dependency graph. In particular, `v` may not occur in `e`.

3.2.3 Class Invariants

A *class invariant* can be thought of as a boolean-valued specification variable that Jahob implicitly conjoins with preconditions and postconditions of public procedures. The developer can declare a public invariant as private or public (the default annotation is private).

Private class invariants. The developer can use private class invariants to achieve one more level of hiding compared to public specification invariants, by hiding not only the description of a property, but also its existence. A private class invariant is visible only inside the implementation of a class. Procedures outside C should not be able to violate the truth value of a private invariant (a common way to establish this is that the invariant depends only on the state encapsulated inside C). Jahob conjoins private class invariants of a class C to preconditions and postconditions of procedures declared in C. Jahob also conjoins a private class invariant of class C to each reentrant call to a procedure p declared in C_1 for $C_1 \neq C$ to ensure soundness in the presence of callbacks. This policy ensures that the invariant C will hold if $C_1.p$ subsequently invokes a procedure in C.

Public class invariants. A public class invariant relates values of variables visible outside the class. A public class invariant is either *ensured*, if it is prefixed by the `ensured` keyword, or *non-ensured*, otherwise.

If a public invariant I of class C is ensured, the clients of C can assume I but need not check that I is preserved, because C by itself guarantees that I can never be violated. When verifying class C, Jahob therefore both 1) conjoins I to preconditions and postconditions of procedures inside C and 2) makes sure that no sequence of external actions can invalidate I. A sufficient condition for 2) is that I only depends on parts of state that are encapsulated inside C. A more general condition allows a public ensured invariant I to depend on `Object.alloc` of all allocated objects, even though this set is not encapsulated inside C.

Jahob verifies this condition by establishing that I is preserved under state changes that change `Object.alloc` in a monotonic way and preserve the state encapsulated in C.

If the developer declares a public invariant I of class C as non-ensured, then the clients of C share the responsibility for maintaining the validity of I, which means that Jahob conjoins I to methods of clients of C as well. Public non-ensured invariants can therefore decrease the modularity in a verification task, because they add conjuncts to the contracts of procedures declared elsewhere.

Specifying when an invariant should hold. The developer can effectively adjust Jahob's simple policy for when an invariant holds, as follows.

To require an invariant to hold more frequently, the developer can explicitly restate it using the desired annotations, such as a precondition or postcondition of a private procedure, a loop invariant, or an assertion inside a procedure. The developer can name the invariants to avoid restating them. Jahob's construct (`theinv` L) within a formula expands into the definition of the invariant labelled L. Alternatively, the developer can introduce a boolean specification variable and provide the invariant as its definition.

To require an invariant I to hold less frequently, introduce a boolean-valued static ghost variable b and restate the invariant as $b \rightarrow I$. To temporarily disable the invariant, perform an abstract assignment $b :=$ `False`, which makes $b \rightarrow I$ trivially true without requiring I to hold. To enable the invariant, perform an abstract assignment $b :=$ `True`. To enable or disable an invariant on a per-object basis, declare b as an instance ghost variable as opposed to a static ghost variable.

3.2.4 Encapsulating State in Jahob

Encapsulation of state in Jahob is driven by the need to prove a simulation relation [77] between the program that uses the implementation and the program that uses the specification of a class. When verifying a class `C`, Jahob establishes that the implementation of `C` satisfies the public specification of `C` by verifying the following three conditions:

1. showing that the initial state of `C` satisfies class invariants of `C`;

2. showing that each public procedure in `C` preserves class invariants and establishes its postcondition and the frame condition;

3. showing that any external sequence of actions

   (a) leaves the values of specification variables of `C` unchanged; and

   (b) preserves the class invariants.

Hiding the definition of a variable `v` in the public specification of a class `C` means that Jahob will assume that `v` does not change except through invocations of procedures of `C`. The condition (3) ensures that such hiding is sound. A class invariant can be thought of as an implicit boolean-valued specification variable, so (3b) is a special case of (3a). Jahob relies on encapsulation of state as well as the properties of Java semantics to ensure Condition 3. The detailed description of verifying the condition 3 are beyond the scope of this thesis; we only describe the three mechanisms that developers have at their disposal for encapsulating state in Jahob.

Private variables. Declaring a variable private to the class makes it impossible for methods outside the class to modify its value. Note that the value of a reference variable `x` is simply the object identifier, so making a reference `x` private does not prevent fields of the object `x` from being modified from outside the class.

Claimed fields. If `f` is a field declared in class `N` the developer can make `f` be part of representation of another class `C` by using an annotation `claimedby C` in front of the declaration of `f`. Semantically, `f` is a function from objects to objects, so `claimedby C` declaration corresponds to declaring such function `f` as a private variable inside `C`. Claimed fields allow all fields of one class to be declared local to another class, which is useful for helper classes that represent memory cells used in implementation of data structures. Claimed fields also allow multiple classes to claim fields on one class, as in multi-module records in [54], or as in Hob's formats [163]. Such separation of fields is useful for modular analysis of objects that participate in multiple data structures represented by distinct classes.

Object hiding. When building layered abstractions using instantiable data structures, different data structure instances can occur as part of representation of different data structures. A static state partitioning mechanism such as private and claimed fields is not sufficient to specify the representations of such layered data structures (although its power could be substantially increased by a simple parameterization mechanism such as the one present in parameterized ownership types [39]). To allow more flexible notations of encapsulation, Jahob provides a notion of a *hidden set* of objects. A hidden set `C.hidden` associated with class `C` stores the set of objects that are part of the representation of the entire class `C`. A method in `C` can add a newly allocated object to `C.hidden` using the `hidden` modifier before the `new` keyword when allocating an object. The developer can use the `hidden` variable in class invariants to specify, for example, that the desired reachable objects are part of the representation of `C`. Jahob currently uses a simple syntactic checks and assertions to ensure that no procedure in `C` leaks a reference to an object in `hidden` by returning it, passing it as a parameter, or storing it in non-private fields of objects that are not in the `hidden` set. Because hidden objects are allocated inside `C` and never leaked, they are not observable outside `C`. Procedures in `C` can therefore omit modifications to these objects, and `C` can use these objects in definitions of specification variables and invariants.

To enable the invocation of methods on such hidden objects, Jahob needs to know that these methods do not introduce new aliases to hidden objects. More precisely, the desired property is the following:

> If the receiver parameter of a method `D.p` is the only way for `D.p` to access the receiver object, then `D.p` will not introduce any new aliases to the receiver object (by e.g. storing it into another field or global variable).

This property appears to be a good default specification for the receiver parameter. In the data structures that I have examined it can be enforced using simple syntactic checks.

3.2.5 Annotations within Procedure Bodies

The developer can use several kinds of annotations inside a procedure body to refine the expectations about the behavior of the code, help the analysis by stating intermediate facts, or debug the verification process.

Loop invariant. The developer states a loop invariant of a `while` loop immediately after the `while` keyword using the keyword `invariant` (or `inv` for short). A loop invariant must be true before evaluating the loop condition and it must be preserved by each loop iteration. The developer can omit from a loop invariant the conditions that depend only on variables not modified in the loop because Jahob uses a simple syntactic analysis to conclude that the loop preserves such conditions. In this thesis I assume that the developer has supplied all loop invariants, although Jahob contains a specialized loop invariant inference engine [254, 253, 211].

Local specification variables. In addition to specification variables at the class level, the developer can introduce ghost or non-ghost specification variables that are local to a particular procedure and are stated syntactically as statements in procedure body. Such variables can be helpful to state relationships between the values of variables at different points in the program, and can help the developer or an automated analysis in discovering a loop invariant.

Specification assignment. A specification assignment changes the value of a ghost specification variable. If `x` is a variable and `f` a formula, then a specification assignment `x := e` changes the value of `x` to the value of formula `e`. Jahob also supports a field assignment of the form `x.f := e` which is a shorthand for `f := f(x := e)` where the second occurrence of `:=` denotes the function update statement (see Figure 4-3).

Assert. An `assert e` annotation at program point p in the body of the procedure requires the formula `e` formula to be true at the program point p. The developer can use an assert statement similarly to Java assertions that produce run-time checks. An important difference is that a Jahob assertion produces a proof obligation that guarantees that `e` will be true in all program executions that satisfy the precondition of the procedure, and not merely in those preconditions that are exercised during a particular finite set of program executions. Another difference compared to Java assertions is that the condition being checked is given as a logical formula instead of a Java expression.

Assume. An `assume e` statement is dual to the assert statement. Whereas an assert requires the verification system to demonstrate that `e` holds, an assume statement allows the verification system to assume that `e` is true at a given program point. Note that developer-supplied use of `assume` statements violates soundness. The intended use of `assume` is debugging, because it allows the system to verify a procedure under the desired restricted conditions. For example, a specification statement `assume "False"` at the beginning of a branch of `if` statement means that the system will effectively skip the verification of that branch.

NoteThat. A `noteThat e` statement is simply a sequential composition of `assert e` followed by `assume e`. It is always sound for the developer to introduce a `noteThat` statement because the system first checks the condition `e` before assuming it. Therefore, `noteThat e` is semantically equivalent to `assert e`, but instructs the verification system to use `e` as a useful lemma in proving subsequent conditions.

Specifying assumptions to use. `noteThat` f and `assert` f statements can optionally use a clause `from` $l_1, \ldots, l_n$ to specify a list of labels that identify the facts from which the formula f should follow. This construct can be very helpful when the number and the complexity of the invariants becomes large. The identifiers l_i can refer to the labels of facts introduced by any previous `noteThat` and `assume` statements, preconditions, invariants, or the conditions encoding a path in the program. Currently, Jahob implements this mechanism as follows. Before proving a sequent, Jahob removes all sequent assumptions whose comment does not contain as a substring any of the identifiers $l_1, \ldots, l_n$.

3.2.6 Meaning of Formulas

This section summarizes key aspects of the semantics of formulas in Jahob annotations.

Interpreting state variables in a formula. Formulas in Jahob annotations can mention state variables of the Jahob program, such as instance field names, static variables, parameters, local variables, as well as the developer-supplied specification variables. As in standard Hoare logic [122], a program variable in a formula typically denotes variable's value in the *current state* of the program. For an annotation at program point p inside procedure body, the current state is the state at program point p. For a precondition, the current state is the state at procedure entry, and for postcondition the current state is the state at the end of procedure execution. For a class invariant, the current state is given by the annotation to which the invariant is conjoined: when the invariant becomes part of a precondition, then the current state is the state at procedure entry, and when it becomes part of a postcondition, the current state is the state at procedure end. The value of a specification variable with a definition `v==e` in a given state is the value of the expression `e` in the same state.

`old` **construct.** Procedure postcondition and the annotations within procedure body often need to refer to the value of a variable at the entry of the currently executing procedure. Writing an operator `old` in front of a variable name allows the developer to denote the value of the variable at procedure entry. For example, a formula `x = 1 + old x` in procedure postcondition denotes the fact that the procedure increases the value of the variable `x` by one. As a shorthand, Jahob allows the developer to apply the construct `old` to expressions as well. The value of such expression is given by propagating the `old` operator to the identifiers within the expression, so `old (x + y)` denotes `old x + old y`.

Representing memory cells. In Jahob terminology, the notion of an *object* approximately corresponds to what Java calls a reference: an object is merely given by its unique identifier and not by the content of its fields. Jahob represents a field of a class as a function mapping objects to objects or integers. Jahob's logic is a logic of total functions, so the value of a field is defined even for the null object. For concreteness, Jahob assumes that `f null = null` for a field `f` mapping objects to objects. All objects, regardless of their class, have type `obj` in specifications. Jahob represents classes as immutable sets of objects: if the dynamic type of a Java object `x` is `C` or a subclass of `C`, then `x:C` holds.

Dereferencing. If `x` is an object and `f` is a field, then `f x` denotes the value of `f` for the object `x`. Jahob's language also supports postfix notation for functions, `x..f`, making specifications resemble the corresponding Java expression `x.f`. Note that Jahob uses the single dot "`.`" to qualify names of variables (including fields), so if `f` is a field declared in class `C` then `C.f` is used outside `C` to unambiguously denote the name of the field. Therefore, the expression `x..C.f` denotes the value of the Java expression `x.f`. The type system of Jahob formulas allows applying any field to any object; when object `x` has no field `f` during Java program execution, Jahob assumes that `x..f = null`.

Semantics of object allocation. Jahob assumes that the set of all objects in the universe is a fixed finite set that does not change, but whose size is not known. This model provides simple semantics for reasoning about values of fields of objects at different program points. The set of all objects (and, therefore, the scope of all quantifiers that range over objects) is the same at every program point. To model object allocation, Jahob uses a special set of objects called `Object.alloc` that stores all objects that have been allocated (including the special `null` object). Jahob considers `null` to be an allocated object as a matter of convenience: with this assumption, all static variables, parameters, and local variables point to allocated objects, and all fields of allocated objects point to allocated objects. In addition to allocated objects, Jahob allows the existence of non-null objects that are *yet to be allocated*, which we call simply *unallocated objects*. The object allocation statement, given by the parameterless `new C()` statement in Java that invokes the default constructor, non-deterministically selects an unallocated object and inserts it into the set `Object.alloc`. Note that the developer can always avoid relying on the existence of unallocated objects by explicitly guarding the quantification over objects by membership in `Object.alloc`.

3.2.7 Receiver Parameters

Java supports implicit receiver parameters in methods and denotes them by the `this` keyword. Furthermore, it allows writing `f` instead of `this.f`. In the same spirit, Jahob supports receiver parameters in specifications. Before the verification process Jahob makes receiver parameters explicit by translating each instance method into a procedure that has an additional parameter named `this`.

Implicit receiver parameter in formulas. When Jahob finds an occurrence of a non-static variable `f` in a formula (regardless of whether it is a specification variable or a concrete variable), and the occurrence is *not* of the form `x..f` for some expression `x`, Jahob will replace the occurrence of `f` with `this..f`. Jahob performs this replacement in preconditions, postconditions, frame conditions, class invariants, specification variable definitions, and annotations within procedure bodies. The developer can prevent the addition of the receiver parameter by fully qualifying `f` as `C.f` where `C` is the class where the field `f` is declared.

Implicit receiver in non-static specification variables. When a developer introduces a non-static specification variable of a declared type `t`, Jahob transforms the type into `obj => t` where the new argument denotes the receiver parameter. Moreover, if the developer introduces a definition `v==e` for such a variable, Jahob transforms it into `v==(\<lambda> this. e)`, supplying the binding for any occurences of the receiver parameter in `e`. (These occurrences can be explicit or introduced by the previous transformation.)

Implicit universal quantification of class invariants. When a class invariant `I` of a class `C` contains an (implicit or explicit) occurrence of the receiver parameter, Jahob transforms it into the invariant

```
ALL this. this : Object.alloc & this : C --> I
```

stating that the invariant holds for all allocated objects of the class `C`. In this way, Jahob reduces the invariant on individual objects to a global invariant, which is flexible and simple, although it generates proof obligations with universal quantifiers. The fact that the bound variable `this` ranges only over allocated objects means that a class invariant on an instance field need not hold in the initial state of an object. It suffices that a method p that allocates an object x establishes the invariant for x by the end of the execution of p and before any other method calls within p.

3.2.8 Meaning of Frame Conditions

When interpreting a procedure contract, Jahob transforms the frame condition into a formula f stating that certain parts of program state remain the same. Jahob then conjoins f to the postcondition of the contract.

$$x.f = y \quad \rightsquigarrow \quad f := f(x := y)$$

$$a[i] = b \quad \rightsquigarrow \quad \mathsf{arrayState} := \mathsf{arrayState}(a := (\mathsf{arrayState}\, a)(i := b))$$

Figure 3-2: Transformations of Assignments using Function Update

A frame condition specification is a list of expressions, syntactically written using the same syntax for formula as in, for example, `assert` statements. However, Jahob interprets these formulas differently than formulas in other annotations: each frame condition formula denotes a set of memory locations. If the formula is a variable `v`, the interpretation is the set of all locations corresponding to this variable (this is a bounded set for variables of type `obj` or `int`, or a statically unbounded set if `v` is the name of a field). Writing the field name `C.f` in a frame condition, where `C` is a class name, indicates that the procedure may modify the `f` field of all allocated objects of class `C`. To indicate that the procedure modifies the field `f` only for a specific object `x`, the developer can use the frame condition `x..f`. Note that, for non-static methods, writing the name of a field `f` by itself (as opposed to `x..f` or `C.f`) in frame condition causes Jahob to expand `f` to `this..f`.

Jahob's frame condition constructs currently have limited expressive power. However, the developer can always write an overapproximation of the desired set of modified locations (because the set of all fields and global variables is finite). After overapproximating this set of locations, the developer can restrict this set by explicitly stating in the postcondition that the desired values are preserved, using the full expressive power of Jahob's logic and the `old` construct to refer to the initial values of variables.

The encapsulation of state means that Jahob uses a slightly different formula f when proving the postcondition at the end of the procedure and when assuming the postcondition after a procedure call. This difference allows a procedure p in class `C` to implicitly modify certain locations without mentioning them in the frame condition. These locations include private and claimed variables, any objects that p allocates during its execution, and any objects that are in `C.hidden`. When analyzing a procedure call to p from a class `D` different from `C`, Jahob can soundly ignore these implicit modifications. When, on the other hand, analyzing a potentially reentrant call inside a procedure of class `C`, Jahob must take into account these modifications of private state of `C` to ensure soundness.

3.3 Generating Verification Conditions

This section describes the process of generating verification conditions, which are proof obligations that entail that Jahob procedures correctly implement their specification, preserve class invariants, and cause no run-time errors. I start with a brief summary of transformation of the input language into a guarded-command language, then focus on transforming the guarded-command language into logical formulas in the presence of specification variables.

3.3.1 From Java to Guarded Command Language

Jahob first transforms the annotated Java input program into a guarded-command language with loops and procedure calls. This translation flattens Java expressions by introducing fresh variables, converts the expressions to HOL formulas, and inserts assertions that check

$$
\begin{array}{rcll}
c & ::= & x := \mathsf{formula} & \text{(assignment statement)} \\
 & | & \mathsf{havoc}\ x & \text{(non-deterministic change of } x\text{)} \\
 & | & \mathsf{assume\ formula} & \text{(assume statement)} \\
 & | & \mathsf{assert\ formula} & \text{(assert statement)} \\
 & | & c_1\,;c_2 & \text{(sequential composition)} \\
 & | & c_1 \,\Box\, c_2 & \text{(non-deterministic choice)}
\end{array}
$$

Figure 3-3: Loop-free guarded-command language

$$
\begin{array}{rcl}
\mathsf{wlp}(\mathsf{havoc}\ x, G) & \equiv & \forall x.\ G \\
\mathsf{wlp}(\mathsf{assert}\ F, G) & \equiv & F \land G \\
\mathsf{wlp}(\mathsf{assume}\ F, G) & \equiv & F \to G \\
\mathsf{wlp}(c_1\,;c_2, G) & \equiv & \mathsf{wlp}(c_1, \mathsf{wlp}(c_2, G)) \\
\mathsf{wlp}(c_1 \,\Box\, c_2, G) & \equiv & \mathsf{wlp}(c_1, G)\ \land\ \mathsf{wlp}(c_2, G)
\end{array}
$$

Figure 3-4: Weakest Precondition Semantics

for run-time errors such as null dereference, array out of bounds access, and type cast failure.

During this transformation Jahob also maps field assignments and array assignments into assignments to state variables, using the rules in Figure 3-2. Jahob treats a field f as a function mapping objects to objects, so a field assignment to $x.f$ translates into an assignment that assigns to f the function updated at point x. Similarly, Jahob represents the content of all arrays as a function arrayState that maps an object a and an array index i to the content (arrayState $a\,i$) of the array object a at index i. An assignment to array therefore updates arrayState at object a and index i.

Jahob uses loop invariants to eliminate loops from the guarded command language using loop desugaring similar to [98, Section 7.1]. Furthermore, Jahob replaces each procedure call p with guarded command statements that over-approximate procedure call using the contract of p. Figure 3-3 shows the syntax of the resulting loop-free guarded-command language.

3.3.2 Weakest Preconditions

Figure 3-4 shows the standard weakest precondition semantics of the guarded-command language in Figure 3-3. For exposition purposes we omit the assignment statement from the guarded-command language and instead represent $x := e$ as

$$\mathsf{havoc}\ x;\ \mathsf{assume}\ (x = e)$$

where e does not contain variable x.

Jahob generates a proof obligation for the correctness of a procedure as a weakest precondition of a guarded-command language command corresponding to the procedure. The rules in Figure 3-3 show a close correspondence between the guarded command language and the generated logical formulas. The weakest preconditions for the guarded-command language are conjunctive [20], meaning that, for every guarded command c, the function $\lambda F.\mathsf{wlp}(c, F)$ is a morphism that distributes (modulo formula equivalence) over finite and infinite conjunctions. Note that Jahob's reasoning procedures can skolemize the quantifiers introduced by the havoc statements, because these quantifiers occur positively in the verification condition.

3.3.3 Handling Specification Variables

In the presence of dependent (non-ghost) specification variables, weakest precondition computation must take into account variable dependencies.

Limitation of the substitution approach. A simple way to address variable dependencies is to substitute in all relevant definitions of dependent variables before computing weakest preconditions. Such approach is sound and we have used it in [267]. However, this approach would prevent Jahob's reasoning procedures from taking full advantage of abstraction, and would limit the possibilities for synergy between different reasoning techniques. For example, a decision procedure for BAPA can prove properties such as $|A \cup B| \leq |A| + |B|$, but is not directly applicable if the sets A, B are expanded by their definitions in terms of, for example, reachability in a graph.

Encoding dependencies into guarded commands. Jahob produces verification conditions that preserve dependent variables. The key observation of this approach is that a dependent variable x behaves like a ghost variable with implicit updates that occur whenever one of the variables affecting the definition of x changes. We say that a variable x depends on a variable y if x occurs in the formula defining the value of y; transitive dependency is the transitive closure of this relation on variable names. (Jahob requires variable definitions to have no cyclic dependencies.)

For each statement havoc y in the guarded-command language Jahob's verification condition generator therefore finds all variables $x_1, \ldots, x_n$ transitively affected by the change of y. Let f_i be the definition of variable x_i for $1 \leq i \leq n$. The verification condition generator then works as if each havoc y was implicitly followed by

$$\begin{array}{l}
\mathsf{havoc}\ x_1; \\
\quad \ldots \\
\mathsf{havoc}\ x_n; \\
\mathsf{assume}\ (x_1 = f_1); \\
\quad \ldots \\
\mathsf{assume}\ (x_n = f_n)
\end{array}$$

To eliminate some of the unnecessary assumptions, the verification condition generator potentially reorders the statements assume $(x_i = f_i)$ and introduces each assume statement assume $(x_i = f_i)$ on demand, only when the variable x_i occurs in the second argument of wlp in Figure 3-4.

3.3.4 Avoiding Unnecessary Assumptions

The verification condition generator also contains a general-purpose mechanism for dropping certain assumptions to reduce the size of generated verification conditions. The transformations into guarded-command language can designate each assume statement as being "about a variable x". For example, Jahob considers assume statements of the form $x : C$ where C is a class name and x a local variable, to be "about variable" x. The verification condition generator soundly approximates the computation of $\mathsf{wlp}(\mathsf{assume}\ f, G)$ with a potentially stronger statement G when the $\mathsf{assume}\ f$ statement is designated as being about a variable x that does not occur in G. Such elimination of unnecessary assumptions not only makes verification conditions easier to prove automatically, but also makes them more readable, which is important for interactive theorem proving and error diagnosis.

3.4 Related Work

I next discuss existing software verification systems and their relationship to Jahob. I leave the discussion of the logics used in specification languages to Section 4.5 and leave the discussion of particular reasoning techniques (along with the results achieved using them) to Section 5.11, Section 6.5, and Section 7.10.

Software verification tools. Software verification tools based on verification condition generation and theorem proving include the Program Verifier [141], Gypsy [109], Stanford Pascal Verifier [195], VDM [135], the B method [2], RAISE [73], Larch [116], Krakatoa [92, 178], Spec# [28], ESC/Modula-3 [83], ESC/Java [96], ESC/Java2 [51], LOOP [133], KIV [25], KeY [4]. Notations such as Z [255] are also designed for formal reasoning about software, with or without tool support. In the subsequent chapters I show that Jahob provides additional levels of automation in comparison to most of these tools, at least for properties naturally expressible in first-order logic, monadic second-order logic of tree-like graphs, and Boolean Algebra with Presburger Arithmetic, as well as for combinations of such properties.

Counterexample search. Alloy notation [131] enables formal descriptions of software designs and is supported by a finite model generator based on translation to propositional logic [242]. Alloy has also been applied to the analysis of software implementations [80]. Like approaches based on model checking of software [118, 66, 192, 107], and run-time checking [53, 79] combined with test generation [179], Alloy tool can identify violations of specification in rich specification languages, but does not guarantee the absence of infinite classes of bugs as the verification approaches.

Proving the absence of run-time errors. ESC/Modula-3 [83] and ESC/Java [96] emphasize usability over soundness and use loop unrolling to permit users to omit loop invariants. Techniques such as [95, 97] can infer loop invariants for some simple properties, but have not been proven to work on complex data structures. SparkAda [99] reports a high success rate on statically proving the absence of run-time errors. Our experience with Jahob suggests that a high percentage of errors are easy to prove because they follow from preconditions and path conditions for conditional branches, but the remaining small percentage requires complex data structure invariants and therefore requires more sophisticated analyses (such as shape analyses discussed in subsequent chapters of this dissertation).

Methodology for specifying object-oriented programs. Although data refinement methods have been studied in detail [77], their practical application to object-oriented programs is still the subject of ongoing research. The difficulty arises from the desire to extend the notion of simulation relations to the scenario with an unbounded number of instantiated mutable objects which have pointers to auxiliary objects. Ownership types [39] can indicate the relationship between the object and its auxiliary objects. Ownership types were used to prove specialized classes of properties, but they would be useful in the context of Jahob as well as a generalization of Jahob's `claimedby` construct. Another way of making `claimedby` construct more general would be to incorporate constructs such as scopes [163] into Jahob. Naumann and Barnett [193] make explicit the binary ownership relation that is implicit in ownership types. As a result, it is possible to use such binary relation in the specification language. Jahob also allows the use of the `hidden` set specification variable in annotations such as invariants. However, Jahob has only one `hidden` set variable per module. Jahob transforms per-object invariants into global invariants by implicitly quantifying over `this` and relies on invariants to determine the constraints on state sharing between objects of the same class. Jahob and Spec# [28] use different notions of encapsulation, which is reflected in the semantics of procedure calls. Spec# assumes implicitly that a procedure can implicitly modify all objects that are "packed", because such objects must have their invariants preserved across procedure calls. In contrast, Jahob assumes that all objects that were allocated before procedure call and are not explicitly mentioned in modifies clauses have their fields unchanged. As a result, Jahob's verification conditions for procedure calls are simpler. On the other hand, Jahob prevents references to objects encapsulated by the `hidden` construct to be visible outside the class (but allows such references to objects encapsulated using the `claimedby` construct). Jahob currently prevents representation exposure using simple syntactic checks. More flexible approaches that provide similar guarantees can be incorporated into Jahob.

3.5 Jahob System Implementation

This completes my overview of Jahob and it's techniques for converting annotated programs written in a subset of Java into logical formulas. The overview omits many details, but hopefully gives a general idea of how Jahob reduces the verification problem to the problem of validity of HOL formulas. The remainder of this dissertation focuses on these techniques for deciding the validity of HOL formulas. Figure 3-5 gives an overview of the techniques implemented in Jahob and indicates the chapters in which I discuss them.

Jahob is written in Objective Caml [170], currently has about 30 thousand lines of code (including implementation of techniques not described in this dissertation). It is under active development and is freely available. Its goal in the near future is to remain a research prototype for exploring verification techniques for complex program properties.

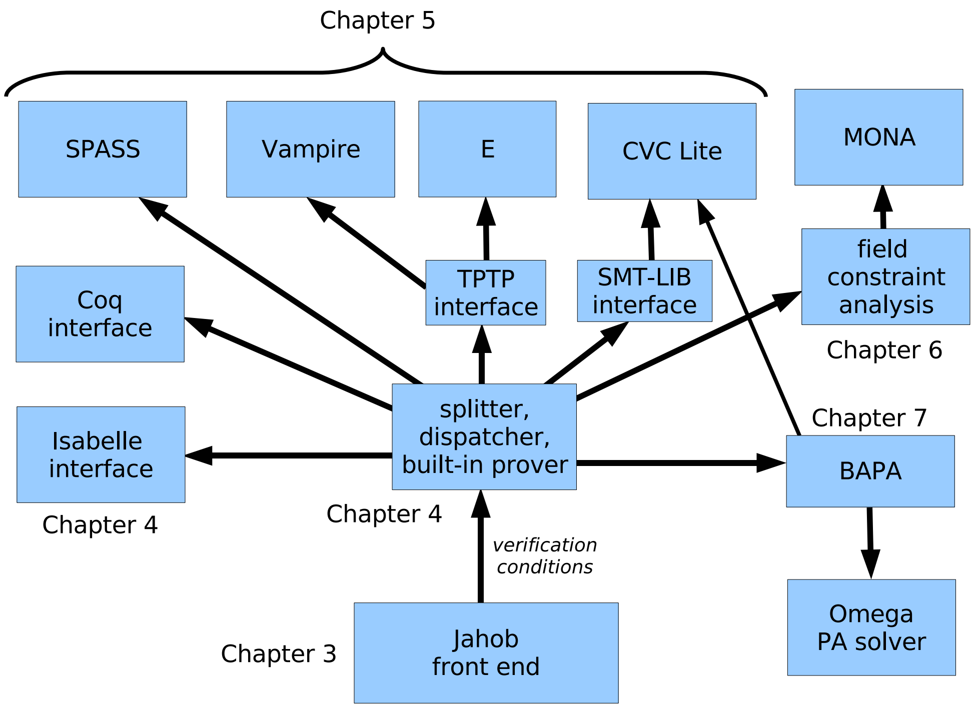


Figure 3-5: Architecture of the Jahob Data Structure Analysis System

Chapter 4

A Higher-Order Logic and its Automation

This chapter presents Jahob's formula language for proof obligations and program annotations, as well as my approach for proving formulas in this language. This language is a fragment of the classical higher-order logic [11, Chapter 5]; I call it HOL and describe it in Section 4.1. I use the standard model [11, Section 54] as the HOL semantics, where a function type is interepreted as the set of all total functions from the domain set to the codomain set. HOL is very expressive and directly supports reasoning about sets and relations. This makes HOL appropriate for the verification methodology presented in the previous chapter, which uses sets and relations as specification variables.

In general, proving arbitrarily complex HOL formulas requires interactive theorem proving. I therefore devote Section 4.2 to describing Jahob's interface to the Isabelle interactive theorem prover [201] and describe reasons why such interfaces are useful in the context of a verification system. However, the main contributions of this dissertation are more automated techniques for proving HOL formulas, so I spend the remaining sections and chapters on the automation problem.

My approach to automation is to develop and adapt specialized techniques for reasoning about fragments of HOL and then translate as much as possible from HOL into these fragments. The idea of the translation is to soundly approximate subformulas by mapping HOL constructs to the constructs supported in the fragment, and conservatively approximating constructs that cannot be mapped in a natural way. Section 4.3 gives a skeleton of such translations and presents several transformations that we found generally useful in preprocessing HOL formulas. The subsequent chapters describe particular translations that we found useful in the Jahob system and explains how to prove the translated formulas. Specifically, I describe a translation into first-order logic in Chapter 5, a translation into monadic second-order logic of trees in Chapter 6, and a translation into Boolean Algebra with Presburger Arithmetic (BAPA), as well as a decision procedure for BAPA, in Chapter 7.

4.1 Higher Order Logic as a Notation for Sets and Relations

Figure 4-1 shows the core syntax of HOL used in Jahob. The basis for Jahob's version of HOL is the Isabelle/HOL language described in [201]. Jahob contains only minor syntactic sugar defined in a small theory file provided with the Jahob distribution. In general, my

$$
\begin{array}{rcll}
f & ::= & \lambda x :: t.\, f & \text{lambda abstraction} \\
 & & \multicolumn{2}{l}{(\lambda \text{ is also written } \texttt{\textbackslash<lambda>} \text{ and } \texttt{\%})} \\
 & | & f_1\, f_2 & \text{function application} \\
 & | & = & \text{equality} \\
 & | & x & \text{variable or constant} \\
 & | & f :: t & \text{typed formula} \\
t & ::= & \mathsf{bool} & \text{truth values} \\
 & | & \mathsf{int} & \text{integers} \\
 & | & \mathsf{obj} & \text{uninterpreted objects} \\
 & | & \text{'}\alpha & \text{type variable} \\
 & | & t_1 \Rightarrow t_2 & \text{total functions} \\
 & | & t\ \mathsf{set} & \text{sets} \\
 & | & t_1 * t_2 & \text{pairs}
\end{array}
$$

Figure 4-1: Core syntax of HOL

$$
\begin{array}{rcl}
[\![\lambda x :: t.\, f]\!]e & = & \{(v, [\![f]\!](e[x := v])) \mid v \in [\![t]\!]\} \\
[\![f_1\, f_2]\!]e & = & ([\![f_1]\!]e)\,([\![f_2]\!]e) \\
[\![\mathsf{=} :: t \Rightarrow t \Rightarrow \mathsf{bool}]\!] & = & \{f \mid \forall u, v \in [\![t]\!].\ (f u)v = \mathsf{true} \leftrightarrow (u = v)\} \\
[\![x]\!]e & = & e\, x \\
[\![f :: t]\!]e & = & [\![f]\!]e
\end{array}
$$

$$\{\mathsf{true}, \mathsf{false}\}, \mathbb{Z}, \mathbb{O} \text{ are pairwise disjoint sets}$$

$$\mathbb{O} \text{ is a finite set of unspecified size } N$$

$$
\begin{array}{rcl}
[\![\mathsf{bool}]\!] & = & \{\mathsf{true}, \mathsf{false}\} \\
[\![\mathsf{int}]\!] & = & \mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\} \\
[\![\mathsf{obj}]\!] & = & \mathbb{O} = \{o_1, o_2, \ldots, o_N\} \\
[\![t_1 \Rightarrow t_2]\!] & = & \{f \mid f \subseteq [\![t_1]\!] \times [\![t_2]\!] \wedge \forall x \in [\![t_1]\!].\exists_1 y \in [\![t_2]\!].\ (x, y) \in f\} \\
[\![t\ \mathsf{set}]\!] & = & \{f \mid f \subseteq [\![t]\!]\} \\
[\![t_1 * t_2]\!] & = & \{(x_1, x_2) \mid x_1 \in [\![t_1]\!] \ \wedge\ x_2 \in [\![t_2]\!]\}
\end{array}
$$

Figure 4-2: Semantics of HOL formulas and ground types

intention is that the axiomatic definitions in Isabelle theory files serve as the authoritative semantics of Jahob's HOL.

My first goal in this section is to summarize the relevant HOL subset and to clarify the semantics of formulas. For an introduction to classical HOL as a foundation of mathematics see [11, Chapter 5], for an introduction to the use of HOL in program semantics see [20, Chapter 3], and for a mechanization of HOL in the HOL system see [110]. My second goal is to motivate the decision to use the Isabelle/HOL language as a basis for formula notation in Jahob.

Lambda calculus core. As Figure 4-1 suggests, the basis of HOL is typed lambda calculus [26]. The term $\lambda x :: t.\, f$ denotes a function h such such that $h(x) = f$ for all x, where f typically contains x as a free variable. The $h\, x$ notation denotes the function application $h(x)$ where both h and x. In lambda calculus terminology, expressions are usually called *terms*, but we call them *formulas*, keeping in mind that a formula can denote not only a truth value but also a value of any other type.

Equality. Equality is a curried function of two arguments that returns true if and only if the arguments are equal; we write it in the usual infix notation.

Type system. Jahob uses simply typed lambda calculus [26]. As a matter of convenience, Jahob allows omitting types for bound variables and infers them using Hindley-Milner type inference [208, Chapter 9], taking into account any explicitly supplied types for subformulas. The developer can in principle use parametric polymorphism in HOL formulas, but Jahob expects that, after applying simple transformations such as definition substitution and beta-reduction, all identifier occurrences can be inferred to have ground types. When a formula with a subformula of the form $f :: t$ type checks for t a ground type, then the meaning of formula f is an element of the set representing the meaning of the type t. The choice of primitive types in Jahob reflects the intended use of formulas in verification:

- bool represents truth values and is standard in lambda calculi used to represent logics. (Like the HOL system [110] and unlike the Isabelle framework, we do not distinguish in our formulas generic meta-logic and the object logic but directly use the object logic.)

- int represents mathematical integers. Integers model bounded integers in a Java program, express cardinalities of sets, index arrays, and represent keys in finite maps.

- obj represents objects in the heap. Such objects are uninterpreted elements in the logic (admitting only mathematical equality), because their content is given by the values of their fields, which are represented as functions from objects to other values. We assume that the set of objects is finite, but do not specify its size. We found this approach to be sufficient for reasoning about safety properties of programs.

- $t\,\mathsf{set}$ denotes sets of elements of type t; it is isomorphic to $t \Rightarrow \mathsf{bool}$, but we introduce it in the spirit of common notational practice that is also reflected in Isabelle/HOL.

- $t_1 * t_2$ denotes the type whose interpretation is the Cartesian product of the interpretations $[\![t_1]\!]$ and $[\![t_2]\!]$.

Standard semantics. Figure 4-2 presents the standard set theoretic semantics of HOL, which interprets a function type $t_1 \Rightarrow t_2$ as the set of all total functions from $[\![t_1]\!]$ to $[\![t_2]\!]$.

constant (ASCII)	type	semantics	notation
$\neg$ ($\sim$)	$\mathsf{bool} \Rightarrow \mathsf{bool}$	negation	prefix
$\wedge, \vee, \rightarrow$ (`&`, `|`, `-->`)	$\mathsf{bool} \Rightarrow \mathsf{bool} \Rightarrow \mathsf{bool}$	and, or, implication	infix
$\forall$ (`ALL`)	$('\alpha \Rightarrow \mathsf{bool}) \Rightarrow \mathsf{bool}$	$[\![\forall]\!]\, h \ = \ \forall v.\, hv$	$\forall x.\, f$ denotes $\forall\, (\lambda x.\, f)$
$\exists$ (`EX`)	$('\alpha \Rightarrow \mathsf{bool}) \Rightarrow \mathsf{bool}$	$[\![\exists]\!]\, h \ = \ \exists v.\, hv$	$\exists x.\, f$ denotes $\exists\, (\lambda x.\, f)$
Collect	$('\alpha \Rightarrow \mathsf{bool}) \Rightarrow {'\alpha}\ \mathsf{set}$	$\mathsf{Collect}\, h = \{v \mid h\, v\}$	$\{x.\ f\}$ denotes $\mathsf{Collect}\, (\lambda x.\, f)$
$\in$ (`:`)	$'\alpha \Rightarrow {'\alpha}\ \mathsf{set} \Rightarrow \mathsf{bool}$	set membership	
$\cup$ (`Un`)	$'\alpha\ \mathsf{set} \Rightarrow {'\alpha}\ \mathsf{set} \Rightarrow {'\alpha}\ \mathsf{set}$	union	
$\cap$ (`Int`)	$'\alpha\ \mathsf{set} \Rightarrow {'\alpha}\ \mathsf{set} \Rightarrow {'\alpha}\ \mathsf{set}$	intersection	
$\setminus$ (`\<setminus>`)	$'\alpha\ \mathsf{set} \Rightarrow {'\alpha}\ \mathsf{set} \Rightarrow {'\alpha}\ \mathsf{set}$	set difference	
$\subseteq$ (`\<subseteq>`)	$'\alpha\ \mathsf{set} \Rightarrow {'\alpha}\ \mathsf{set} \Rightarrow \mathsf{bool}$	subset	
Univ	$'\alpha\ \mathsf{set}$	all objects of type $'\alpha$	
Pair	$'\alpha \Rightarrow {'\beta} \Rightarrow {'\alpha} * {'\beta}$	$\mathsf{Pair}\, x\, y = (x, y)$	(x, y) denotes $\mathsf{Pair}\, x\, y$
fst	$'\alpha * {'\beta} \Rightarrow {'\alpha}$	$\mathsf{fst}\, (x, y) = x$	
snd	$'\alpha * {'\beta} \Rightarrow {'\beta}$	$\mathsf{snd}\, (x, y) = y$	
rtranclp	$('\alpha \Rightarrow {'\alpha} \Rightarrow \mathsf{bool}) \Rightarrow$ $'\alpha \Rightarrow {'\alpha} \Rightarrow \mathsf{bool}$	transitive closure $\mathsf{rtranclp}\, p\, z_0\, y = (z_0 = y) \vee$ $(\exists z_1, \ldots, z_n.\ z_n = y \wedge$ $\bigwedge_{i=0}^{n-1} (p\, z_i z_{i+1}))$	$(a, b) \in \{(x, y).f\}^*$ denotes $\mathsf{rtranclp}(\lambda xy.f)\, a\, b$
update	$('\alpha \Rightarrow {'\beta}) \Rightarrow {'\alpha} \Rightarrow {'\beta}$ $\Rightarrow {'\alpha} \Rightarrow {'\beta}$	$f(x := v)x = v$ $f(x := v)y = fy, \text{ for } y \neq x$	$f(x := v)$ denotes $\mathsf{update}\, f\, x\, v$
$<, \ \leq$ (`<=`)	$\mathsf{int} \Rightarrow \mathsf{int} \Rightarrow \mathsf{bool}$	ordering	$x > y$ denotes $y < x$
$+, -$	$\mathsf{int} \Rightarrow \mathsf{int} \Rightarrow \mathsf{int}$	plus, minus	
$*$	$\mathsf{int} \Rightarrow \mathsf{int} \Rightarrow \mathsf{int}$	multiplication	
div, mod	$\mathsf{int} \Rightarrow \mathsf{int} \Rightarrow \mathsf{int}$	division, modulo	
cardinality	$\mathsf{obj}\ \mathsf{set} \Rightarrow \mathsf{int}$	cardinality of a set	
comment	$'\alpha \Rightarrow \mathsf{bool} \Rightarrow \mathsf{bool}$	comment "*text*" $f = f$	

Figure 4-3: Logical constants and shorthands

As usual in model theory, our set-theoretic meta notation is no simpler than the language whose semantics we are giving, but hopefully helps clarify the meaning of the notation. In the meta notation, I represent functions as functional relations, that is, as particular sets of pairs. I give the usual set-theoretic semantics to set types $t\, \mathsf{set}$ and product types $t_1 * t_2$. These types could be encoded using function types, but I instead give direct semantics in the set-theoretic meta-notation, which should be equivalent and simpler.

Logical constants and shorthands. Figure 4-3 shows some of the logical constants and shorthands that we use in HOL. I do not aim to define all logical constants in terms of primitives (for such development, see [11, Chapters 4, 5] and the theory files of the Isabelle distribution). Instead, I just clarify the meaning of Jahob's HOL notation and give the model theoretic semantics to the less common symbols.

Following the notation in [201], constants typically have several representations. For example, the universal quantifier has an ASCII notation `ALL`, but also the mathematical notation $\forall$. Users can enter symbols such as $\forall$ using the `x-symbol` package for XEmacs

[256] and the Proof General environment [17], which, for example, displays special ASCII sequence `\<forall>` as the symbol $\forall$. These packages were developed for Isabelle, but are available to Jahob because it uses the same formula syntax and the same convention on writing formulas in quotes as Isabelle. This illustrates one benefit of adopting an existing notation.

HOL uses logical connectives of classical logic $\neg, \wedge, \vee, \rightarrow$ and quantifiers $\forall, \exists$. The equality on bool type serves as the logical equivalence.

The set comprehension symbol $\mathsf{Collect}$ maps a predicate into a set of objects satisfying this predicate; notation $\{x.Px\}$ gives a convenient notation for set comprehensions. Conversely, the membership operation $\in$ allows defining a predicate $\lambda x.x \in S$ for any set S. Symbols $\cap, \cup, \setminus$ represent standard set theoretic operations of union, intersection, and set difference. The $\subseteq$ symbol represents set inclusion, and Univ represents the universal set whose type is inferred from the context. As a shorthand, we write $\mathsf{Objects}$ to denote formula $\mathsf{Univ} :: \mathsf{obj}\,\mathsf{set}$ whose meaning is the set of all object individuals. The (x, y) notation denotes the pair of values x and y, whereas fst and snd denote projection functions for selecting the first and the second element of a pair.

The $\mathsf{rtranclp}$ symbol denotes reflexive transitive closure of a binary predicate and is a variation of the Isabelle's `rtrancl` operator of type $('\alpha * '\alpha)\,\mathsf{set} \Rightarrow ('\alpha * '\alpha)\,\mathsf{set}$. Formula $\mathsf{rtranclp}(\lambda xy.f)\,a\,b$ and its set-theoretic syntactic version $(a, b) \in \{(x, y).f\}^*$ are natural higher-order representations of the construct $(\mathsf{TC}_{xy}f)(a, b)$ used in transitive closure logics [127, Section 9.2].

The function update operator $f(x := v)$ maps a given function f into a new function that differs from f at a single point x. This operator often occurs in mathematical definitions; in verification conditions it represents field and array assignments.

We use standard operations and relations on the set of integers. The $\mathsf{cardinality}$ operator represents the cardinality of a finite set of objects. It differs from Isabelle's built-in card operator in being monomorphic and and in returning an integer as opposed to a natural number. We use it to represent constraints of Chapter 7.1. Moreover, in combination with set comprehensions it naturally represents counting quantifiers [205]; for example $\mathsf{cardinality}\,\{x.F\} = 1$ means that there exists exactly one object x satisfying the formula F.

In addition, Jahob's HOL allows a notation for comments in formulas using the operator $\mathsf{comment}$ "*text*" f, whose meaning is the same as the formula f, but which stores the annotation *text* within the formula. Jahob maintains these annotations when processing formulas and displays them in case a proof obligation fails, helping the user to identify the source of a failed proof obligation.

4.1.1 Rationale for Using Higher-Order Logic in Jahob

I decided to use HOL in Jahob (both as the surface specification language and as the language for communication between Jahob components) because HOL is a simple and expressive language that supports reasoning about sets and relations. The ability to reason about sets and relations is important because these concepts naturally describe data structure content and enable modular analysis in Jahob, separating the verification of data structure implementation from the verification of data structure uses. I next discuss some further aspects of using HOL as the specification language.

How expressive logic do we need? The expressive power of full higher-order logic is beyond the logics for which we presently know how to do non-interactive automated reasoning. However, if one combines different decidable logics in a logic that has a naturally defined syntax, the result is inevitably a very expressive and undecidable logic. It is therefore reasonable to adopt a logic without artificial limitations that would hamper further extensions of the framework. Having adopted an expressive language, the research goal then shifts from the question of developing a specification language to the algorithmic question of developing decision procedures for subsets of an expressive language, and developing combination techniques for different decision procedures. Currently, the combination of logics described in this dissertation does not require the full generality of HOL; what suffices is the second-order logic of objects combined with the first-order logic of integers.

Uses of set quantification. The logics in Chapter 6 and Chapter 7 allow set quantification. The presence of quantification over sets in HOL allows me to easily embed these logics into HOL. The explicit presence of sets is also a prerequisite for having the cardinality operator from Chapter 7.

Uses of lambda expressions. The presence of lambda expressions allows Jahob to specify definitions of functions that represent specification fields in programs. It also allows Jahob users to define shorthand constructs that Jahob eliminates by substitution and beta-reduction. Finally, lambda expressions enable encoding of set comprehensions. In combination with set algebra operations and abstract set variables, set comprehensions can naturally express properties that would otherwise require universal quantifiers and element-wise reasoning.

Interactive theorem provers as the semantic foundation. When choosing an expressive logic as a specification mechanism, it is important that the users of the system can relate to the meaning of the notation and bridge the gap between informal requirements and specifications. The appropriateness of different notations from this perspective is difficult to objectively quantify, which is reflected in different communities using specification languages with different foundations. However, expressive logical notations (just like general-purpose programming lanuages) are somewhat interchangeable, which is why many areas of mainstream mathematics do not depend on the particular axiomatization of set theory or the axiomatization of arithmetic. One practical consideration I took into account is that using directly a core logical notation without a rich predefined set of mathematical notions would require users to write more universally quantified axioms to axiomatize complex properties. In my experience this activity all too often leads to contradictory axioms that can make the results of verification worthless. When using an approach of interactive theorem provers, this axiomatization is part of the libraries of mathematical facts and its appropriateness is experimentally verified by a larger body of formalized mathematical knowledge.

Among the alternative choices for an expressive logic, I chose the classical higher-order logic as a language close to the language used in mainstream mathematics. My particular choice was inspired by the higher-order logic used in the Isabelle [200] interactive theorem prover, a system that is actively maintained and freely distributed in source code form. This choice made it easier to build the Isabelle interface described in Section 4.2. The choice of Isabelle's HOL as opposed to a different variant is not essential. Indeed, Charles Buillaguet has subsequently built (and successfully used) a Jahob interface to the Coq interactive theorem prover [33]. The fact that Coq uses a different underlying logical foundation than Isabelle did cause substantial difficulties in creating the Coq interface in Jahob.

4.2 Interface to an Interactive Theorem Prover

This section discusses the benefits of an interface to an interactive theorem prover in a system whose goal is to automatically prove formulas in an expressive logic. Our experience comes from the use of such an interface in Jahob and Hob [152] systems. A more detailed description of our experience with Isabelle in the context of Hob is in [267].

4.2.1 A Simple Interface

Jahob manipulates abstract syntax trees of formulas that correspond to those in Section 4.1. Jahob also contains a parser and a pretty printer for a subset of HOL formulas. These components support the concrete syntax of Isabelle formulas. A simple Isabelle interface therefore consists of pretty printing formulas into a file and invoking Isabelle. Jahob assumes a small Isabelle theory on top of Isabelle/HOL; this theory introduces `obj` as a new uninterpreted type and defines several shorthands.

In terms of the user-interface, the question is how to combine the interactive nature of Isabelle and the batch nature of a data structure verifier that generates proof obligations. We take the following approach:

1. Jahob first invokes Isabelle in batch mode, by generating an input file with the proof obligation and the default proof script (with a timeout that can be adjusted by the user). The default proof script invokes Isabelle's built in `auto` tactic. If the proof attempt succeeds, the formula is valid.

2. If the proof fails, Jahob saves it in an Isabelle file that contains a list of unproved lemmas. The user can then load the file into Isabelle and interactively prove the formula.

3. In subsequent invocations, Jahob loads the list of previously proved lemmas for the current verification task and recognizes formulas as true if they occur in the list.

4.2.2 A Priority Queue Example

One of the examples we have verified using theorem proving in [267] is a priority queue implemented as a binary heap stored in an array [67, Section 6.5]. Figure 4-4 shows the `insert` operation approximately corresponding to one of the operations verified in [267]. The postcondition of `insert` ensures that the set of (key,value)-pairs increases by the inserted element. I use the priority queue example to describe the Isabelle interface, even though the verification was originally done in the context of Hob, because the interface is essentially the same in both Hob and Jahob.

4.2.3 Formula Splitting

Our initial experience with the Isabelle interface showed that formulas passed to Isabelle can be large and consist of many independent subgoals. This would lead to interactive proofs that prove each of the subgoals in a single, long, unreadable proof script. In response to this, we implemented *formula splitting*, presented in in Figure 4-5, which takes a formula and generates a set of sequents. By a sequent I here mean a formula of the form $A_1 \wedge \ldots \wedge A_n \rightarrow G$. The splitting in Figure 4-5 has several desirable properties:

- the original formula is equivalent to the conjunction of the generated sequents;

```
class Element {
  public /*: claimedby PriorityQueue */ int key;
  public /*: claimedby PriorityQueue */ Object value;
}
class PriorityQueue {
  private static Element[] queue;
  public /*: readonly */ static int length;
  /*:
    public static ghost specvar init :: bool = "False";
    public static specvar content :: "(int * obj) set"
    vardefs "content ==
       {(k,v). EX j. 1 <= j & j <= length &
                (EX e. e = queue.[j] &
                 k = e..Element.key & v = e..Element.value)}";

    public static specvar maxlen :: int;
    vardefs "maxlen == queue..Array.length - 1";

    invariant "init --> 0 <= length &
                        length <= maxlen &
                        queue ~= null &
                        (ALL j. 0 < j & j <= length --> queue.[j] ~= null)";
  */

  public static void insert(int key, Object value)
  /*: requires "init & length < maxlen"
      modifies content, length
      ensures "content = old content Un {(key, value)} &
               length = old length + 1" */
  {
    length = length + 1;
    int i = length;
    while /*: invariant "1 <= i & i <= length &
                old content = {(k,v). EX j. 1 <= j & j <= length & j ~= i &
                                  (EX e. e = queue.[j] &
                                    k = e..Element.key & v = e..Element.value)} &
                (ALL j. 0 < j & j <= length --> queue.[j] ~= null)"; */
      (i > 1 && queue[i/2].key < key) {
        queue[i] = queue[i/2];
        i = i/2;
    }
    Element e = new Element();
    e.key = key;
    e.value = value;
    queue[i] = e;
  }
}
```

Figure 4-4: Insertion into Priority Queue

$$
\begin{array}{rcl}
A_1 \wedge \ldots \wedge A_n \rightarrow G_1 \wedge G_2 & \rightsquigarrow & A_1 \wedge \ldots \wedge A_n \rightarrow G_1,\ A_1 \wedge \ldots \wedge A_n \rightarrow G_2 \\
A_1 \wedge \ldots \wedge A_n \rightarrow (B_1 \wedge \ldots \wedge B_m \rightarrow G) & \rightsquigarrow & A_1 \wedge \ldots \wedge A_n \wedge B_1 \wedge \ldots \wedge B_m \rightarrow G \\
A_1 \wedge \ldots \wedge A_n \rightarrow \forall x.G & \rightsquigarrow & A_1 \wedge \ldots \wedge A_n \rightarrow G[x := x_{\mathsf{fresh}}]
\end{array}
$$

Figure 4-5: Formula splitting rules

- each sequent is smaller than the original formula;
- the number of generated sequents is linear in the size of the original formula (therefore, the maximal increase in total formula size is quadratic);
- even if Isabelle's `auto` cannot prove the original formula, it can often prove some of the generated sequents, simplifying the remaining manual task.

The overall effect of splitting is to isolate and separate non-trivial cases in the proof obligation. As an illustration, when we verified the priority queue insertion procedure [267] corresponding to Figure 4-4, the result were 11 proof obligations. Isabelle proved 5 of those automatically, and we proved 6 interactively and stored them in a file as lemmas (the average proof length was 14 lines). Subsequent verification of the procedure loads the stored lemmas and succeeds without user interaction.

Note that, although the use of splitting described in this section is useful, the idea of splitting proved even more useful in the context of combining different reasoning techniques, because Jahob can use different techniques to prove different sequents. Therefore, splitting acts as a rudimentary backtracking-free form of propositional solving whose complete versions appear in lazy approaches for satisfiability checking [94].

4.2.4 Lemma Matching

When a data structure changes, or when the developer wishes to strengthen its invariants during verification, the resulting proof obligations change. Given the amount of effort involved in interactive theorem proving, it is desirable to reuse as many of the previously proved lemmas as possible.

Consider the verification of an abstract data type, such as the priority queue example in Figure 4-4. In general, to show that a state transformation t preserves an invariant I, Jahob generates a proof obligation of the form

$$I_1 \rightarrow \mathsf{wlp}(t, I_1) \tag{4.1}$$

where the notation $\mathsf{wlp}(t_0, I_0)$ represents the weakest liberal precondition of a predicate I_0 with respect to a state transformation t_0. While the representation invariants stated in Figure 4-4 are enough to verify the postcondition of the `insert` operation, they are not enough to verify the postcondition of the `extractMax` operation in Figure 4-6. The `extractMax` operation requires an additional property saying that distinct indices store distinct objects. Adding this property as an invariant requires showing again that `insert` preserves all class invariants. Effectively, it is necessary to show a proof obligation

$$I_1 \wedge I_2 \rightarrow \mathsf{wlp}(t, I_1 \wedge I_2)$$

```
/*: invariant "init -->
            (ALL i j. 0 < i & i < j & j <= length --> queue.[i] ~= queue.[j])"; */

  public static Object extractMax()
  /*: requires "init & length > 0"
      modifies content, length
      ensures "length = old length - 1 &
                  (EX key. (key,result) : content &
                      content = old content \<setminus> {(key,result)})";
    */
  {
    if (isEmpty()) return null; // queue is empty
    Object res = queue[1].value;
    queue[1] = queue[length];
    length = length - 1;
    heapify(1);   // restore heap property, preserving set of elements
    return res;
  }
```

Figure 4-6: Extraction of Maximal Element from a Priority Queue, with an Example of an Additional Invariant

By conjunctivity of wlp [20], this condition reduces to proving two proof obligations

$$I_1 \land I_2 \quad \rightarrow \quad \mathsf{wlp}(t, I_1) \tag{4.2}$$
$$I_1 \land I_2 \quad \rightarrow \quad \mathsf{wlp}(t, I_2) \tag{4.3}$$

The splitting from Section 4.2.3 makes sure that Jahob indeed generates these as separate proof obligations. Proving the condition (4.3) requires a new proof. However, condition (4.2) is a consequence of the condition (4.1) and we would like to avoid repeating any interactive proof for it and instead use the proof for a stronger lemma that was saved in a file.

A system that checks for proved lemmas by simply comparing syntax trees will fail to reuse a proof of (4.1) for (4.2), because they are distinct formulas. It is instead necessary to take into account formula semantics. On the other hand, giving (4.1) as a lemma to Isabelle when proving (4.2) does not guarantee that Isabelle will prove (4.2); in our experience this approach fails for proof obligations that have a large number of assumptions.

We therefore implemented our own lemma matching procedure that compares syntax trees while allowing assumptions to be strengthened and conclusions to be weakened. This technique enabled us to reuse proofs in the case of strengthening invariants, as well as for several other simple changes to proof obligations, such as changing the order of conjuncts and disjuncts.

4.2.5 Summary of Benefits

We have found that the main benefits of an interface to an interactive prover are

- the ability prove difficult formulas that cannot be proved using more automated techniques;
- debugging the system itself while developing more automated techniques; and
- feedback from the proof process about the reasons why the formula is valid or invalid.

Interactively proving difficult formulas. The formulas that we encounter in data structure verification appear to have the property that if they are valid, they they have reasonably short proofs. One explanation of this observation is that data structure verification formalizes informal correctness arguments that programmers use on an everyday basis and therefore do not require intricate mathematical reasoning. As a result, the valid formulas that are of interest in data structure verification are in practice provable interactively given the right proof hints (independently of the well-known incompleteness of axiomatizations of logical systems [108, 59]). Proving a proof obligation is therefore a matter of investing enough effort to describe the proof steps in an interactive theorem prover. This approach is becoming easier with the increase of automation in interactive theorem provers and it is feasible to use it for important problems [171, 15, 138], but it can still be time consuming. The advantage of this approach in the context of Jahob is that the user only needs to use the Isabelle interface to prove certain problematic proof obligations that were not provable by other methods. In comparison, a user of an automated system that does not have a fall-back on an interactive theorem prover will sometimes need to substantially weaken the property of interest that is ultimately proved, or otherwise admit potential "holes in the proof" that are not justified within the system itself. Of course, Jahob users can still choose to leave such holes in their verification task using the `assume` statements. Jahob allows unrestricted use of `assume` statements but emits a warning when it encounters them. It is up to the Jahob user to make the trade off between the verification effort and the correctness assurance that they need.

Debugging benefits. In addition to proving difficult formulas that are directly relevant to the verification task, our Isabelle interface was useful during the development of the verification system itself. Initially we used this interface to make sure that the formulas that we generate are accepted by Isabelle (both in terms of the syntax and in terms of type correctness). More importantly, as we implement new techniques for deciding formulas arising in data structure verification, we sometimes encounter formulas for which the result of the current implementation does not match our understanding of the system or of the example being verified. Being able to examine the formula and try to prove it in Isabelle allows us to discover whether the formula is truly valid or invalid, as well as to find the reason why this is so. We can then use this insight to eliminate any implementation errors (in the case of unsoundness), or to improve the precision of the automated technique to make the formula automatically provable (in case of a formula that was not proved automatically).

Feedback from the proof process. An important aspect of an interactive theorem prover in both of the previous applications is the feedback obtained from the interactive theorem proving process. This feedback allows the user or the developer to identify reasons why the proof fails or succeeds. Because the proving process is under the user's control, the user can trace this feedback to the reason why a formula is valid or invalid. For example, it may be possible to identify the source of contradiction in assumptions, or a missing assumption in the formula. This form of feedback corresponds to the insights one gains in manually trying to prove the correctness of a system, and is orthogonal to presenting feedback by automatically searching for counterexamples in finite scopes [132, 131], which is in principle available in Jahob through the TPTP interface and the Paradox model finder [56], but currently without guarantees that the models found are sound counterexamples. (See [150] for results on soundness of counterexamples over infinite domains.)

4.2.6 Discussion

I described several simple yet useful techniques in Jahob's interface to an interactive theorem prover. Our experience supports the idea that expressive specification languages have their place in data structure verification. Moreover, this section argues that interactive theorem provers that can handle the entire language are a useful component of a verification system. By verifying partial correctness properties (e.g. by not verifying the sortedness of priority queue), this section also illustrates that choosing properties carefully can reduce the burden of interactive theorem proving. In the rest of this dissertation I show how automated techniques can often eliminate the need for interactive theorem proving. Interactive theorem proving then remains a fall-back employed only for the most difficult properties.

A natural question to ask is whether these automated techniques should be 1) incorporated into a data structure verification system, or 2) incorporated into the interactive theorem prover as in [188, 31]. I choose the first approach because it gives a lot of flexibility and avoids the need to understand an existing implementation of an interactive theorem prover. However, the techniques that I describe may also be appropriate for integration into an interactive theorem prover.

4.3 Approximation of Higher-Order Logic Formulas

This section describes a general technique for approximating higher-order logic formulas to obtain formulas in a simpler fragment of logic. The goal is to check the validity of such simpler formulas with more automation than with interactive theorem proving. The basic idea is to approximate the validity of formulas with the validity of stronger formulas in a recursive way, taking into account the monotonicity of logical operations $\land, \lor, \exists, \forall$, as well as the anti-monotonicity of $\neg$. The basis of this recursion is the approximation of formula literals by simpler formulas, and depends on the target subset of formulas. The approximation replaces the formula constructs that are entirely outside the scope of the target fragment by false in a positive context and by true in a negative context, resulting in a sound approximation scheme that is applicable to arbitrary higher-order logic formulas. To increase the precision of the approximation on formulas with nested subterms, the transformation can first flatten such formulas, generating multiple simpler literals each of which can be approximated independently.

4.3.1 Approximation Scheme for HOL Formulas

The process of approximating formulas by simpler ones resembles abstract interpretation [69]. The approximation should preserve soundness for validity, so it replaces formulas by stronger ones. The entailment relation defines a preorder on formulas, which is a partial order modulo formula equivalence. Let HOL denote the set of our higher-order logic formulas and C denote the target class of more tractable formulas. We define two functions

$$\begin{aligned} \gamma &: \mathrm{C} \to \mathrm{HOL} \\ \alpha^0 &: \mathrm{HOL} \to \mathrm{C} \end{aligned}$$

such that

$$\gamma(\alpha^0(F)) \models F \tag{4.4}$$

$$
\begin{array}{rcl}
\alpha & : & \{0,1\} \times F \to \mathrm{C} \\
\alpha^p(f_1 \wedge f_2) & \equiv & \alpha^p(f_1) \wedge \alpha^p(f_2) \\
\alpha^p(f_1 \vee f_2) & \equiv & \alpha^p(f_1) \vee \alpha^p(f_2) \\
\alpha^p(\neg f) & \equiv & \neg \alpha^{\overline{p}}(f) \\
\alpha^p(f_1 \rightarrow f_2) & \equiv & \alpha^{\overline{p}}(f_1) \rightarrow \alpha^p(f_2) \\
\alpha^p(\forall x.f) & \equiv & \forall x.\alpha^p(f) \\
\alpha^p(\exists x.f) & \equiv & \exists x.\alpha^p(f) \\
\alpha^0(f) & \equiv & \mathsf{false},\ \text{if none of the previous cases apply} \\
\alpha^1(f) & \equiv & \mathsf{true},\ \text{if none of the previous cases apply}
\end{array}
$$

Figure 4-7: General Approximation Scheme

We can think of γ as providing the semantics to formulas in C in terms of HOL. The actual approximation is the function α^0, which approximates HOL formulas with simpler ones (for example, with first-order formulas or with formulas in a decidable logic).

The definition of α^0 is specific to the class C of target formulas, but generally proceeds by recursion on the syntax tree of the formula, using the monotonicity and anti-monotonicity of logical operations to ensure the condition (4.4). Figure 4-7 sketches a typical way of defining the approximation function α^0. To handle negation, it is convenient to introduce a dual α^1 such that $F \models \gamma(\alpha^1(F))$, for example by having $\alpha^1(\neg F) = \alpha^0(F)$ and $\alpha^0(\neg F) = \alpha^1(F)$. Figure 4-7 uses notation $\overline{p}$ with the meaning $\overline{0} = 1$ and $\overline{1} = 0$. (An alternative to keeping track of polarity is to use the negation-normal form of formulas in Figure 4-8.) The translation α^1 is likely to be useful in itself for using model finders [131, 56] to find counterexamples to formulas because it preserves the presence of certain models [149]. A simple way to define α^0 is to first construct the embedding γ and then construct α^0 by inverting γ when possible and returning a conservative result false otherwise (for a formula in negation-normal form). To make this simple approach more robust, we first apply rewrite rules that transform formulas into a form where the translation is more effective. Section 4.3.2 presents rewrite rules that we found useful. Chapter 5 presents a series of specific rewrite rules that facilitate transformation into first-order logic.

In Figure 4-7, the translation of a quantification $Qx :: t$ over a variable x of type t applies to the case where the target class C also supports quantification over the elements of the type t. If the class C does not permit quantification over x, the approximation can attempt to skolemize x or eliminate x using substitution. If everything fails, the translation approximates the quantified formula with false or true according to the polarity. A potentially more precise approach is to approximate those atomic formulas in the scope of the quantifier that contain x with an expression not containing x, eliminating the need to quantify over x.

Note that the approximation α^0 can expect the argument to contain a specific top-level assumption, such as the treeness of the structure along certain fields, and return a trivial result if this is not the case. Chapter 6 presents an example of such approximation.

In general, it is useful to apply multiple approximations $\alpha_1^0, \ldots, \alpha_n^0$ to the same formula. If any of the approximations results in a valid formula, the original formula is valid. The splitting in Section 4.2.3 helps this process succeed by allowing Jahob to prove each sequent

```
**proc** NegationNormalForm($G$ : formula with connectives $\land, \lor, \neg$):
  apply the following rewrite rules:
      $\neg(\forall x.G) \rightarrow \exists x.\neg G$
      $\neg(\exists x.G) \rightarrow \forall x.\neg G$
      $\neg\neg G \rightarrow G$
      $\neg(G_1 \land G_2) \rightarrow (\neg G_1) \lor (\neg G_2)$
      $\neg(G_1 \lor G_2) \rightarrow (\neg G_1) \land (\neg G_2)$
```

Figure 4-8: Negation Normal Form

using a different approximation (see Section 4.4 for further discussion).

4.3.2 Preprocessing Transformations of HOL Formulas

We next describe several transformations of HOL formulas that we found generally useful to perform before applying formula approximations. Note that these transformations are typically validity-preserving and are not strict approximations in the sense of losing information. However, they often improve the precision of subsequent approximations. I describe the approximations themselves in subsequent chapters because they are specific to the target class of formulas, see Section 5.2, Section 6.3, and Section 7.6. Just like formula approximations, some preprocessing transformations are specific to the target class of formulas and I describe them in the corresponding chapters (most notably, Section 5.2 contains additional equivalence-preserving transformations that are useful for first-order logic approximation).

Definition substitution and beta reduction. When a Jahob program contains a specification variable with the definition `v==\<lambda> x. e`, the resulting proof obligation will often contain assumptions of the form `v'==\<lambda> x. e'` where `e'` is a result of applying some substitutions to `e`. Jahob typically eliminates such assumptions by substituting the variables with the the right-hand side of the equality and then performing beta reduction, as in standard lambda calculus.

Negation normal form. To avoid any ambiguity, Figure 4-8 presents rules for transforming a formula into negation-normal form. In negation-normal form, all negations apply to literals as opposed to more general formulas. The transformation in Figure 4-8 assumes that implication and equivalence are eliminated from the formula, which can be done in a standard way, by replacing the equivalence $p \leftrightarrow q$ (written $p = q$ for p and q of type bool) with $p \rightarrow q \land q \rightarrow p$ and by replacing $p \rightarrow q$ with $p \lor \neg q$.

Flattening. A useful transformation during formula approximation is flattening of formulas with nested terms, which replaces complex atomic formulas such as $P(f\,x)$ by multiple atomic formulas such as $y = f\,x$ and $P\,y$ where y is a fresh variable. The translation can choose whether to quantify y existentially, generating $\exists y.\, y = f\,x \;\land\; P\,y$ or universally, generating $\forall y.\; y = f\,x \rightarrow P\,y$. Flattening has several benefits:

- Flattening avoids an exponential explosion in rewrite rules that duplicate subterms. Such rewrite rules arise in the translation of conditional expressions and function updates.
- Flattening enables the representation of function applications by predicates or by binary relations that approximate them, as in Chapter 6.

```
preprocess : HOL → HOL              (preprocessing transformations)
[(α_1, d_1), ..., (α_m, d_m)]       (list of prover instances)
  α_j : HOL → C_j                   (approximation function)
  d_j : C_i → bool                  (prover for C_j formulas)

proc validFormula($F$ : HOL) : bool =
  let $[F_1, \ldots, F_n]$ = splitIntoSequents($F$)      (see Figure 4-5)
  for $i = 1$ to $n$ do                                  (prove conjunction of all $F_i$)
    if not validSequent($F_i$) then return false
  endfor
  return true                                            (all sequents proved)

proc validSequent($F_i$ : HOL) : bool =
  let $F_i'$ = preprocess($F_i$)
  for $j = 1$ to $m$ do                      (try disjunction of all provers)
    let $F_i'' = \alpha_j(F_i')$             (approximate formula)
    if $d_j(F_i'')$ then return true         (prover $j$ proved sequent $F_i$)
  endfor
  return false                               (all provers failed)
```

Figure 4-9: Jahob's algorithm for combining reasoning procedures (provers)

- Flattening localizes the effect of conservative approximations. For example, when translating a predicate $P(f(x))$ where f is a construct not supported in C (or a construct containing a quantified variable that ranges over an unsupported domain), it is more precise to flatten it into the formula $\forall y.\ y = f\,x \ \rightarrow\ P\,y$ and then approximate $y = f\,x$ to true, yielding $\forall y.\ P\,y$, than to replace the entire formula $P(f\,x)$ by false.

4.4 Summary and Discussion of the Combination Technique

In this section I summarize Jahob's algorithm for proving the validity of HOL formulas by combining multiple decision procedures and theorem provers. I discuss properties of this algorithm, showing how the structure of verification conditions, as well as user-supplied lemmas and specification variables increase its effectiveness.

4.4.1 Jahob's Combination Algorithm

Figure 4-9 shows a high-level description of Jahob's algorithm for proving HOL formulas. In addition to the preprocessing function, the parameter of the algorithm is a sequence of *prover instances*. A prover instance is a pair of approximation function, which maps HOL formulas into simpler formulas, and a function that accepts a simpler formula and attempts to prove that it is valid. In actual Jahob implementation, a Jahob user specifies the list of prover instances on a command line by instantiating the available translations and interfaces to decision procedures and theorem provers. The instantiation consists in specifying parameters such as time outs and the precision of the approximation (see Section 5.5 for specific examples of parameters). The algorithm is given by function validFormula, which first invokes splitting (Figure 4-5) to generate a list of sequents, and then attempts to prove each sequent in turn using the validSequent function. The validSequent function first applies formula preprocessing and then attempts to prove the sequent using each prover instance until one of the attempts succeeds or they all fail. An attempt number j consists in first approximating the preprocessed sequent using α_j to obtain a formula in the simpler class $\mathrm{C_j}$, and then attempting to prove the resulting simpler formula using the given function d_j. The execution of d_j may involve invoking an external theorem prover or a decision procedure.

At the time of writing, Jahob has the following provers (see also Figure 3-5):

- a simple and fast built-in prover;
- MONA [143] decision procedure for monadic second-order logic, with field constraint analysis (Chapter 6);
- CVC Lite [30] decision procedure for a combination of decidable theories (using the SMT-LIB interface [216]);
- SPASS [248] resolution-based prover;
- E [228] and Vampire [244] resolution based provers (using the TPTP interface);
- Boolean Algebra with Presburger arithmetic solver (Chapter 7), with CVC Lite (and, in the past, Omega) as the Presburger arithmetic back-end;
- Isabelle [200] interactive prover;
- Coq [33] interactive prover.

We have found instances where we needed to combine several provers to prove a formula, using for example, MONA along with a first-order prover or CVC Lite, or using E, SPASS, and Isabelle together to prove all of the generated sequents.

Jahob always invokes its built-in prover before any others. The built-in prover is very useful, despite its simplicity. In addition to establishing many sequents that can be shown syntactically and that arise from, for example, null pointer checks, it also has the property of successfully proving formulas such as $F \rightarrow F$ for arbitrarily complex F. Such trivial tautology instances arise when proving preservation of complex conditions or when giving names to assumptions using `noteThat` statements, but are often obscured by skolemization and the translation, preventing the remaining sophisticated tools to establish them in a reasonable amount of time.

4.4.2 Verifying Independent Invariants

I next illustrate that Jahob's combination algorithm effectively deals with verifying multiple independent invariants that can be translated to languages of different provers.

Using the notation of Figure 4-9, consider two provers (α_1, d_1) and (α_2, d_2) for two specialized logics $\mathrm{C_1}, \mathrm{C_2} \subseteq \mathrm{HOL}$ with $\alpha_i(F) = F$ for $F \in \mathrm{C_i}$ for $i = 1, 2$. Consider the verification of a class with three invariants I_1, I_2, and I_{12}, where $I_1 \in \mathrm{C_1}$, $I_2 \in \mathrm{C_2}$ and $I_{12} \in \mathrm{C_1} \cap \mathrm{C_2}$. The preservation of these three invariants under a sequence of commands c is given by the condition

$$\begin{array}{l}\texttt{assume}\ I_1 \land I_{12} \land I_2;\\ c;\\ \texttt{assert}\ I_1 \land I_{12} \land I_2;\end{array}$$

Taking into account that wlp distributes over conjunctions, a verification condition F for the preservation of invariants is then of the form:

$$I_1 \land I_{12} \land I_2 \rightarrow (\mathsf{wlp}(I_1) \land \mathsf{wlp}(I_{12}) \land \mathsf{wlp}(I_2))$$

Splitting invoked in Figure 4-9 will split this verification condition into at least three conjuncts because the right-hand side of the implication has at least three conjuncts. Suppose that $\mathsf{splitIntoSequents}(F) = [F_1, F_{12}, F_2]$ where

$$\begin{array}{rcl} F_1 & = & I_1 \land I_{12} \land I_2 \rightarrow \mathsf{wlp}(I_1) \\ F_{12} & = & I_1 \land I_{12} \land I_2 \rightarrow \mathsf{wlp}(I_{12}) \\ F_2 & = & I_1 \land I_{12} \land I_2 \rightarrow \mathsf{wlp}(I_2) \end{array}$$

Suppose further that the classes of formulas C_1, C_2, $\mathrm{C}_1 \cap \mathrm{C}_2$ are closed under wlp. We can then reasonably expect the following results of approximation:

$$\begin{array}{rcccl} \alpha_1(F_1) & = & I_1 \land I_{12} \land \mathsf{true} \rightarrow \mathsf{wlp}(I_1) & = & I_1 \land I_{12} \rightarrow \mathsf{wlp}(I_1) \\ \alpha_2(F_2) & = & \mathsf{true} \land I_{12} \land I_2 \rightarrow \mathsf{wlp}(I_2) & = & I_{12} \land I_2 \rightarrow \mathsf{wlp}(I_2) \\ \alpha_1(F_{12}) & = & I_1 \land I_{12} \land \mathsf{true} \rightarrow \mathsf{wlp}(I_{12}) & = & I_1 \land I_{12} \rightarrow \mathsf{wlp}(I_{12}) \\ \alpha_2(F_{12}) & = & \mathsf{true} \land I_{12} \land I_2 \rightarrow \mathsf{wlp}(I_{12}) & = & I_{12} \land I_2 \rightarrow \mathsf{wlp}(I_{12}) \end{array}$$

Note that the approximations α_i in this case essentially act as filters for assumptions that are outside the scope of the prover. For example, $\alpha^1(I_2) = \mathsf{true}$ because $I_2 \notin \mathrm{C}_1$.

We observe that, if the preservation of invariants is independent, the prover p_1 will succesfully prove the preservation of I_1 and the prover p_2 will succesfully prove the preservation of I_2. The preservation of I_{12} will be proved by whichever prover is ran first. This example shows that Jahob's combination technique is precise enough to support the use of multiple specialized analyses, generalizing the basic approach of Hob [152], suggesting that if an independent analysis with an annotation language C_1 would succeed to prove I_1 and an independent analysis with annotation language I_2 would succeed in analyzing the procedure, then Jahob's approach would succeed in showing that both I_1 and I_2 are preserved. The invariant I_{12} illustrates the usefulness of a common specification language: if two invariants have any conjuncts in common, they can be factored out into I_{12} and specified only once. In contrast, a user of analysis that require multiple specification formalisms would need to state the common fact I_{12} twice to make it understood by each of the analyses.

4.4.3 Using Annotations to Aid the Combination Algorithm

We have seen that the structure of verification conditions makes Jahob's simple combination algorithm useful for proving the preservation of independent properties. I next illustrate that user annotations can further increase the effectiveness of the combination algorithm.

Providing Lemmas using `noteThat` Statements

I first show how introducing `noteThat` statements corresponds to introducing lemmas while proving a verification condition. Consider a procedure whose body is a deterministic command b and whose verification condition is $p \rightarrow \mathsf{wlp}(c, q)$ where p is precondition and q postcondition. Then inserting a statement `noteThat` f as the last statement of the procedure results in a different verification condition, of the form $p \rightarrow \mathsf{wlp}(c, f \land (f \rightarrow q))$. For a deterministic command c, this verification condition is identical to $p \rightarrow (\mathsf{wlp}(c, f) \land (\mathsf{wlp}(c, f) \rightarrow \mathsf{wlp}(c, q)))$. Our splitting from Section 4.2.3 therefore transforms such verification condition into two sequents:

$$\begin{array}{l} p \rightarrow \mathsf{wlp}(c, f) \\ p \land \mathsf{wlp}(c, f) \rightarrow \mathsf{wlp}(c, q) \end{array}$$

Therefore, a `noteThat` f statement results in introducing a lemma $\mathsf{wlp}(c, f)$ in proving $\mathsf{wlp}(c, q)$ from p. We have found that supplying such lemmas during verification is a natural way of guiding a complex verification task.

Case Analysis Using Lemmas

In particular, users can introduce lemmas that direct Jahob to do case analysis. Consider a code fragment

```
assume F;
assert G;
```

Here F is a complex formula with many conjuncts and `assume` F represents the effect of analyzing a complex sequence of statements and annotations. To verifying the above piece of code, Jahob would generate a proof obligation of the form $F \rightarrow G$. Suppose that such formula is difficult to prove directly, but becomes eaiser by doing case analysis with two cases C_1 and C_2. The following set of annotations will then Jahob establish the goal.

```
assume F;
noteThat firstLemma:  (C1 → G);
noteThat secondLemma: (C2 → G);
assert G from firstLemma, secondLemma;
```

The first `noteThat` statements establishes the first case, the second `noteThat` statement establishes the second case, and the final `assert` statement proves the conclusion using the two cases.

4.4.4 Lemmas about Sets

We next consider lemmas that belong to the class of formulas that we denote QFBA, and which contains Quantifier-Free Boolean Algebra expressions denoting relationships between sets of objects. QFBA can express relationships between sets that denote contents of data structures and parts of data structures, including the individual objects. Lemmas in QFBA language are interesting because QFBA can be precisely translated to the language of the most effective provers that Jahob relies on: 1) it can be translated to first-order logic (Chapter 5) by representing sets as unary predicates and set operations using universal quantifiers; 2) it can be translated to monadic second-order logic of trees (Chapter 6) because this logic directly supports arbitrary operations on sets; and 3) it is a fragment of Boolean Algebra with Presburger Arithmetic (Chapter 7). As a result, QFBA lemmas can be used to exchange information between all these reasoning procedures.

Users can introduce specification variables to help state QFBA lemmas. Consider an implementation of a linked list whose first node is referenced by a global variable root, whose nodes are liked using the next field and whose data elements are stored in the data field. Consider the task of verifying that a code fragment c

```
List n1 = new List();
Object d1 = new Object();
n1.next = root;
n1.data = d1;
root = n1;
size = size + 1;
```

that inserts a fresh object into a linked list preserves an invariant I given by

$$\mathsf{size} = \mathsf{cardinality}\{x.\ \exists n.\ x = \mathsf{data}\, n \land (\mathsf{root}, n) \in \{(u,v).\mathsf{next}\, u = v\}^* \land n \neq \mathsf{null}\} \tag{4.5}$$

which states that an integer field size is equal to the number of distinct elements stored in the list. We can represent this verification task using the sequence `assume` I; c; `assert` I, resulting in a verification condition of the form $B \land I \rightarrow \mathsf{wlp}(I, c)$ where B is a formula describing some additional properties of program state such as the fact that the data structure is an acyclic singly linked list with `root` variable pointing to its first element. This verification condition is not expressible in first-order logic because of the presence of transitive closure operator $*$, it is not expressible in monadic second-order logic over trees because the list may store duplicates (so it may be the case that $\mathsf{data}\, n_1 = \mathsf{data}\, n_2$ for $n_1 \neq n_2$), and it is not expressible in Boolean Algebra with Presburger Arithmetic because it contains relation symbols such as data and next. Suppose, however, that the user introduces two set-valued specification variables nodes and content of type obj set, containing the set of linked nodes of the list and the set of the stored elements of the list. The invariant (4.5) then reduces to $\mathsf{size} = \mathsf{cardinality}\,\mathsf{content}$. The following additional invariants define the new set variables

$$\begin{aligned}\mathsf{nodesDef}:\quad \mathsf{nodes} &= \{n.\ (\mathsf{root}, n) \in \{(u,v).\mathsf{next}\, u = v\}^* \land n \neq \mathsf{null}\}\\ \mathsf{contentDef}:\quad \mathsf{content} &= \{x.\ \exists n.\ x = \mathsf{data}\, n \land n \in \mathsf{nodes}\}\end{aligned}$$

The user can then specify changes to nodes and content as well as a lemma about the freshness of `d1`, obtaining the following sequence of statements.

$$\begin{array}{l} c;\\ \texttt{noteThat d1} \notin \mathsf{content};\\ \mathsf{nodes} := \{\texttt{n1}\} \cup \mathsf{nodes};\\ \mathsf{content} := \{\texttt{d1}\} \cup \mathsf{content}; \end{array}$$

The assignments to specification variables essentially express lemmas $\mathsf{nodes} = \{\texttt{n1}\} \cup \mathsf{old\ nodes}$ and $\mathsf{content} = \{\texttt{d1}\} \cup \mathsf{old\ content}$ describing how the sets change in response to the changes to the data structure, and they can be seen as introducing variables for subformulas of the original verification condition. After the application of splitting, the result are proof obligations of the following form:

$$\begin{array}{l} (B \land \ldots) \rightarrow \mathsf{wlp}(c, \texttt{d1} \notin \mathsf{content})\\ (B \land \mathsf{nodesDef} \land \ldots) \rightarrow \mathsf{wlp}(c, \mathsf{nodesDef})\\ (B \land \mathsf{contentDef} \land \ldots) \rightarrow \mathsf{wlp}(c, \mathsf{contentDef})\\ (B \land \mathsf{size} = \mathsf{cardinality}\,\mathsf{content}) \rightarrow \mathsf{wlp}(c, \mathsf{size} = \mathsf{cardinality}\,\mathsf{content}) \end{array}$$

The first proof obligation is easy and is proved using a first-order prover, the second is proved using MONA decision procedure, the third one again using a first-order prover, and the final one using BAPA decision procedure.

4.4.5 Comparison to Nelson-Oppen Combination Technique

I next illustrate how our approximation technique along with user-directed case analysis and flattening proves formulas that can be proved by Nelson-Oppen style [197, 195] combination of decision procedures.

Assume that we have available a decision procedure for Presburger arithmetic, with an approximation function α_{PA} that takes a formula and generates a stronger formula in Presburger arithmetic. Assume similarly that we have a decision procedure for uninterpreted function symbols, whose approximation function is α_{UF}. These two languages share only the equality symbol, which means that it suffices to communicate between them equalities and inequalities between the shared variables [237].

Consider the process of proving the validity of the formula F given by

$$a < b+1 \;\wedge\; b < a+1 \;\wedge\; a_1 = f(a) \;\wedge\; b_1 = f(b) \;\rightarrow\; a_1 = b_1$$

If we directly approximate formulas we obtain

$$\begin{aligned}\alpha_{\text{PA}}(F) &= a < b+1 \;\wedge\; b < a+1 \;\rightarrow\; a_1 = b_1 \\ \alpha_{\text{UF}}(F) &= a_1 = f(a) \;\wedge\; b_1 = f(b) \;\rightarrow\; a_1 = b_1\end{aligned}$$

None of the formulas $\alpha_{\text{PA}}(F)$ or $\alpha_{\text{UF}}(F)$ are valid, so the combination would fail to prove F. If, on the other hand, the user uses a `noteThat` statement to introduce case analysis on the formula G given by $a = b$, we obtain a conjunction of formulas F_1 and F_2:

$$\begin{aligned}F_1 &\equiv a < b+1 \;\wedge\; b < a+1 \;\wedge\; a_1 = f(a) \;\wedge\; b_1 = f(b) \;\wedge\; a = b \rightarrow\; a_1 = b_1 \\ F_2 &\equiv a < b+1 \;\wedge\; b < a+1 \;\wedge\; a_1 = f(a) \;\wedge\; b_1 = f(b) \;\wedge\; a \neq b \rightarrow\; a_1 = b_1\end{aligned}$$

When we perform approximation, we obtain the results given by the following table

	F_1	F_2
α_{PA}	invalid	valid
α_{UF}	valid	invalid

Namely, note that

$$\begin{aligned}\alpha_{\text{PA}}(F_1) &\equiv a < b+1 \;\wedge\; b < a+1 \;\wedge\; a = b \rightarrow\; a_1 = b_1 \\ \alpha_{\text{UF}}(F_2) &\equiv a_1 = f(a) \;\wedge\; b_1 = f(b) \;\wedge\; a \neq b \rightarrow\; a_1 = b_1\end{aligned}$$

Therefore, each conjunct is proved by one decision procedure.

Assume that α_{PA} and α_{UF} distribute through conjunction ($\alpha_{\text{PA}}(H_1 \wedge H_2) = \alpha_{\text{PA}}(H_1) \wedge \alpha_{\text{PA}}(H_2)$ and similarly for α_{UF}) and, because C is known to both decision procedures, that $\alpha_{\text{PA}}(C) = \alpha_{\text{UF}}(C) = C$. Let $[H]$ denote the validity of a formula H. We have then replaced the approximation

$$[\alpha_{\text{PA}}(F)] \;\vee\; [\alpha_{\text{UF}}(F)]$$

of the statement $[F]$ with a stronger approximation

$$\begin{aligned}&([\alpha_{\text{PA}}(F) \vee C] \vee [\alpha_{\text{UF}}(F) \vee C]) \;\wedge \\ &([\alpha_{\text{PA}}(F) \vee \neg C] \vee [\alpha_{\text{UF}}(F) \vee \neg C])\end{aligned}$$

Generalizing from this example, observe that the case analysis triggered by user lemmas and splitting forces exploration of the arrangements needed to prove a valid formula in Nelson-Oppen combination. So, if a Nelson-Oppen combination can prove a formula, there exist `noteThat` statements that enable Jahob's combination technique to prove this formula as well.

Trade-offs in combining expressive logics. By comparing Jahob's combination method with Nelson-Oppen combination technique, we conclude that Jahob's method is simpler and does not attempt to guess the arrangement for shared symbols of formulas. Instead, it relies on the structure of the underlying formulas and on user annotations to provide the arrangements. Note however, that Jahob's technique addresses the combination of rich logics that (as Section 4.4.4 explains) share set algebra expressions as opposed to sharing only equalities (as the traditional Nelson-Oppen decision procedures). A complete decision procedure in the presence of shared set algebra expressions would need to non-deterministically guess an exponentially large set of relationships between sets of objects (see, for example, [266]), leading to a doubly exponential process. Therefore, it is likely that the automation of the process of guessing shared formulas should take into account the nature of formulas considered. Thomas Wies implemented one such approach in Jahob's Bohne module [254, 251], but its description is outside of the scope of this dissertation. In summary, Jahob's approach I describe is simple, sound, and generally applicable, but does not attempt to achieve completeness in an automated way.

4.5 Related Work

I survey some of the related work in the area of mechanized expressive logics and in the area of combination techniques.

Mechanized expressive logics. Jahob's notation is based on Isabelle/HOL [200, 207]. I decided to use Isabelle/HOL because I consider its notation to be a natural formalization of mainstream mathematics and find the tool to be open and easily accessible. Other similar systems might have worked just as well, especially given the fact that Jahob's automation does not rely on Isabelle implementation internals. Other popular interactive theorem provers based on classical higher-order logic are the HOL system [110], TPS [10], and PVS [204] (which containing extensions of the type system with dependent types and subtypes). For a comparison between HOL and PVS see [111]. Also popular are systems that use constructive mathematics as the starting point: the Coq system [33] based on the calculus of constructions [65], and NuPRL [62]. Jahob also has an interface to Coq, using Coq libraries that enable the axioms of classical logic and set theory. In contrast to these systems based on higher-order logic, Athena [15] is based on simple multisorted first-order logic. Athena offers a complete tactic language smoothly integrated with the formula language, and guarantees soundness using the notion of assumption bases [12] instead of using sequent proofs. Mizar [222] is a system for writing and checking proofs. Mizar is best known for the Mizar Mathematical Library, a substantial body of formally checked human-readable proofs of theorems in basic mathematics. The ACL2 system [139, 138] is known for a powerful simplifier and automated reasoning by induction. Numerous ACL2 case studies demonstrate that it is often possible to avoid using quantifiers and rich set-theoretic notations when reasoning about computer systems and obtain specifications that can be both executed efficiently and automated using rewriting techniques. The Ωmega system [231] was designed to experiment with the use of proof planing techniques in reasoning about higher-order logic. The Alloy modelling language [131] is based on first-order logic with transitive closure expressed using relational operators. Alloy has a module mechanism and its own type system [85]. The associated finite model finder [242] based on a translation to SAT has proved extremely successful in finding finite counterexamples for a range of specifications. A Jahob interface to Alloy would be very useful for debugging Jahob specifications and implementations. Other tools have been built specifically to support verification of software; I discuss them in Section 3.4.

Combination techniques. Among the most successful combination methods for decision procedures is the Nelson-Oppen technique [197], which forms the basis of successful implementations such as Simplify [82] and more recently CVC Lite [30] and Verifun [93]. The original Nelson-Oppen technique requires disjoint signatures and stably infinite theories [238]. Generalizations that weaken these requirements are described in [263, 104, 239, 105, 237]. Rewriting techniques are promising for deriving and combining decision procedures [142, 16, 241]. These results generally work with decidable classes of quantifier-free formulas with some background theory axioms. Among the results on combining quantified formulas are techniques based on quantifier elimination, such as Feferman-Vaught theorem for products of structures [90], term powers for term-like generalizations of finite powers [156], and decidability results generalizing the decidability of monadic second-order logic over binary strings to strings over elements of a structure with a decidable monadic theory [245].

In the context of interactive theorem provers, it is natural to consider combinations of automated techniques to increase the granularity of interactive proof steps. Such integration is heavily used in PVS [204]. Shankar observes in [230] that higher-order logics are consistent with the idea of automation if they are viewed as unifying logics for more tractable fragments. My approach is very much in the spirit of these observations, but uses a different technique for combining reasoning procedures. My motivation for this technique is data structure verification, but I expect that some of my results will be of use in interactive theorem provers as well. Integration with specialized reasoning was also present in the Boyer-Moore provers [41] and the HOL systems [126, 125, 184]. Isabelle has been integrated with monadic second-order logic over strings [31] but without field constraint analysis of Chapter 6, and with first-order provers [188, 187, 186] using a less specialized translation than the one in Chapter 5. Athena system has also been integrated with first-order provers and model finders [15, 14, 13]. Among software architectures and standards designed to support combinations of reasoning tools are the PROSPER toolkit [81], and MathWeb [102].

4.6 Conclusion

In this chapter I presented Jahob's higher-order logic based on Isabelle/HOL and described an interface to the Isabelle interactive theorem prover. I argued that classical higher-order logic is a natural notation for data structure properties. I outlined the idea of using splitting and approximation to combine different reasoning procedures for deciding interesting fragments of this logic. The combination technique I propose is simple yet useful for independently stated properties. The users can increase the effectiveness of the technique by separating properties using intermediate lemmas and specification variables. In subsequent chapters I illustrate that the automation of fragments of this logic is feasible and enables the verification of relevant data structure properties.

Chapter 5

First-Order Logic for Data Structure Implementation and Use

One of the main challenges in the verification of software systems is the analysis of unbounded data structures with dynamically allocated linked data structures and arrays. Examples of such data structures are linked lists, trees, and hash tables. The goal of these data structures is to efficiently implement sets and relations, with operations such as lookup, insert, and removal. This chapter explores the verification of programs with such data structures using resolution-based theorem provers for first-order logic with equality. The material in this chapter is based on [38]. The main result of this chapter is a particular translation from HOL to first-order logic. In addition to being useful for leveraging resolution-based theorem provers, this translation is useful for leveraging Nelson-Oppen style theorem provers with instantiation heuristics, whose one example is CVC Lite [30]. However, current Nelson-Oppen style implementations typically accept a *sorted* first-order language [216], so the question of omitting sorts in Section 5.3 does not arise.

Initial goal and the effectiveness of the approach. The initial motivation for using first-order provers is the observation that quantifier-free constraints on sets and relations that represent data structures can be translated to first-order logic or even its fragments [160]. This approach is suitable for verifying clients of data structures, because such verification need not deal with transitive closure present in the implementation of data structures. My initial goal was to incorporate first-order theorem provers into Jahob to verify data structure *clients*. While we have indeed successfully verified some data structure clients (such as the library example), we also discovered that this approach has a wider range of applicability than we had initially anticipated.

- We were able to apply this technique not only to data structure clients, but also to data structure implementations, using recursion and ghost variables and, in some cases, confining data structure mutation to newly allocated objects only.

- We found that there is no need in practice to restrict first-order properties to decidable fragments of first-order logic as suggested in [160], because many formulas that are not easily categorized into known decidable fragments have short proofs, and theorem provers can find these proofs effectively.

- Theorem provers were effective at dealing with quantified invariants that often arise when reasoning about unbounded numbers of objects.

- Using a simple partial axiomatization of linear arithmetic, we were able to verify ordering properties in a binary search tree, hash table invariants, and bounds for all array accesses.

The context of our results. I find our current results encouraging and attribute them to several factors. The use of ghost variables eliminated the need for transitive closure in our specifications, similarly to the use of more specialized forms of ghost variables in [185, 151]. Our use of recursion in combination with Jahob's approach to handling procedure calls resulted in more tractable verification conditions, suggesting that functional programming techniques are useful for reasoning about programs even in imperative programing languages. The semantics of procedure calls that we used in our examples is based on complete hiding of modifications to encapsulated objects. This semantics avoids the pessimistic assumption that every object is modified unless semantically proven otherwise, but currently prevents external references to encapsulated objects using simple syntactic checks. Finally, for those of our procedures that were written using loops instead of recursion, we manually supplied loop invariants.

Key ideas. The complexity of the properties we are checking made verification non-trivial even under the previously stated simplifying circumstances, and we found it necessary to introduce the following techniques for proving the generated verification conditions.

1. We introduce a translation to first-order logic with equality that avoids the potential inefficiencies of a general encoding of higher-order logic into first-order logic by handling the common cases and soundly approximating the remaining cases.

2. We use a translation to first-order logic that ignores information about sorts that would distinguish integers from objects. The results are smaller proof obligations and substantially better performance of provers. Moreover, we prove a somewhat surprising result: omitting such sort information is always sound and complete for disjoint sorts of the same cardinality. This avoids the need to separately check the generated proofs for soundness. Omitting sorts was essential for obtaining our results. Without it, difficult proof obligations are impossible to prove or take a substantially larger amount of time.

3. We use heuristics for filtering assumptions from first-order formulas that reduce the input problem size, speed up the theorem proving process, and improve the automation of the verification process.

The first two techniques are the main contribution of this chapter; the use of the third technique confirms previous observations about usefulness of assumption filtering in automatically generated first-order formulas [186].

Verified data structures and properties. Together, these techniques enabled us to verify, for example, that binary search trees and hash tables correctly implement their relational interfaces, including an accurate specification of removal operations. Such postconditions of operations in turn required verifying representation invariants: in binary search tree, they require proving sortedness of the tree; in hash table, they require proving that keys belong to the buckets given by their hash code. To summarize, our technique verifies that

1. representation invariants hold in the initial state;

2. each data structure operation

   - establishes the postcondition specifying the change of a user-specified abstract variable such as a set or relation (for example, an operation that updates a key is given by postcondition

     $$\mathsf{content} = (\mathsf{old\,content} \setminus \{(x, y) \mid x = \mathsf{key}\}) \cup \{(\mathsf{key}, \mathsf{value})\}$$

   - does not modify unintended parts of the state, for example, a mutable operation on an instantiable data structure preserves the values of all instances in the heap other than the receiver parameter;
   - preserves the representation invariants;
   - never causes run-time errors such as null dereference or array bounds violation.

We were able to prove such properties for an implementation of a hash table, a mutable list, a functional implementation of an ordered binary search tree, and a functional association list. All these data structures are instantiable (as opposed to global), which means that data structure clients can create an unbounded number of their instances. Jahob verifies that changes to one instance do not cause changes to other instances. In addition, we verified a simple client, a library system, that instantiates several set and relation data structures and maintains object-model like constraints on them in the presence of changes to sets and relations.

What is remarkable is that we were able to establish these results using a general-purpose technique and standard logical formalisms, without specializing our system for particular classes of properties. The fact that we can use continuously improving resolution-based theorem provers with standardized interfaces suggests that this technique is likely to remain competitive in the future.

From the theorem proving perspective, we expect the techniques we identify in this chapter to help make future theorem provers even more useful for program verification tasks. From the program verification perspective, our experience suggests that we can expect to use reasoning based on first-order logic to verify a wide range of data structures and then use these data structures to build and verify larger applications.

5.1 Binary Tree Example

We illustrate our technique using an example of a binary search tree implementing a finite map. Our implementation is persistent (immutable), meaning that the operations do not mutate existing objects, but only newly allocated objects. This makes the verification easier and provides a data structure which is useful in, for example, backtracking algorithms (see [203] for other advantages and examples of immutable data structures).

Figure 5-1 shows the public interface of our tree data structure. The interface introduces an abstract specification variable `content` as a set of (key,value)-pairs and specifies the contract of each procedure using a precondition (given by the `requires` keyword) and postcondition (given by the `ensures` keyword). The methods have no `modifies` clauses, indicating that they only mutate newly allocated objects.

Figure 5-2 presents the lookup operation. The operation examines the tree and returns the appropriate element. Note that, to prove that `lookup` is correct, one needs to know the relationship between the abstract variable `content` and the data structure fields `left`,

```
public ghost specvar content :: "(int * obj) set" = "{}";

public static FuncTree empty_set()
  ensures "result..content = {}"

public static FuncTree add(int k, Object v, FuncTree t)
  requires "v ~= null & (ALL y. (k,y) ~: t..content)"
  ensures "result..content = t..content + {(k,v)}"

public static FuncTree update(int k, Object v, FuncTree t)
  requires "v ~= null"
  ensures "result..content = t..content - {(x,y). x=k} + {(k,v)}"

public static Object lookup(int k, FuncTree t)
  ensures "(result ~= null & (k, result) : t..content)
         | (result = null & (ALL v. (k,v) ~: t..content))"

public static FuncTree remove(int k, FuncTree t)
  ensures "result..content = t..content - {(x,y). x=k}"
```

Figure 5-1: Method contracts for a tree implementation of a map

```
public static Object lookup(int k, FuncTree t)
/*: ensures "(result ~= null & (k, result) : t..content)
          | (result = null & (ALL v. (k,v) ~: t..content))" */
{
    if (t == null)
        return null;
    else
        if (k == t.key) return t.data;
        else if (k < t.key) return lookup(k, t.left);
        else return lookup(k, t.right);
}
```

Figure 5-2: Lookup operation for retrieving the element associated with a given key

```
class FuncTree {
  private int key;
  private Object data;
  private FuncTree left;
  private FuncTree right;

 /*:
  public ghost specvar content :: "(int * obj) set" = "{}";

  invariant nullEmpty: "this = null --> content = {}"

  invariant contentDefinition: "this ~= null & this..init -->
              content = {(key, data)} + left..content + right..content"

  invariant noNullData: "this ~= null --> data ~= null"

  invariant leftSmaller: "ALL k v. (k,v) : left..content --> k < key"
  invariant rightBigger: "ALL k v. (k,v) : right..content --> k > key"
  */
```

Figure 5-3: Fields and representation invariants for the tree implementation

`right`, `key`, and `data`. In particular, it is necessary to conclude that if an element is not found, then it is not in the data structure. Such conditions refer to private fields, so they are given by representation invariants. Figure 5-3 presents the representation invariants for our tree data structure. Using these representation invariants and the precondition, Jahob proves (in 4 seconds) that the postcondition of the `lookup` method holds and that the method never performs null dereferences. For example, when analyzing tree traversal in `lookup`, Jahob uses the sortedness invariants (`leftSmaller`, `rightBigger`) and the definition of tree content `contentDefinition` to narrow down the search to one of the subtrees.

Jahob also ensures that the operations preserve the representation invariants. Jahob reduces invariants in Figure 5-3 to global invariants by implicitly quantifying them over all allocated objects of `FuncTree` type. This approach yields simple semantics to constraints that involve multiple objects in the heap. When a method allocates a new object, the set of all allocated objects is extended, so a proof obligation will require that these newly allocated objects also satisfy their representation invariants at the end of the method.

Figure 5-4 shows the map update operation in our implementation. The postcondition of `update` states that all previous bindings for the given key are absent in the resulting tree. Note that proving this postcondition requires the sortedness invariants `leftSmaller`, `rightBigger`. Moreover, it is necessary to establish all representation invariants for the newly allocated `FuncTree` object.

The specification field `content` is a ghost field, so its value changes only in response to specification assignment statements, such as the one in the penultimate line of Figure 5-4. The use of ghost variables is sound an can be explained using simulation relations [77]. For example, if the developer incorrectly specifies specification assignments, Jahob will detect the violation of the representation invariants such as `contentDefinition`. If the developer specifies incorrect representation invariants, Jahob will fail to prove postconditions of observer operations such as `lookup` in Figure 5-2.

```
public static FuncTree update(int k, Object v, FuncTree t)
/*: requires "v ~= null"
    ensures "result..content = t..content - {(x,y). x=k} + {(k,v)}"   */
{
    FuncTree new_left, new_right;
    Object new_data;
    int new_key;
    if (t==null) {
        new_data = v;
        new_key = k;
        new_left = null;
        new_right = null;
    } else {
        if (k < t.key) {
            new_left = update(k, v, t.left);
            new_right = t.right;
            new_key = t.key;
            new_data = t.data;
        } else if (t.key < k) {
            new_left = t.left;
            new_right = update(k, v, t.right);
            new_key = t.key;
            new_data = t.data;
        } else {
            new_data = v;
            new_key = k;
            new_left = t.left;
            new_right = t.right;
        }
    }
    FuncTree r = new FuncTree();
    r.left = new_left;
    r.right = new_right;
    r.data = new_data;
    r.key = new_key;
    //: "r..content" := "t..content - {(x,y). x=k} + {(k,v)}";
    return r;
}
```

Figure 5-4: Map update implementation for functional tree

Jahob verifies (in 10 seconds) that the update operation establishes the postcondition, correctly maintains all invariants, and performs no null dereferences. As discussed in Section 3.3, Jahob establishes such conditions by first converting the Java program into a loop-free guarded-command language using user-provided or automatically inferred loop invariants. (The examples in this chapter mostly use recursion instead of loops.) A verification condition generator then computes a formula whose validity entails the correctness of the program with respect to its explicitly supplied specifications (such as invariants and procedure contracts) as well as the absence of run-time exceptions (such as null pointer dereferences, failing type casts, and array out of bounds accesses). The specification language and the generated verification conditions in Jahob are expressed in higher-order logic described in Section 4.1. In the rest of this chapter we show how we translate such verification conditions to first-order logic and prove them using theorem provers such as SPASS [248] and E [228].

5.2 Translation to First-Order Logic

This section presents our translation from an expressive subset of Isabelle formulas (the input language) to first-order unsorted logic with equality (the language accepted by first-order resolution-based theorem provers). The soundness of the translation is given by the condition that, if the translated formula is valid, so is the input formula.

Input language. The input language allows constructs such as lambda expressions, function update, sets, tuples, quantifiers, cardinality operators, and set comprehensions. The translation first performs type reconstruction. It uses the type information to disambiguate operations such as equality, whose translation depends on the type of the operands.

Splitting into sequents. Generated proof obligations can be represented as conjunctions of multiple statements, because they represent all possible paths in the verified procedure, the validity of multiple invariants and postcondition conjuncts, and the absence of run-time errors at multiple program points. The first step in the translation splits formulas into these individual conjuncts to prove each of them independently. This process does not lose completeness, yet it improves the effectiveness of the theorem proving process because the resulting formulas are smaller than the starting formula. Moreover, splitting enables Jahob to prove different conjuncts using different techniques, allowing the translation described in this chapter to be combined with other translation and approximation approaches, as described in Section 4.3. After splitting, the resulting formulas have the form of implications $A_1 \wedge \ldots \wedge A_n \rightarrow G$, which we call *sequents*. We call $A_1, \ldots, A_n$ the *assumptions* and call G the *goal* of the sequent. The assumptions typically encode a path in the procedure being verified, the precondition, class invariants that hold at procedure entry, as well as properties of our semantic model of memory and the relationships between sets representing Java types. During splitting, Jahob also performs syntactic checks that eliminate some simple valid sequents such as the ones where the goal G of the sequent is equal to one of the assumptions A_i.

Definition substitution and function unfolding. When one of the assumptions is a variable definition, the translation substitutes its content in the rest of the formula (using rules in Figure 5-5). This approach supports definitions of variables that have complex and higher-order types, but are used simply as shorthands, and avoids the full encoding of lambda abstraction in first-order logic. When the definitions of variables are lambda abstractions, the substitution enables beta reduction, which is done subsequently. In addition to beta reduction, this phase also expands the equality between functions using the extensionality rule ($f = g$ becomes $\forall x. f\, x = g\, x$).

Cardinality constraints. Constant cardinality constraints express natural generalizations of quantifiers. For example, the statement "there exists at most one element satisfying P" is given by $\mathsf{card}\,\{x.\, P\, x\} \leq 1$. Our translation reduces constant cardinality constraints to first-order logic with equality (using rules in Figure 5-6).

Set expressions. Our translation uses universal quantification to expand set operations into their set-theoretic definitions in terms of the set membership operator. This process also eliminates set comprehensions by replacing $x \in \{y \mid \varphi\}$ with $\varphi[y \mapsto x]$. (Figure 5-7 shows the details.) These transformations ensure that the only set expressions in formulas are either set variables or set-valued fields occurring on the right-hand side of the membership operator.

Our translation maps set variables to unary predicates: $x \in S$ becomes $S(x)$, where S is a predicate in first-order logic. This translation is applicable when S is universally quantified at the top level of the sequent (so it can be skolemized), which is indeed the case for the proof obligations arising from the examples in this chapter. Fields of type object or integer become uninterpreted function symbols: `y = x.f` translates as $y = f(x)$. Set-valued fields become binary predicates: `x ∈ y.f` becomes $F(y, x)$ where F is a binary predicate.

Function update. Function update expressions (encoded as functions fieldWrite and arrayWrite in our input language) translate using case analysis (Figure 5-8). If applied to arbitrary expressions, such case analysis would duplicate expressions, potentially leading to exponentially large expressions. To avoid this problem, the translation first flattens expressions by introducing fresh variables and then duplicates only variables and not expressions, keeping the translated formula polynomial.

Flattening. Flattening introduces fresh quantified variables, which could in principle create additional quantifier alternations, making the proof process more difficult. However, each variable can be introduced using either existential or universal quantifier because $\exists x. x{=}a \wedge \varphi$ is equivalent to $\forall x. x{=}a \rightarrow \varphi$. Our translation therefore chooses the quantifier kind that corresponds to the most recently bound variable in a given scope (taking into account the polarity), preserving the number of quantifier alternations. The starting quantifier kind at the top level of the formula is $\forall$, ensuring that freshly introduced variables for quantifier-free expressions become skolem constants.

Arithmetic. Resolution-based first-order provers do not have built-in arithmetic operations. Our translation therefore introduces axioms (Figure 5-10) that provide a partial axiomatization of integer operations $+, <, \leq$. In addition, the translation supplies axioms for the ordering relation between all numeric constants appearing in the input formula. Although incomplete, these axioms are sufficient to verify our list, tree, and hash table data structures.

Tuples. Tuples in the input language are useful, for example, as elements of sets representing relations, such as the `content` ghost field in Figure 5-3. Our translation eliminates tuples by transforming them into individual components. Figure 5-9 illustrates some relevant rewrite rules for this transformation. The translation maps a variable x denoting an n-tuple into n individual variables $x_1, \ldots, x_n$ bound in the same way as x. A tuple equality becomes a conjunction of equalities of components. The arity of functions changes to

Var-True

$$\frac{(H_1 \land \cdots \land H_{i-1} \land v \land H_{i+1} \land \cdots \land H_n) \Longrightarrow G}{\big[(H_1 \land \cdots \land H_{i-1} \land H_{i+1} \land \cdots \land H_n) \Longrightarrow G\big]\{v \mapsto \mathit{True}\}}$$

Var-False

$$\frac{(H_1 \land \cdots \land H_{i-1} \land \neg v \land H_{i+1} \land \cdots \land H_n) \Longrightarrow G}{\big[(H_1 \land \cdots \land H_{i-1} \land H_{i+1} \land \cdots \land H_n) \Longrightarrow G\big]\{v \mapsto \mathit{False}\}}$$

Var-Def

$$\frac{(H_1 \land \cdots \land H_{i-1} \land v = \varphi \land H_{i+1} \land \cdots \land H_n) \Longrightarrow G}{\big[(H_1 \land \cdots \land H_{i-1} \land H_{i+1} \land \cdots \land H_n) \Longrightarrow G\big]\{v \mapsto \varphi\}}$$

$v \notin FV(\varphi)$
Var-True cannot be applied
Var-False cannot be applied

Figure 5-5: Rules for definition substitution

accommodate all components, so a function taking an n-tuple and an m-tuple becomes a function symbol of arity $n + m$. The translation handles sets as functions from elements to booleans. For example, a relation-valued field `content` of type `obj => (int * obj) set` is viewed as a function `obj => int => obj => bool` and therefore becomes a ternary predicate symbol.

Approximation. Our translation maps higher-order formulas into first-order logic without encoding lambda calculus or set theory, so there are constructs that it cannot translate exactly. Examples include transitive closure (which, in the case of tree-like data structures, can be translated into monadic second-order logic using the techniques of Chapter 6) and symbolic cardinality constraints (which, in the absence of relations, can be handled using the BAPA decision procedure of Chapter 7). Our first-order translation approximates such subformulas in a sound way, by replacing them with true or false depending on the polarity of the subformula occurrence. The result of the approximation is a stronger formula whose validity implies the validity of the original formula.

Simplifications and further splitting. In the final stage, the translation performs a quick simplification pass that reduces the size of formulas, by, for example, eliminating most occurrences of true and false. Next, because constructs such as equality of sets and functions introduce conjunctions, the translation performs further splitting of the formula to improve the success of the proving process. [1]

5.3 From Multisorted to Unsorted Logic

This section discusses our approach for handling type and sort information in the translation to first-order logic with equality. This approach proved essential for making verification of our examples feasible. The key insight is that omitting sort information 1) improves the performance of the theorem proving effort, and 2) is guaranteed to be sound in our context.

To understand our setup, note that the verification condition generator in Jahob produces proof obligations in higher-order logic notation whose type system essentially corresponds to simply typed lambda calculus [26] (we allow some simple forms of parametric

[1]We encountered an example of a formula $\varphi_1 \land \varphi_2$ where a theorem prover proves each of φ_1 and φ_2 independently in a few seconds, but requires more than 20 minutes to prove $\varphi_1 \land \varphi_2$.

Card-Constraint-eq
$$\frac{\mathsf{card}\ S = k}{\mathsf{card}\ S \leq k\ \wedge \mathsf{card}\ S \geq k}$$

Card-Constraint-Leq
$$\frac{\mathsf{card}\ S \leq k}{\exists x_1, \dots, x_k . S \subseteq \{x_1, \dots, x_k\}}$$

Card-Constraint-Geq
$$\frac{\mathsf{card}\ S \geq k}{\exists x_1, \dots, x_k .\ \{x_1, \dots, x_k\} \subseteq S \wedge \bigwedge_{1 \leq i < j \leq k} x_i \neq x_j}$$

Figure 5-6: Rules for constant cardinality constraints

Set-Inclusion
$$\frac{S_1 \subseteq S_2}{\forall x. x \in S_1 \rightarrow x \in S_2}$$

Set-Equality
$$\frac{S_1 = S_2}{\forall x. x \in S_1 \iff x \in S_2}$$

Intersection
$$\frac{x \in S_1 \cap S_2}{x \in S_1 \wedge x \in S_2}$$

Union
$$\frac{x \in S_1 \cup S_2}{x \in S_1 \vee x \in S_2}$$

Difference
$$\frac{x \in S_1 \setminus S_2}{x \in S_1 \wedge x \notin S_2}$$

FiniteSet
$$\frac{x \in \{O_1, \dots, O_k\}}{x = O_1 \vee \dots \vee x = O_k}$$

Comprehension
$$\frac{x \in \{y \mid \varphi\}}{\varphi[y \mapsto x]}$$

Figure 5-7: Rules for complex set expressions

polymorphism but expect each occurrence of a symbol to have a ground type). The type system in our proof obligations therefore has no subtyping, so all Java objects have type obj. The verification-condition generator encodes Java classes as immutable sets of type obj set. It encodes primitive Java integers as mathematical integers of type int (which is disjoint from obj). The result of the translation in Section 5.2 is a formula in *multisorted first-order logic with equality* and two disjoint sorts, obj and int.[2] On the other side, the standardized input language for first-order theorem provers is *untyped first-order logic with equality*. The key question is the following: *How should we encode multisorted first-order logic into untyped first-order logic?*

The standard approach [177, Chapter 6, Section 8] is to introduce a unary predicate P_s for each sort s, replace $\exists x{::}s.F(x)$ with $\exists x.P_s(x) \wedge F(x)$, and replace $\forall x{::}s.F(x)$ with $\forall x.P_s(x) \rightarrow F(x)$ (where $x :: s$ in multisorted logic denotes that the variable x has the sort s). In addition, for each function symbol f of sort $s_1 \times \dots s_n \rightarrow s$, introduce a Horn clause $\forall x_1, \dots, x_n.\ P_{s_1}(x_1) \wedge \dots \wedge P_{s_n}(x_n) \rightarrow P_s(f(x_1, \dots, x_n))$.

The standard approach is sound and complete. However, it makes formulas larger, often substantially slowing down the automated theorem prover. What if we omitted the sort information given by unary sort predicates P_s, representing, for example, $\forall x{::}s.F(x)$ simply as $\forall x.F(x)$? For potentially overlapping sorts, this approach is unsound. As an example, take the conjunction of two formulas $\forall x{::}\mathsf{Node}.F(x)$ and $\exists x{::}\mathsf{Object}.\neg F(x)$ for distinct sorts Object and Node where Node is a subsort of Object. These assumptions are consistent in

[2]The resulting multisorted logic has no sort corresponding to booleans (as in [177, Chapter 6]). Instead, propositional operations are part of the logic itself.

Object-Field-Write-Read
$$\frac{V_1 = \mathsf{fieldWrite}\,(f, V_2, V_3)(V_4)}{(V_4 = V_2 \wedge V_1 = V_3) \vee (V_4 \neq V_2 \wedge V_1 = f(V_4))}$$

Object-Array-Write-Read
$$\frac{V_1 = \mathsf{arrayWrite}\,(f_a, V_2, V_3, V_4)(V_5, V_6)}{(V_5 = V_2 \wedge V_6 = V_3 \wedge V_1 = V_4) \vee (\neg\,(V_5 = V_2 \wedge V_6 = V_3) \wedge V_1 = f_a(V_5, V_6))}$$

Function-argument
$$\frac{V = g(V_1, ..., V_{i-1}, C, V_{i+1}, ..., V_k)}{\exists u.u = C \wedge V = g(V_1, ..., V_{i-1}, u, V_{i+1}, ..., V_k)}$$

Equality-Normalization
$$\frac{C = V}{V = C}$$

Equality-Unfolding
$$\frac{C_1 = C_2}{\exists v.v = C_1 \wedge v = C_2}$$

Set-Field-Write-Read
$$\frac{V_1 \in \mathsf{fieldWrite}\,(f, V_2, V_3)(V_4)}{(V_4 = V_2 \wedge V_1 \in V_3) \vee (V_4 \neq V_2 \wedge V_1 \in f(V_4))}$$

Membership-Unfolding
$$\frac{C \in T}{\exists v.v = C \wedge v \in T}$$

Figure 5-8: Rewriting rules to rewrite complex field expressions. C denotes a complex term; V denotes a variable; f denotes a field or array function identifier (not a complex expression).

$$\frac{(x_1, ..., x_n) = (y_1, ..., y_n)}{\bigwedge_{i=1}^{n} x_i = y_i} \qquad \frac{z = (y_1, ..., y_n)}{\bigwedge_{i=1}^{n} z_i = y_i} \qquad \frac{z = y \qquad z : S_1 \times ... \times S_n}{\bigwedge_{i=1}^{n} z_i = y_i}$$

$$\frac{(y_1, ..., y_n) \in S}{S(y_1, ..., y_n)} \qquad \frac{(y_1, ..., y_n) \in x.f}{F(x, y_1, ..., y_n)} \qquad \frac{z \in S \qquad z : S_1 \times ... \times S_n}{S(z_1, ..., z_n)}$$

$$\frac{z \in x.f \qquad z : S_1 \times ... \times S_n}{F(x, z_1, ..., z_n)} \qquad \frac{Q(z : S_1 \times ... \times S_n).\varphi}{Q(z_1 : S_1, ..., z_n : S_n).\varphi}$$

Figure 5-9: Rules for removal of tuples

$$
\begin{array}{rl}
\forall n. & n \le n \\
\forall n\,m. & (n \le m \land m \le n) \to n = m \\
\forall n\,m. & (n \le m \land m \le o) \to n \le o \\
\forall n\,m. & (n \le m) \iff (n = m \lor \neg(m \le n)) \\
\forall n\,m\,p\,q. & (n \le m \land p \le q) \to n + p \le m + q \\
\forall n\,m\,p\,q. & (n \le m \land p \le q) \to n - q \le m - p \\
\forall n\,m\,p. & n \le m \to n + p \le m + p \\
\forall n\,m\,p. & n \le m \to n - p \le m - p \\
\forall n\,m. & n + m = m + n \\
\forall n\,m\,p. & (n + m) + p = n + (m + p) \\
\forall n. & n + 0 = n \\
\forall n. & n - 0 = n \\
\forall n. & n - n = 0
\end{array}
$$

Figure 5-10: Arithmetic axioms optionally conjoined with the formulas

multisorted logic. However, their unsorted version $\forall x.F(x) \land \exists x.\neg F(x)$ is contradictory, and would allow a verification system to unsoundly prove arbitrary claims.

In our case, however, the two sorts considered (int and obj) are disjoint. Moreover, there is no overloading of predicate or function symbols. If we consider a standard resolution proof procedure for first-order logic [19] (without paramodulation) under these conditions, we can observe the following.

Observation 1 *Performing an unsorted resolution step on well-sorted clauses (while ignoring sorts in unification) generates well-sorted clauses.*

As a consequence, there is a bijection between resolution proofs in multisorted and unsorted logic. By completeness of resolution, omitting sorts and using unsorted resolution is a sound and complete technique for proving multisorted first-order formulas.

Observation 1 only applies if each symbol has a unique sort (type) signature (i.e., there is no overloading of symbols), which is true for all symbols *except for equality*. To make it true for equality, a multi-sorted language with disjoint sorts would need to have one equality predicate for each sort. Unfortunately, theorem provers we consider have a built-in support only for one privileged equality symbol. Using user-defined predicates and supplying congruence axioms would fail to take advantage of the support for paramodulation rules [199] in these provers. What if, continuing our brave attempt at omitting sorts, we merge translation of all equalities, using the special equality symbol regardless of the sorts to which it applies? The result is unfortunately unsound in general. As an example, take the conjunction of formulas $\forall x{::}\mathsf{obj}.\forall y{::}\mathsf{obj}.x = y$ and $\exists x{::}\mathsf{int}.y{::}\mathsf{int}.\neg(x = y)$. These formulas state that the obj sort collapses to a single element but the int sort does not. Omitting sort information yields a contradiction and is therefore unsound. Similar examples exists for statements that impose other finite bounds on distinct sorts.

In our case, however, we can assume that both int and obj are countably infinite. More generally, when the interpretation of disjoint sorts are sets of equal cardinality, such examples have the same truth value in the multisorted case as well as in the case with equality. More precisely, we have the following result. Let φ^* denote the result of omitting all sort

Benchmark	Time (s)				Proof length		Generated clauses			
	SPASS		E		SPASS		SPASS		E	
	w/o	w.	w/o	w.	w/o	w.	w/o	w.	w/o	w.
remove	1.1	5.3	30.0	349.0	155	799	9425	18376	122508	794860
	0.3	3.6	10.4	42.0	309	1781	1917	19601	73399	108910
	4.9	9.8	15.7	18.0	174	1781	27108	33868	100846	256550
	0.5	8.1	12.5	45.9	301	1611	3922	31892	85164	263104
	4.7	8.1	17.9	19.3	371	1773	28170	37244	109032	176597
	0.3	7.9	10.6	41.8	308	1391	3394	41354	65700	287253
remove_max	0.22	$+\infty$	59.0	76.5	97	-	1075	-	872566	953451
	6.8	78.9	14.9	297.6	1159	2655	19527	177755	137711	1512828
	0.8	34.8	38.1	0.7	597	4062	5305	115713	389334	7595

Figure 5-11: Verification time, and proof data using the prover SPASS, on the hardest formulas from the functional tree example.

information from a multisorted formula φ and representing the equality (regardless of the sort of arguments) using the built-in equality symbol.

Theorem 2 *Assume that there are finitely many pairwise disjoint sorts, that their interpretations are sets of equal cardinality, and that there is no overloading of predicate and function symbols other than equality. Then there exists a function mapping each multisorted structure $\mathcal{I}$ into an unsorted structure $\mathcal{I}^*$ and each multisorted environment ρ to an unsorted environment ρ^*, such that the following holds: for each formula φ, structure $\mathcal{I}$, and a well-sorted environment ρ,*

$$[\![\varphi^*]\!]^{\mathcal{I}^*}_{\rho^*} \qquad \textit{if and only if} \qquad [\![\varphi]\!]^{\mathcal{I}}_{\rho}$$

The proof of Theorem 2 is in Section 5.9. It constructs $\mathcal{I}^*$ by taking a new set S of same cardinality as the sort interpretations $S_1, \ldots, S_n$ in $\mathcal{I}$, and defining the interpretation of symbols in $\mathcal{I}^*$ by composing the interpretation in $\mathcal{I}$ with bijections $f_i : S_i \rightarrow S$. Theorem 2 implies that if a formula $(\neg\psi)^*$ is unsatisfiable, then so is $\neg\psi$. Therefore, if ψ^* is valid, so is ψ. In summary, *for disjoint sorts of same cardinality, omitting sorts is a sound method for proving validity, even in the presence of an overloaded equality symbol.*

A resolution theorem prover with paramodulation rules can derive ill-sorted clauses as consequences of φ^*. However, Theorem 2 implies that the existence of a refutation of φ^* implies that φ is also unsatisfiable, guaranteeing the soundness of the approach. This approach is also complete. Namely, notice that stripping sorts only *increases* the set of resolution steps that can be performed on a set of clauses. Therefore, we can show that if there exists a proof for φ, there exists a proof of φ^*. Moreover, *the shortest proof for the unsorted case is no longer than any proof in multisorted case.* As a result, any advantage of preserving sorts comes from the reduction of the branching factor in the search, as opposed to the reduction in proof length.

Impact of omitting sort information. Figure 5-11 shows the effect of omitting sorts on some of the most problematic formulas that arise in our benchmarks. They are the formulas that take more than one second to prove using SPASS with sorts, in the two hardest methods of our tree implementation. The figure shows that omitting sorts usually yields a speed-up of one order of magnitude, and sometimes more. In our examples, the converse situation, where omitting sorts substantially slows down the theorem proving process, is rare.

5.4 Assumption Filtering

Typically, the theorem prover only needs a subset of assumptions of a sequent to establish its validity. Indeed, the `FuncTree.remove` procedure has a median proof length of 4; with such a small number of deduction steps only a fraction of all the assumptions are necessary. Unnecessary assumptions can dramatically increase the running time of theorem provers and cause them to fail to terminate in a reasonable amount of time, despite the use of selection heuristics in theorem prover implementations.

Finding a minimal set of assumption is in general as hard as proving the goal. We therefore use heuristics that run in polynomial time to select assumptions likely to be relevant. Our technique is based on [186], but is simpler and works at the level of formulas (after definition substitution and beta reduction) as opposed to clauses. The technique ranks the assumptions and sorts them in the ranking order. A command-line option indicates the percentage of the most highly ranked assumptions to retain in the proof obligation.

Impact of filtering. We verified the impact of assumption filtering on a set of 2000 valid formulas generated by our system, with the average number of assumptions being 48.5 and the median 43. After ranking the assumptions, we measured the number of the most relevant assumptions that we needed to retain for the proof to still succeed. With our simple ranking technique, the average required number of relevant assumptions was 16, and the median was 11. One half of the formulas of this set are proved by retaining only the top one third of the original assumptions.

Assumption filtering yields an important speed-up in the verification of the hash table implementation of a relation. The hash table is implemented using an array, and our system checks that all array accesses are within bounds. This requires the ordering axioms for the $\leq$ operator. However, when proving that operations correctly update the hash table content, these axioms are not required, and confuse SPASS: the verification of the insertion method takes 211 seconds with all assumptions, and only 1.3 second with assumption filtering set to 50%. In some cases this effect could be obtained manually, by asking the system to try to prove the formula first without, and then with the arithmetic axioms, but assumption filtering makes the specification of command-line parameters simpler and decreases the overall running time.

5.5 Experimental Results

We implemented our translation to first-order logic and the interfaces to the first-order provers E [228] (using the TPTP format for first-order formulas [234]) and SPASS [248] (using its native format). We also implemented filtering described in Section 5.4 to automate the selection of assumptions in proof obligations. We evaluated our approach by implementing several data structures, using the system during their development. In addition to the implementation of a relation as a functional tree presented in Section 5.1, we ran our system on dynamically instantiable sets and relations implemented as a functional singly-linked list, an imperative linked list, and a hash table. We also verified operations of a data structure client that instantiates a relation and two sets and maintains invariants between them.

Table 5-12 illustrates the benchmarks we ran through our system and shows their verification times. Lines of code and of specifications are counted without blank lines or

Benchmark	lines of code	lines of specification	number of methods
Relation as functional list	76	26	9
Relation as functional Tree	186	38	10
Set as imperative list	60	24	9
Library system	97	63	9
Relation as hash table	69	53	10

Benchmark	Prover	method	total time (sec)	prover time (sec)	formulas proved
AssocList	E	cons	0.9	0.8	9
		remove_all	1.7	1.1	5
		remove	3.9	2.6	7
		lookup	0.7	0.4	3
		image	1.3	0.6	4
		inverseImage	1.2	0.6	4
		domain	0.9	0.5	3
		entire class	**11.8**	**7.3**	**44**
FuncTree	SPASS+E	add	7.2	5.7	24
		update	9.0	7.4	28
		lookup	1.2	0.6	7
		min	7.2	6.6	21
		max	7.2	6.5	22
		removeMax	106.5 (12.7)	46.6+59.3	9+11
		remove	17.0	8.2+ 0	26+0
		entire class	**178.4**	**96.0+65.7**	**147+16**
Imperative List	SPASS	add	1.5	1.2	9
		member	0.6	0.3	7
		getOne	0.1	0.1	2
		remove	11.4	9.9	48
		entire class	**17.9**	**14.9+0.1**	**74**
Library	E	currentReader	1.0	0.9	5
		checkOutBook	2.3	1.7	6
		returnBook	2.7	2.1	7
		decommissionBook	3.0	2.2	7
		entire class	**20.0**	**17.6**	**73**
HashTable	SPASS	init	25.5 (3.8)	25.2 (3.4)	12
		add	2.7	1.6	7
		add1	22.7	22.7	14
		lookup	20.8	20.3	9
		remove	57.1	56.3	12
		update	1.4	0.8	2
		entire class	**119**	**113.8**	**75**

Figure 5-12: Benchmarks Characteristics and Verification Times

comments. [3]

Our system accepts as command-line parameters timeouts, percentage of retained assumptions in filtering, and two flags that indicate desired sets of arithmetic axioms. For each module, we used a fixed set of command line options to verify all the procedures in that module. Some methods can be verified faster (in times shown in parentheses) by choosing a more fine-tuned set of options. Jahob allows specifying a cascade of provers to be tried in sequence; when we used multiple provers we give the time spent in each prover and the number of formulas proved by each of them. Note that all steps of the cascade run on the same input and are perfectly parallelizable. Running all steps in parallel is an easy way to reduce the total running time. Similar parallelization opportunities arise across different conjuncts that result from splitting, because splitting is done ahead of time, before invoking any theorem provers.

The values in the "entire class" row for each module are not the sum of all the other rows, but the time actually spent in the verification of the entire class, including some methods not shown and the verification that the invariants hold initially. Running time of first-order provers dominates the verification time, the remaining time is mostly spent in our simple implementation of polymorphic type inference for higher-order logic formulas.

Verification experience. The time we spent to verify these benchmarks went down as we improved the system and gained experience using it. It took approximately one week to code and verify the ordered trees implementation. However, it took only half a day to write and verify a simple version of the hash table. It took another few days to verify an augmented version with a rehash function that can dynamically resize its array when its filling ratio is too high.

On formulas generated from our examples, SPASS seems to be overall more effective. However, E is more effective on some hard formulas involving complex arithmetic. Therefore, we use a cascading system of multiple provers. We specify a sequence of command line options for each prover, which indicate the timeout to use, the sets of axioms to include, and the amount of filtering to apply. For example, to verify the entire `FuncTree` class, we used the following cascade of provers: 1) SPASS with two-second timeout and 50% assumption filtered; 2) SPASS with two-second timeout, axioms of the order relation over integers and 75% assumption filtered; and 3) E without timeout, with the axioms of the order relation and without filtering. Modifying these settings can result in a great speed-up (for example, `FuncTree.removeMax` verifies in 13 seconds with tuned settings as opposed to 106 seconds with the global settings common to the entire class). Before we implemented assumption filtering, we faced difficulties finding a set of options allowing the verification of the entire `FuncTree` class. Namely, some proof obligations require arithmetic axioms, and for others adding these settings would cause the prover to fail. Next, some proof obligations require background axioms (general assumptions that encode our memory model), but some work much faster without them. Assumption filtering allows the end-user to worry less about these settings.

[3]We ran the verification on a single-core 3.2 GHz Pentium 4 machine with 3GB of memory, running GNU/Linux. As first-order theorem provers we used SPASS and E in their automatic settings. The E version we used comes from the CASC-J3 (Summer 2006) system archive and calls itself v0.99pre2 "Singtom". We used SPASS v2.2, which comes from its official web page.

$$\begin{array}{rcl} \varphi & ::= & P(t_1,\ldots,t_n) \mid t_1 = t_2 \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \exists x.\, F \\ t & ::= & x \mid f(t_1,\ldots,t_n) \end{array}$$

Figure 5-13: Syntax of Unsorted First-Order Logic with Equality

$$\varphi \quad ::= \quad P(t_1,\ldots,t_n) \mid t_1 = t_2 \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \exists x{::}s.\, F$$

Figure 5-14: Syntax of Multisorted First-Order Logic with Equality

5.6 First-Order Logic Syntax and Semantics

To avoid any ambiguity, this section presents the syntax and semantics of unsorted and multisorted first-order logic. We use this notation in the proofs in the following sections.

5.6.1 Unsorted First-Order Logic with Equality

An unsorted signature Σ is given by:

- a set $\mathcal{V}$ of variables;
- a set $\mathcal{P}$ of predicate symbols, each symbol $P \in \mathcal{P}$ with arity $\mathsf{ar}(P) > 0$;
- a set $\mathcal{F}$ of function symbols, each symbol $f \in \mathcal{F}$ with arity $\mathsf{ar}(f) \geq 0$.

Figure 5-13 shows the syntax of unsorted first-order logic. Constants are function symbols of arity 0.

An unsorted Σ-structure $\mathcal{I}$ is given by:

- the domain set $X = \mathsf{dom}(\mathcal{I})$;
- for every predicate $P \in \mathcal{P}$ with $\mathsf{ar}(P) = n$, the interpretation $[\![P]\!]^{\mathcal{I}} \subseteq X^n$ defining the tuples on which P is true;
- for every function symbol f in $\mathcal{F}$ of arity n, a set of tuples $[\![f]\!]^{\mathcal{I}} \subseteq X^{n+1}$, which represents the graph of a total function $X^n \to X$.

An $\mathcal{I}$-environment ρ is a function $\mathcal{V} \to X$ from variables to domain elements.

The interpretation of a term t in structure $\mathcal{I}$ and environment ρ is denoted $[\![t]\!]^{\mathcal{I}}_{\rho}$ and is given inductively as follows:

- $[\![x]\!]^{\mathcal{I}}_{\rho} = \rho(x)$, if $x \in \mathcal{V}$ is a variable;
- $[\![f(x_1,\ldots,x_n)]\!]^{\mathcal{I}}_{\rho} = y$ where $([\![x_1]\!]^{\mathcal{I}}_{\rho},\ldots,[\![x_n]\!]^{\mathcal{I}}_{\rho}, y) \in [\![f]\!]^{\mathcal{I}}$, if $f \in \mathcal{F}$ is a function symbol of arity $n \geq 0$.

Interpretation of a formula φ in structure $\mathcal{I}$ and environment ρ is denoted $[\![\varphi]\!]^{\mathcal{I}}_{\rho}$ and is given inductively as follows:

$$\begin{array}{rcl}
[\![P(t_1,\ldots,t_n)]\!]^{\mathcal{I}}_{\rho} & = & ([\![t_1]\!]^{\mathcal{I}}_{\rho},\ldots,[\![t_n]\!]^{\mathcal{I}}_{\rho}) \in [\![P]\!]^{\mathcal{I}} \\
[\![t_1 = t_2]\!]^{\mathcal{I}}_{\rho} & = & ([\![t_1]\!]^{\mathcal{I}}_{\rho}{=}[\![t_2]\!]^{\mathcal{I}}_{\rho}) \\
[\![\varphi_1 \land \varphi_2]\!]^{\mathcal{I}}_{\rho} & = & [\![\varphi_1]\!]^{\mathcal{I}}_{\rho} \land [\![\varphi_2]\!]^{\mathcal{I}}_{\rho} \\
[\![\neg\varphi]\!]^{\mathcal{I}}_{\rho} & = & \neg[\![\varphi]\!]^{\mathcal{I}}_{\rho} \\
[\![\exists x.\varphi]\!]^{\mathcal{I}}_{\rho} & = & \exists a \in \mathsf{dom}(\mathcal{I}).[\![\varphi]\!]^{\mathcal{I}}_{\rho[x\mapsto a]}
\end{array}$$

where $\rho[x \mapsto a](y) = \rho(y)$ for $y \neq x$ and $\rho[x \mapsto a](x) = a$.

5.6.2 Multisorted First-Order Logic with Equality

A multisorted signature Σ with sorts $\sigma = \{s_1,\ldots,s_n\}$ is given by:

- a set $\mathcal{V}$ of variables, each variable $x \in \mathcal{V}$ with its sort $\mathsf{ar}(x) \in \sigma$;
- a set $\mathcal{P}$ of predicates, each symbol $P \in \mathcal{P}$ with a sort signature $\mathsf{ar}(P) \in \sigma^n$ for some $n > 0$;
- a set $\mathcal{F}$ of function symbols, each symbol $f \in F$ with a sort signature $\mathsf{ar}(f) \in \sigma^{n+1}$; we write $\mathsf{ar}(f) : s_1 * \ldots * s_n \longrightarrow s_{n+1}$ if $\mathsf{ar}(f) = (s_1,\ldots,s_n,s_{n+1})$.

Figure 5-14 shows the syntax of multisorted first-order logic with equality, which differs from the syntax of the unsorted first-order logic with equality in that each quantifier specifies the sort of the bound variable. In addition, we require the terms and formulas to be well-sorted, which means that predicates and function symbols only apply to arguments of the corresponding sort, and equality applies to terms of the same sort.

A multisorted Σ-structure $\mathcal{I}$ is given by:

- for each sort s_i, a domain set $S_i = [\![s_i]\!]^{\mathcal{I}}$;
- for every predicate P in $\mathcal{P}$ of type $s_1 * \ldots * s_n$, a relation $[\![P]\!]^{\mathcal{I}} \subseteq S_1 \times \ldots \times S_n$ for $[\![s_i]\!]^{\mathcal{I}} = S_i$, defining the tuples on which P is true;
- for every function symbol f in $\mathcal{F}$ of type $s_1 * \ldots * s_n \longrightarrow s_{n+1}$, the function graph $f \subseteq S_1 \times \ldots \times S_n \times S_{n+1}$ of a total function that interprets symbol f.

A multisorted environment ρ maps every variable $x \in \mathsf{Var}$ with sort s_i to an element of S_i, so $\rho(x) \in [\![\mathsf{ar}(x)]\!]^{\mathcal{I}}$;

We interpret terms the same way as in the unsorted case. We interpret formulas analogously as in the unsorted case, with each bound variable of sort s_i ranging over the interpretation S_i of the sort s_i.

$$\begin{array}{rcl}
[\![P(t_1,\ldots,t_n)]\!]^{\mathcal{I}}_{\rho} & = & ([\![t_1]\!]^{\mathcal{I}}_{\rho},\ldots,[\![t_n]\!]^{\mathcal{I}}_{\rho}) \in [\![P]\!]^{\mathcal{I}} \\
[\![t_1 = t_2]\!]^{\mathcal{I}}_{\rho} & = & ([\![t_1]\!]^{\mathcal{I}}_{\rho}{=}[\![t_2]\!]^{\mathcal{I}}_{\rho}) \\
[\![\varphi_1 \land \varphi_2]\!]^{\mathcal{I}}_{\rho} & = & [\![\varphi_1]\!]^{\mathcal{I}}_{\rho} \land [\![\varphi_2]\!]^{\mathcal{I}}_{\rho} \\
[\![\neg\varphi]\!]^{\mathcal{I}}_{\rho} & = & \neg[\![\varphi]\!]^{\mathcal{I}}_{\rho} \\
[\![\exists x{::}s.\varphi]\!]^{\mathcal{I}}_{\rho} & = & \exists a \in [\![s]\!]^{\mathcal{I}}.[\![\varphi]\!]^{\mathcal{I}}_{\rho[x\mapsto a]}
\end{array}$$

$$\begin{array}{rcl}
x^* & \equiv & x \\
f(t_1, \ldots, t_n)^* & \equiv & f({t_1}^*, \ldots, {t_n}^*) \\
P(t_1, \ldots, t_n)^* & \equiv & P({t_1}^*, \ldots, {t_n}^*) \\
(t_1 = t_2)^* & \equiv & ({t_1}^* = {t_2}^*) \\
(\varphi_1 \land \varphi_2)^* & \equiv & {\varphi_1}^* \land {\varphi_2}^* \\
(\neg\varphi)^* & \equiv & \neg\,(\varphi^*) \\
(\exists x{::}s.\varphi)^* & \equiv & \exists x.\,(\varphi^*)
\end{array}$$

Figure 5-15: Unsorted formula associated with a multisorted formula

5.6.3 Notion of Omitting Sorts from a Formula

If φ is a multisorted formula, we define its unsorted version φ^* by eliminating all type annotations. For a term t, we would write the term t^* in the same way as t, but we keep in mind that the function symbols in t^* have an unsorted signature. The rules in Figure 5-15 make this definition more precise.

5.7 Omitting Sorts in Logic without Equality

In this section we prove that omitting sorts is sound in the first-order language without equality. We therefore assume that there is no equality symbol, and that each predicate and function symbol has a unique (ground) type. Under these assumptions we show that unification for multisorted and unsorted logic coincide, which implies that resolution proof trees are the same as well. Completeness and soundness of resolution in multisorted and unsorted logic then implies the equivalence of the validity in unsorted and multisorted logics without equality.

5.7.1 Multisorted and Unsorted Unification

Unification plays a central role in the resolution process as well as in the proof of our claim. We review it here for completeness, although the concepts we use are standard. We provide definitions for the multisorted case. To obtain the definitions for the unsorted case, assume that all terms and variables have one "universal" sort.

Definition 3 (Substitution) *A substitution σ is a mapping from terms to terms such that $\sigma(f(t_1, \ldots, t_n)) = f(\sigma(t_1), \ldots, \sigma(t_2))$.*

Substitutions are homomorphisms in the free algebra of terms with variables.

Definition 4 (Unification problem) *A* unification problem *is a set of pairs of terms of the form: $\mathcal{P} = \{s_1 \doteq t_1, \ldots, s_n \doteq t_n\}$, where all terms are well-sorted, and both sides of the $\doteq$ operator have the same sort.*

Definition 5 (Unifier) *A* unifier *σ for a problem $\mathcal{P}$ is a substitution such that $\sigma(s_i) = \sigma(t_i)$ for all constraints $s_i \doteq t_i$ in $\mathcal{P}$.*

Decompose

$$\frac{\mathcal{P} \cup \{f(s_1, \ldots, s_n) \doteq f(t_1, \ldots, t_n)\}}{\mathcal{P} \cup \{s_1 \doteq t_1, \ldots, s_n \doteq t_n\}}$$

Orient

$$\frac{\mathcal{P} \cup \{t \doteq x\} \qquad t \notin \mathcal{V}}{\mathcal{P} \cup \{x \doteq t\}}$$

Replace

$$\frac{\mathcal{P} \cup \{x \doteq s\} \qquad x \in Var(\mathcal{P}) \qquad x \notin FV(s)}{(\mathcal{P}[x \mapsto s]) \cup \{x \doteq s\}}$$

Erase

$$\frac{\mathcal{P} \cup \{s \doteq s\}}{\mathcal{P}}$$

Figure 5-16: Unification algorithm

Definition 6 (Resolved form) *A problem $\mathcal{P}$ is in* resolved form *iff it is of the form $\{x_1 \doteq t_1, \ldots, x_n \doteq t_n\}$, where, for each $1 \leq i \leq n$:*

1. *all x_i are pairwise distinct variables ($i \neq j \rightarrow x_i \neq x_j$).*
2. *x_i does not appear in t_i ($x_i \notin FV(t_i)$).*

Definition 7 (Unifier for resolved form) *Let $\mathcal{P} = \{x_1 \doteq t_1, \ldots, x_n \doteq t_n\}$ be a problem in resolved form. The* unifier associated with $\mathcal{P}$ *is the substitution $\sigma_{\mathcal{P}} = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$.*

We define the unification algorithm as the set of rewriting rules in Figure 5-16. We assume a fixed strategy for applying these rules (for example, always apply the first applicable rule in the list). The resulting algorithm is terminating: when given a unification problem $\mathcal{P}$, their application yields a unification problem in resolved form $\mathcal{P}'$. If the result is in resolved form, then consider $\sigma_{\mathcal{P}'}$, the unifier associated with $\mathcal{P}'$. We call $\sigma_{\mathcal{P}'}$ the *most general unifier* of the unification problem $\mathcal{P}$ and denote it $\mathsf{mgu}(\mathcal{P})$. If the result $\mathcal{P}'$ is not in resolved form, then there does not exist a unifier for $\mathcal{P}$ and we define $\mathsf{mgu}(\mathcal{P}) = \bot$ and say that $\mathcal{P}$ is not unifiable. If $\mathcal{P} = \{s \doteq t\}$, we denote the most general unifier of $\mathcal{P}$ by $\mathsf{mgu}(s \doteq t)$. For the purpose of unification we treat predicate symbols just like function symbols returning boolean sort, and we treat boolean operations as function symbols with boolean arguments and results; we can therefore write $\mathsf{mgu}(A \doteq B)$ for the most general unifier of literals A and B.

If σ is a substitution in multisorted logic, we write σ^* for the unsorted substitution such that $\sigma^*(x) = \sigma(x)^*$. It follows that $(\sigma(t))^* = \sigma^*(t^*)$ for any term t. For a unification problem $\mathcal{P} = \{s_1 \doteq t_1, \ldots, s_n \doteq t_n\}$, we define $\mathcal{P}^* = \{{s_1}^* \doteq {t_1}^*, \ldots, {s_n}^* \doteq {t_n}^*\}$.

The key observation about multisorted unification with disjoint sorts is the following lemma.

Lemma 8 *Let $\mathcal{P}$ be a multisorted unification problem and $\mathsf{step}(\mathcal{P})$ denote the result of applying one step of the unification algorithm in Figure 5-16. Then $\mathsf{step}(\mathcal{P})^* = \mathsf{step}(\mathcal{P}^*)$ where $\mathsf{step}(\mathcal{P}^*)$ is the result of applying one step of the unification algorithm to the unsorted unification problem $\mathcal{P}^*$. Consequently,*

$$\mathsf{mgu}(\mathcal{P})^* = \mathsf{mgu}(\mathcal{P}^*)$$

In particular, $\mathcal{P}$ is unifiable if and only if $\mathcal{P}^*$ is unifiable.

Resolution
$$\frac{C_1 \vee L_1 \qquad C_2 \vee L_2}{\sigma(C_1) \vee \sigma(C_2)} \qquad \sigma = \mathsf{mgu}(L_1 \doteq \overline{L}_2)$$

Factorisation
$$\frac{C \vee L_1 \vee L_2}{\sigma(C_1) \vee \sigma(L_1)} \qquad \sigma = \mathsf{mgu}(L_1 \doteq L_2)$$

Figure 5-17: Resolution rules

Lemma 8 essentially shows that omitting sorts during unification yields the same result as preserving them. The proof uses the fact that $\doteq$ relates terms or formulas of the same type and that substituting terms with variables of the same type preserves sort constraints.

5.7.2 Multisorted and Unsorted Resolution

We next show that omitting sorts from a set of clauses does not change the set of possible resolution steps, which implies the soundness of omitting sorts.

We consider a finite set $C_1, \ldots, C_n$ of well-sorted clauses. A clause is a disjunction of literals, where a literal is an atomic formula $P(t_1, \ldots, t_n)$ or its negation $\neg P(t_1, \ldots, t_n)$. If A denotes atomic formulas then we define $\overline{A}$ as $\neg A$ and $\overline{\neg A}$ as A. A set $C_1, \ldots, C_n$ is well-sorted if $C_1, \ldots, C_n$ are formulas with free variables in the same multisorted signature, which implies that the same free variable occurring in two distinct clauses $C_i \neq C_j$ has the same sort.

Consider a multisorted clause set $S = \{C_1, \ldots, C_n\}$, and its unsorted counterpart $S^* = \{{C_1}^*, \ldots, {C_n}^*\}$. Consider the resolution procedure rules in Figure 5-17.

Lemma 9 *If $D_0 \in S^*$ is the result of applying the* **Resolution** *rule to ${C_1}^*, {C_2}^* \in S^*$, then D_0 is of the form ${C_0}^*$ where C_0 can be obtained by applying the resolution rule to C_1 and C_2.*

If $D_0 \in S^*$ is the result of applying the **Factoring** *rule to $C^* \in S^*$, then D_0 is of the form ${C_0}^*$ where C_0 can be obtained by applying factoring to C.*

The proof of Lemma 9 follows from Lemma 8: the most general unifier in the multisorted proof step is σ such that σ^* is the most general unifier in the unsorted step.

By induction on the length of the resolution proof, Lemma 9 implies that if an empty clause can be derived from S^*, then an empty clause can be derived from S. By soundness and completeness of resolution in both the unsorted and sorted case and the fact that the skolemization process is isomorphic in the unsorted and multisorted case, we obtain the desired theorem.

Theorem 10 *Let φ be a multisorted formula without equality. If φ^* is valid, so is φ.*

5.8 Completeness of Omitting Sorts

This section continues Section 5.7 and argues that eliminating sort information does not reduce the number of provable formulas. The following lemma is analogous to Lemma 9 and states that resolution steps on multisorted clauses can be performed on the corresponding unsorted clauses.

Lemma 11 *If C_0 is the result of applying the resolution rule to clauses C_1 and C_2, then ${C_0}^*$ can be obtained by applying the resolution rule to clauses ${C_1}^*$ and ${C_2}^*$.*

If C_0 is the result of applying the factoring rule to a clause C, then ${C_0}^*$ can be obtained by applying the factoring rule to clause C^*.

Analogously to Theorem 10 we obtain Theorem 12.

Theorem 12 *Let φ be a many-sorted formula without equality. If φ is valid then so is φ^*.*

5.9 Soundness of Omitting Sorts in Logic with Equality

Sections 5.7 and 5.8 show that in the absence of an interpreted equality symbol there is an isomorphism between proofs in the multisorted and unsorted case. This isomorphism breaks in the presence of equality. Indeed, consider the following clause C:

$$x = y \lor f(x) \neq f(y)$$

expressing injectivity of a function symbol f of type $s_1 \to s_2$ for two disjoint sorts s_1 and s_2. In the unsorted case it is possible to resolve C with itself, yielding

$$x = y \lor f(f(x)) \neq f(f(y))$$

Such a resolution step is, however, impossible in the multisorted case.

In general, eliminating sorts in the presence of equality is unsound, as the conjunction of formulas

$$\forall x{::}\mathsf{obj}.\forall y{::}\mathsf{obj}.x = y$$
$$\exists x{::}\mathsf{int}.y{::}\mathsf{int}.\neg(x = y)$$

shows. In this section we assume that sorts are of the same cardinality, which eliminates such examples without being too restrictive in practice. We then prove Theorem 2 stated in Section 5.3, which implies soundness of omitting sorts even in the presence of an overloaded equality operator. The key step in the proof of Theorem 2 is the construction of a function that maps each multisorted structure $\mathcal{I}$ into an unsorted structure $\mathcal{I}^*$.

We fix a multisorted signature Σ with sorts $s_1, \ldots, s_m$ and denote by Σ^* its unsorted version.

Definition of $\mathcal{I}^*$ and ρ^*. Consider a multisorted structure $\mathcal{I}$ over the signature Σ with m sort interpretations $S_1, ..., S_m$. Because all S_i have equal cardinality, there exists a set S and m functions $f_i : S_i \to S$, for $1 \leq i \leq m$, such that f_i is a bijection between S_i and S. (For example, take S to be one of the S_i.) We let S be the domain of the unsorted model $\mathcal{I}^*$.

We map a multisorted environment ρ into an unsorted environment ρ^* by defining $\rho^*(x) = f_i(\rho(x))$ if x is a variable of sort s_i.

We define a similar transformation for the predicate and function symbols of $\mathcal{I}$. For each predicate P of type $s_{i_1} * \ldots * s_{i_n}$, we let

$$[\![P]\!]^{\mathcal{I}^*} = \{(f_{i_1}(x_1), \ldots, f_{i_n}(x_{i_n})) \mid (x_1, \ldots, x_n) \in [\![P]\!]^{\mathcal{I}}\}$$

Similarly, for each function symbol f of type $s_{i_1} * \ldots * s_{i_n} \to s_{i_{n+1}}$ we let

$$[\![f]\!]^{\mathcal{I}^*} = \{(f_{i_1}(x_1), \ldots, f_{i_{n+1}}(x_{n+1})) \mid (x_1, \ldots, x_{n+1}) \in [\![f]\!]^{\mathcal{I}}\}$$

which is a relation denoting a function because the functions f_i are bijections.

This completes our definition of ρ^* and $\mathcal{I}^*$. We next show that these definitions have the desired properties.

Lemma 13 *If t is a multisorted term of sort s_u, and ρ a multi-sorted environment, then*

$$[\![t^*]\!]^{\mathcal{I}^*}_{\rho^*} = f_u([\![t]\!]^{\mathcal{I}}_{\rho})$$

Proof. Follows from definition, by induction on term t. ■

Proof of Theorem 2. The proof is by induction on φ.

- If φ is $(t_1 = t_2)$ and t_1, t_2 have sort s_u, the claim follows from Lemma 13 by injectivity of f_u.
- If φ is $P(t_1, \ldots, t_n)$ where P is a predicate of type $s_{i_1} * \ldots * s_{i_n}$, we have:

$$\begin{array}{rcl} [\![P(t_1,\ldots,t_n)]\!]^{\mathcal{I}}_{\rho} & = & ([\![t_1]\!]^{\mathcal{I}}_{\rho},\ldots,[\![t_n]\!]^{\mathcal{I}}_{\rho}) \in [\![P]\!]^{\mathcal{I}} \\ \text{(by definition of } \mathcal{I}^* \text{ and } f_i \text{ injectivity)} & = & (f_{i_1}([\![t_1]\!]^{\mathcal{I}}_{\rho}),\ldots,f_{i_n}([\![t_n]\!]^{\mathcal{I}}_{\rho})) \in [\![P]\!]^{\mathcal{I}^*} \\ \text{(by Lemma 13)} & = & ([\![t_1{}^*]\!]^{\mathcal{I}^*}_{\rho^*},\ldots,[\![t_n{}^*]\!]^{\mathcal{I}^*}_{\rho^*}) \in [\![P]\!]^{\mathcal{I}^*} \\ & = & [\![P(t_1,\ldots,t_n)^*]\!]^{\mathcal{I}^*}_{\rho^*} \end{array}$$

- The cases $\varphi = \varphi_1 \land \varphi_2$ and $\varphi = \neg\varphi_1$ follow directly from the induction hypothesis.
- $\varphi = \exists z{::}s_v.\varphi_0$.

  $\Longrightarrow$ Assume $[\![\varphi]\!]^{\mathcal{I}}_{\rho}$ is true. Then, there exists an element e of the sort s_v such that $[\![\varphi_0]\!]^{\mathcal{I}}_{\rho[z\mapsto e]}$ is true. By induction, $[\![\varphi_0{}^*]\!]^{\mathcal{I}^*}_{(\rho[z\mapsto e])^*}$ is true. Because $(\rho[z \mapsto e])^* = \rho^*[z \mapsto f_v(e)]$, we have $[\![\exists z.\varphi_0{}^*]\!]^{\mathcal{I}^*}_{\rho^*}$.

  $\Longleftarrow$ Assume $[\![\varphi^*]\!]^{\mathcal{I}^*}_{\rho^*}$. Then, there exists $e \in S$ of $\mathcal{I}^*$ such that $[\![\varphi_0{}^*]\!]^{\mathcal{I}^*}_{\rho^*[z\mapsto e]}$. Let $\rho_0 = \rho[z \mapsto f_v^{-1}(e)]$. Then $\rho_0{}^* = \rho^*[z \mapsto e]$. By the induction hypothesis, $[\![\varphi_0]\!]^{\mathcal{I}}_{\rho_0}$, so $[\![\exists z{::}s_v.\varphi_0]\!]^{\mathcal{I}}_{\rho}$.

5.10 Sort Information and Proof Length

Theorem 2 shows that omitting sort information is sound for disjoint sorts of the same cardinality. Moreover, experimental results in Section 5.3 show that omitting sorts is often beneficial compared to the standard relativization encoding of sorts using unary predicates [177, Chapter 6, Section 8], even for SPASS [248] that has built-in support for sorts. While there may be many factors that contribute to this empirical fact, we have observed that in most cases *omitting sort information decreases the size of the proofs found by the prover.*

We next sketch an argument that, in the simple settings without the paramodulation rule [199], removing unary sort predicates only decreases the length of resolution proofs. Let $P_1, \ldots, P_n$ be unary sort predicates. We use the term *sort literal* to denote a literal of the form $P_i(t)$ or $\neg P_i(t)$ for some $1 \le i \le n$. The basic idea is that we can map clauses with sort predicates into clauses without sort predicates, while mapping resolution proofs into correct new proofs. We denote this mapping α. The mapping α removes sort literals and potentially some additional non-sort literals, and potentially performs generalization. It therefore maps each clause into a stronger clause.

Consider mapping an application of a resolution step to clauses C_1 and C_2 with resolved literals L_1 and L_2 to obtain a resolvent clause C. If L_1 and L_2 are not sort literals, we can perform the analogous resolution step on the result of removing sort literals from C_1 and C_2. If, on the other hand, L_1 and L_2 are sort literals, then $\alpha(C_1)$ and $\alpha(C_2)$ do not contain L_1 or L_2. We map such a proof step into a trivial proof step that simply selects as $\alpha(C)$ one of the premises $\alpha(C_1)$ or $\alpha(C_2)$. For concreteness, let $\alpha(C) = \alpha(C_1)$. Because C in this case contains an instance of each non-sort literal from C_1, we have that $\alpha(C)$ is a generalization of a subset of literals of C. The mapping α works in an analogous way for the factoring step in a resolution proof, mapping it either to an analogous factoring step or a trivial proof step.

The trivial proof steps are the reason why α removes not only sort literals but also non-sort literals. Because α removes non-sort literals as well, even some proof steps involving non-sort literals may become inapplicable. However, they can all be replaced by trivial proof steps. The resulting proof tree has the same height and terminates at an empty clause, because α maps each clause into a stronger one. Moreover, trivial proof steps can be removed, potentially reducing the height of the tree. This shows that the shortest resolution proof without guards is the same or shorter than the shortest resolution proof with guards.

5.11 Related Work

We are not aware of any other system capable of verifying such strong properties of operations on data structures that use arrays, recursive memory cells and integer keys and does not require interactive theorem proving.

Verification systems. Boogie [29] is a sound verification system for the Spec# language, which extends C# with specification constructs and introduces a particular methodology for ensuring sound modular reasoning in the presence of aliasing and object-oriented features. This methodology creates potentially more difficult frame conditions when analyzing procedure calls compared to the ones created in Jahob, but the correctness of this methodology seems easier to establish.

ESC/Java 2 [61] is a verification system for Java that uses JML [166] as a specification language. It supports a large set of Java features and sacrifices soundness to achieve higher usability for common verification tasks.

Boogie and ESC/Java2 use Nelson-Oppen style theorem provers [82, 23, 30], which have potentially better support for arithmetic, but have more difficulties dealing with quantified invariants. Jahob also supports a prototype SMT-LIB interface to Nelson-Oppen style theorem provers. Our preliminary experience suggests that, for programs and properties described in this chapter, resolution-based theorem provers are no worse than current Nelson-Oppen style theorem provers. Combining these two theorem proving approaches is an active area of research [16, 213], and our system could also take advantage of these ideas, potentially resulting in more robust support for arithmetic reasoning.

Specification variables are present in Boogie [169] and ESC/Java2 [60] under the name *model fields*. We are not aware of any results on non-interactive verification that data structures such as trees and hash tables meet their specifications expressed in terms of model fields. The properties we are reporting on have previously been verified only interactively [117, 124, 148, 267].

The Krakatoa tool [178] can verify JML specifications of Java code. We are not aware of its use to verify data structures in an automated way.

Abstract interpretation. Shape analyses [226, 167, 224] typically verify weaker properties than in our examples. The TVLA system [173] has been used to verify insertion sort and bubble sort [174] as well as to verify implementations of insertion and removal operations on sets implemented as mutable lists and binary search trees [218, Page 35]. The approach [218] uses manually supplied predicates and transfer functions and axioms for the analysis, but is able to infer loop invariants in an imperative implementation of trees. Our implementation of trees is functional and uses recursion, which simplifies the verification. The analysis I describe in this dissertation does not infer loop invariants, but does not require transfer functions to be specified either. The only information that the data structure user needs to trust is that procedure contracts correctly formalize the desired behavior of data structure operations; if the developer incorrectly specifies an invariant or an update to a specification variable, the system will detect an error.

Translation from higher-order to first-order logic. In [126, 188, 187] the authors also address the process of proving higher-order formulas using first-order theorem provers. Our work differs in that we do not aim to provide automation to a general-purpose higher-order interactive theorem prover. Therefore, we were able to avoid using general encoding of lambda calculus into first-order logic and we believe that this made our translation more effective.

The authors in [126, 187] also observe that encoding the full type information slows down the proof process. The authors therefore omit type information and then check the resulting proofs for soundness. A similar approach was adopted to encoding multi-sorted logic in the Athena theorem proving framework [15]. In contrast, we were able to prove that omitting sort information preserves soundness and completeness when sorts are disjoint and have the same cardinality.

The filtering of assumptions also appears in [186]. Our technique is similar but simpler and works before transformation to clause normal form. Our results confirm the usefulness of assumption filtering in the context of problems arising in data structure verification.

A quantifier-free language that contains set operations can be translated into the universal fragment of first-order logic [160] (see also [68]). In our experience so far we found no need to limit the precision of the translation by restricting ourselves to the universal fragment.

Type systems. Type systems have been used to verify algebraic data types [76], array bounds [257], and mutable structures [269], usually enforcing weaker properties than in our case. Recently, researchers have developed a promising approach [198] based on separation logic [130] that can verify shape and content properties of imperative recursive data structures (although it has not been applied to hash tables yet). Our approach uses standard higher-order and first-order logic and seems conceptually simpler, but generates proof obligations that have potentially more quantifiers and case analyses.

Constraint solving and testing. In [80] the authors use a constraint solver based on translation to propositional logic to identify all errors within a given scope in real-world implementations of linked lists. Another approach for finding bugs is exhaustive testing by generating tests that satisfy given preconditions [179, 140]. These techniques are very effective at finding bugs, but do not guarantee the absence of errors.

5.12 Conclusions

We presented a technique for verifying complex data structure properties using resolution-based first-order theorem provers. We used a simple translation that expands higher-order definitions and translates function applications to applications of uninterpreted function symbols, without encoding set theory or lambda calculus in first-order logic. We have observed that omitting sort information in our translation speeds up the theorem proving process. This motivated us to prove that omitting such sort information is sound for disjoint sorts of same cardinality, even in the presence of an overloaded equality operator. We have also confirmed the usefulness of filtering to reduce the size of formulas used as an input to the theorem prover.

Using these techniques we were able to prove strong properties for an implementation of a hash table, an instantiable mutable list, for a functional implementation of ordered binary search tree, and for a functional association list. We also verified a simple library system that instantiates two sets and a relation and maintains constraints on them in the presence of changes to the sets and relation. Our system proves that operations act as expected on the abstract content of the data structure (that is, they establish their postcondition such as insertion or removal of tuples from a relation), that they preserve non-trivial internal representation invariants, such as sortedness of a tree and structural invariants of a hash table, and that they do not cause run-time errors such as null dereference or array out of bounds access.

Chapter 6

Field Constraints and Monadic Second-Order Logic for Reachability

This chapter studies a technique for proving formulas that arise from verification of linked imperative data structures and is based on [253, 252]. I present the examples in this chapter using Jahob notation, although we originally verified them using the Hob system [152, 165] before incorporating field constraint analysis into Jahob. Field constraints analysis is used in the Bohne shape analysis plugin developed by Thomas Wies [254, 211, 251]. I omit detailed discussion of Bohne; loop invariant inference is outside the scope of this dissertation.

The verification of imperative linked data structures is an active area of research [191, Chapter 4], [106, 226, 190, 21, 72, 185, 101, 167]. The starting point for our technique is the decidability of monadic second-order logic of trees [235]. Among the shape analysis approaches, techniques based on monadic second-order logic of trees [190, 144, 134] are interesting for several reasons. First, the correctness of such analysis is easier to establish than for approaches based on *ad hoc* representations; the use of a decidable logic separates the problem of generating constraints that imply program properties from the problem of solving these constraints. Next, such analyses can be used in the context of assume-guarantee reasoning because the logic provides a language for specifying the behaviors of code fragments. Finally, the decidability of the logic leads to completeness properties for these analyses, eliminating false alarms and making the analyses easier to interact with. We were able to confirm these observations in the context of Hob system [152] for analyzing data structure consistency, where we have integrated one such tool [190] with other analyses, allowing us to use shape analysis in the context of larger programs: in particular, Hob enabled us to leverage the power of shape analysis, while avoiding the associated performance penalty, by applying shape analysis only to those parts of the program where its extreme precision is necessary.

Our experience with such analyses has also taught us that some of the techniques that make these analyses predictable also make them inapplicable to many useful data structures. Among the most striking examples is the restriction on pointer fields in the Pointer Assertion Logic Engine [190]. This restriction states that all fields of the data structure that are not part of the data structure's tree backbone must be functionally determined by the backbone; that is, such fields must be specified by a formula that uniquely determines where they point to. Formally, we have

$$\forall x\ y.\ f(x){=}y \ \leftrightarrow\ F(x,y) \tag{6.1}$$

where f is a function representing the field, and F is a formula mentioning only backbone fields. The restriction that F is functional means that, although data structures such as doubly linked lists with backward pointers can be verified, many other data structures remain beyond the scope of the analysis. This includes data structures where the exact value of pointer fields depends on the history of data structure operations, and data structures that use randomness to achieve good average-case performance, such as skip lists [214]. In such cases, the invariant on the pointer field does not uniquely determine where the field points to, but merely gives a constraint on the field, of the form

$$\forall x\ y.\ f(x){=}y \ \rightarrow\ F(x,y) \tag{6.2}$$

This constraint is equivalent to $\forall x.\ F(x, f(x))$, which states that the function f is a solution of a given binary predicate. The motivation for this chapter is to find a technique that supports reasoning about constraints of this, more general, form. In a search for existing approaches, we have considered structure simulation [129, 127], which, intuitively, allows richer logics to be embedded into existing logics that are known to be decidable, and of which [190] can be viewed as a specific instance. Unfortunately, even the general structure simulation requires definitions of the form $\forall x\ y.\ r(x,y) \ \leftrightarrow\ F(x,y)$ where $r(x,y)$ is the relation being simulated. When the relation $r(x,y)$ is a function, which is the case with most reference fields in programming languages, structure simulation implies the same restriction on the functionality of the defining relation. To handle the general case, an alternative approach therefore appears to be necessary.

Field constraint analysis. This chapter presents field constraint analysis, our approach for analyzing fields with general constraints of the form (6.2). Field constraint analysis is a proper generalization of the existing approach and reduces to it when the constraint formula F is functional. It is based on approximating the occurrences of f with F, taking into account the polarity of f, and is always sound. It is expressive enough to verify constraints on pointers in data structures such as two-level skip lists. The applicability of our field constraint analysis to non-deterministic field constraints is important because many complex properties have useful non-deterministic approximations. Yet despite this fundamentally approximate nature of field constraints, we were able to prove its completeness for some important special cases. Field constraint analysis naturally combines with structure simulation, as well as with a symbolic approach to shape analysis [251, 211]. Our presentation and current implementation are in the context of the monadic second-order logic (MSOL) of trees [143], but our results extend to other logics. We therefore view field constraint analysis as a useful component of shape analysis approaches that makes shape analysis applicable to a wider range of data structures.

Contributions. This chapter makes the following contributions:

- We introduce an **algorithm** (Figure 6-7) that uses field constraints to eliminate derived fields from verification conditions.
- We prove that the algorithm is both **sound** (Theorem 17) and, in certain cases, **complete**. The completeness applies not only to deterministic fields (Theorem 19), but also to the preservation of field constraints themselves over loop-free code (Theorem 25). The last result implies a complete technique for checking that field con-

straints hold, if the programmer adheres to a discipline of maintaining them e.g. at the beginning of each loop.

6.1 Examples

We next explain our field constraint analysis with a set of examples. The doubly-linked list example shows that our analysis handles, as a special case, the ubiquitous back pointers of data structures. The skip list example shows how field constraint analysis handles non-deterministic field constraints on derived fields, and how it can infer loop invariants. Finally, the students example illustrates inter-data-structure constraints, which are simple but useful for high-level application properties.

6.1.1 Doubly-Linked List with an Iterator

This section presents a doubly-linked list with a built-in iterator. It illustrates the usefulness of field constraints for specifying pointers that form doubly-linked structures.

Our doubly-linked list implementation is a global data structure with operations `add` and `remove` that insert and remove elements from the list, as well as operations `initIter`, `nextIter`, and `lastIter` for manipulating the iterator built into the list. We have verified all these operations using our system; we here present only the `remove` operation. We implement the doubly-linked list using two fields of the `Node` class, `next` and `prev`, the private `first` variable of the doubly-linked list, and the private `current` variable that indicates the position of the iterator in the list. The contract for the `remove` operation in Figure 6-1 specifies the behavior of the operation `remove` using two sets: `content` contains the set of elements in the list, whereas `iterRest` specifies the set of elements that remain to be iterated over. These two sets abstractly characterize the behavior of operations, allowing the clients to soundly reason about the hidden implementation of the list. Such reasoning within clients is sound because our analysis verifies that the implementation conforms to the specification, using the definitions of sets `content` and `iterRest` in Figure 6-1. The definitions of sets are expressed in our higher-order logic subset and contain reachability expressions. The shorthand `reach` denotes binary reachability relation defined as the reflexive transitive closure of the `Node.next` field and is denoted using the `rtrancl` symbol of our logic. The class then defines `content` as the set of all objects reachable from `root` and defines `iterRest` as the set of all objects reachable from `current`.

The definitions of `content` and `iterRest` are not visible to clients of `DLLIter` class, allowing changes to the implementation without necessarily requiring changes to clients. What is visible to clients are the values of `content` and `iterRest` variables, so it is desirable for the class to specify a public class invariant `content` $\subseteq$ `iterRest` that relates them.

The `DLLIter` class also contains private representation invariants; our system ensures that these invariants are maintained by each operation in the module. The first invariant states that nodes outside the data structure content have have `null` as the value of the `next` field. Our system can prove that the entire program preserves such invariant because the fields of the `Node` class are marked as belonging to `DLLIter` class, which makes these fields inaccessible to classes other than `DLLIter` (although the other classes can pass around `Node` objects).

The second invariant, `tree[Node.next]`, indicates that `next` fields form a tree. Jahob uses such `tree` invariants to identify the backbone fields of the data structure. In this case, `next` as the sole kind of the backbone field, so the tree reduces to a list, which enables Jahob to speed up the verification by using the string mode of the MONA tool [143] as opposed to the more general tree mode.

Jahob recognizes the form of the final invariant as a field constraint on the `prev` field. The field constraint indicates to Jahob that `prev` is a derived field that is the inverse of the `next` field.

Jahob verifies that the `remove` procedure implementation in Figure 6-1 conforms to its contract as follows. Jahob expands the modifies clause into a frame condition, which it conjoins with the ensures clause. Next, Jahob conjoins the public and private invariants to the requires and ensures clauses, converts the program into a guarded command language, and generates verification condition taking into account the definitions of variables `content` and `iterRest`.

To decide the resulting verification condition, Jahob analyzes the verification condition and recognizes the assumptions of the form `tree`$[f_1, \ldots, f_n]$ as well as of the form `ALL` $x\, y.\, f\, x = y \longrightarrow F(x, y)$. The first form of the assumption determines the backbone fields of the data structure, the second form of an assumption determines the derived fields and their constraint. In our case, Jahob exploits the fact that `next` is a backbone field and `prev` is a field given by a field constraint to reduce the verification condition to one expressible using only the `next` field. (This elimination is given by the algorithm in Figure 6-7.) Because `next` fields form a tree, Jahob can decide the verification condition using monadic second-order logic on trees [143]. To ensure the soundness of this approach, Jahob also verifies that the structure remains a tree after each operation.

We note that our first implementation of the doubly-linked list with an iterator was verified using a Hob plugin [152] that relies on Pointer Assertion Logic Engine tool [190]. While verifying the initial version of the list module, we discovered an error in `remove`: the first three lines of `remove` procedure implementation in Figure 6-1 were not present, resulting in a violation of the specification of `remove` in the special case when the element being removed is the next element to iterate over. What distinguishes our approach from the previous Hob analysis based on PALE is the ability to handle the cases where the field constraints are non-deterministic. We illustrate such cases in the examples that follow.

Compared to our deployment of field constraint analysis in Hob, the deployment in Jahob also has the advantage of being applicable to procedures that require a combination of reasoning based on decision procedures other than monadic second-order logic over trees. We remark that Thomas Wies integrated field constraint analysis with symbolic shape analysis [251] that infers loop invariants for linked data structures; we omit the discussion of loop invariant inference from this thesis and direct the reader to [252, 253, 254].

6.1.2 Skip List

We next present the analysis of a two-level skip list. Skip lists [214] support logarithmic average-time access to elements by augmenting a sorted linked list with sublists that skip over some of the elements in the list. The two-level skip list is a simplified implementation of a skip list, which has only two levels: the list containing all elements, and a sublist of this list. Figure 6-3 presents an example two-level skip list. Our implementation uses the `next` field to represent the main list, which forms the backbone of the data structure, and uses the derived `nextSub` field to represent a sublist of the main list. We focus on the `add` procedure, which inserts an element into an appropriate position in the skip list. The implementation of `add` first searches through `nextSub` links to get an estimate of the position of the entry, then finds the entry by searching through `next` links, and inserts the element into the main

```
class Node {
    public /*: claimedby DLLIter */ Node next;
    public /*: claimedby DLLIter */ Node prev;
}
class DLLIter {
   private static Node first, current;
   /*:
     public static specvar content :: objset;
     public static specvar iterRest :: objset;

     private static specvar reach :: "obj => obj => bool";
     vardefs "reach == (% a b. (a,b) : rtrancl {(x, y). Node.next x = y})";

     private vardefs "content == {x. x ~= null & reach first x}";
     private vardefs "iterRest == {x. x ~= null & reach current x}";

     public invariant "iterRest \<subseteq> content";

     invariant "ALL n. n ~: content --> n..Node.next = null";

     invariant "tree [Node.next]";
     invariant "ALL x y. Node.prev x = y -->
                         (y ~= null --> Node.next y = x) &
                         (y = null & x ~= null --> (ALL z. Node.next z ~= x))";
   */

   public static void remove(Node n)
      /*: requires "n : content"
          modifies content, iterRest
          ensures "content = old content - {n} &
                   iterRest = old iterRest - {n}"
      */
   {
      if (n==current) {
         current = current.next;
      }
      if (n==first) {
         first = first.next;
      } else {
         n.prev.next = n.next;
      }
      if (n.next != null) {
         n.next.prev = n.prev;
      }
      n.next = null; n.prev = null;
   }
}
```

Figure 6-1: Iterable list implementation and specification

next-linked list. With certain probability, the procedure also inserts the element into the nextSub list. The postcondition of add indicates that add always inserts the element into the content set of elements, which denotes the elements stored in the list. The class defines content as the set of nodes reachable from root along the next field.

The Skiplist class defines several class invariants. The first invariant, tree[Node.next], defines next as a tree backbone field. The second invariant is a field constraint on nextSub. The field constraint indicates that, for non-null objects, the nextSub field points from x to an object y that is reachable along the next field from the successor of x. Note that the field constraint on nextSub is non-deterministic, because it only states there exists a path of length at least one from x to y, without indicating where exactly in the list nextSub points. Indeed, the simplicity of the skip list implementation stems from the fact that the position of nextSub is not uniquely given by next; it depends not only on the history of invocations, but also on the random number generator used to decide when to introduce new nextSub links. The ability to support such non-deterministic constraints is what distinguishes our approach from approaches that can only handle deterministic fields.

Our analysis successfully verifies that add preserves all invariants, including the non-deterministic field constraint on nextSub. While doing so, the analysis takes advantage of these invariants as well, as is usual in assume/guarantee reasoning. In this example, the analysis also infers the loop invariants in add, using symbolic shape analysis [251, 211, 254].

6.1.3 Students and Schools

Our next example illustrates the power of non-deterministic field constraints. This example contains two linked lists: one containing students and one containing schools. Figure 6-4 shows an example instance of this data structure. Each Elem object may represent either a student or a school. Both students and schools use the next backbone pointer to indicate the next student or school in the relevant linked list. In addition, students have an attends field referencing the school which they attend (the attends field is null for school objects).

Figures 6-5 shows the data structure declaration for this example and presents the addStudent operation on the data structure. The specification of the data structure is given in terms of two sets: ST denotes the students in the data structure and SC denotes the schools. The representation invariants formalize the shape of the data structure: the public invariants ensure that students and schools are disjoint sets that do not contain null, whereas the three private invariants specify more detailed properties of next and attends fields. The tree invariant indicates that the data structure contains a union of two lists. The subsequent invariant indicates that the elements outside the list of students and schools have no incoming or outgoing references.

The final invariant is a field constraint on the attends field: it indicates that an attends field points from the students in the ST set to one of the schools in SC set. Note that the field constraint is also non-deterministic in this example: the target of the attends field is not determined by the next fields; instead, attends field carries important non-redundant information about the data structure. Nevertheless, the attends field is not entirely unconstrained: an attends field pointing from a student to another student would indicate a broken data structure.

Our analysis successfully verifies that the addStudent operation preserves the data structure invariants and correctly inserts a student into the list of students. To preserve the field constraint x:ST --> y:SC on the attends field, the analysis uses the precondition of addStudent to conclude that the supplied school is already in the list of schools. Veri-

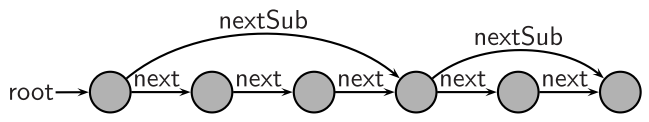


Figure 6-2: An instance of a two-level skip list

```
class Skiplist {
    private static Node root;
    /*:
      public static specvar content :: objset;
      private static specvar reach :: "obj => obj => bool";
      vardefs "reach == (% a b. rtrancl_pt (% x y. x..Node.next = y) a b)";
      private vardefs "content == {x. reach root x & x ~= null}";

      invariant "tree [Node.next]";
      invariant "ALL x y. Node.nextSub x = y --> ((x = null --> y = null)
                            & (x ~= null --> reach (Node.next x) y))";
      invariant "ALL x. x ~= null & ~(reach root x) -->
                           Node.next x = null &
                           (ALL y. y ~= null --> Node.next y ~= x)";
    */

   public static void add(Node e)
   /*: requires "e ~= null & e ~: content"
       modifies content
       ensures "content = old content Un {e}" */
    {
        if (root==null) {
            root = e;
        } else {
            int v = e.v;
            Node sprev = root,  scurrent = root.nextSub;
            while ((scurrent != null) && (scurrent.v < v)) {
                sprev = scurrent;  scurrent = scurrent.nextSub;
            }
            Node prev = sprev, current = sprev.next;
            while ((current != scurrent) && (current.v < v)) {
                prev = current;  current = current.next;
            }
            e.next = current;  prev.next = e;
            if (randomBoolean()) {
                sprev.nextSub = e;  e.nextSub = scurrent;
            } else {
                e.nextSub = null;
            }
        }
    }
}
```

Figure 6-3: Two-level skip-list with add operation

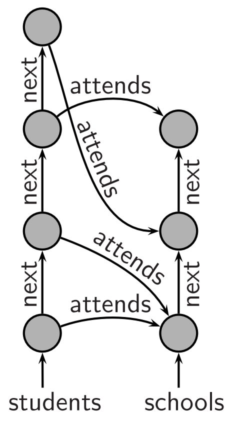


Figure 6-4: Data structure instance for students and schools example

fying such non-deterministic constraints is beyond the power of previous analyses based on monadic second-order logic over trees, which required the identity of the school pointed to by the student to be functionally determined by the identity of the student. The example therefore illustrates how our analysis eliminates a key restriction of previous approaches.

6.2 Field Constraint Analysis

This section presents the field constraint analysis algorithm and proves its soundness as well as, for some important cases, completeness.

We consider a logic $\mathcal{L}$ over a signature Σ where Σ consists of unary function symbols $f \in \mathsf{Fld}$ corresponding to fields in data structures and constant symbols $c \in \mathsf{Var}$ corresponding to program variables. We use monadic second-order logic (MSOL) over trees as our working example, but in general we only require $\mathcal{L}$ to support conjunction, implication and equality reasoning.

A Σ-structure S is a first-order interpretation of symbols in Σ. For a formula F in $\mathcal{L}$, we denote by $\mathsf{Fields}(F) \subseteq \Sigma$ the set of all fields occurring in F.

We assume that $\mathcal{L}$ is decidable over some set of well-formed structures and we assume that this set of structures is expressible by a formula I in $\mathcal{L}$. We call I the *simulation invariant* [129]. For simplicity, we consider the simulation itself to be given by the restriction of a structure to the fields in $\mathsf{Fields}(I)$, i.e. we assume that there exists a decision procedure for checking validity of implications of the form $I \rightarrow F$ where F is a formula such that $\mathsf{Fields}(F) \subseteq \mathsf{Fields}(I)$. In our running example, MSOL, the simulation invariant I states that the fields in $\mathsf{Fields}(I)$ span a forest.

We call a field $f \in \mathsf{Fields}(I)$ a *backbone field*, and call a field $f \in \mathsf{Fld} \setminus \mathsf{Fields}(I)$ a *derived field*. We refer to the decision procedure for formulas with fields in $\mathsf{Fields}(I)$ over the set of structures defined by the simulation invariant I as *the underlying decision procedure*. Field constraint analysis enables the use of the underlying decision procedure to reason about non-deterministically constrained derived fields. We state invariants on the derived fields using field constraints.

Definition 14 (Field constraints on derived fields) *A* field constraint D_f *for a simu-*

```
class Elem {
    public /*: claimedby Students */ Elem attends;
    public /*: claimedby Students */ Elem next;
}
class Students {
    private static Elem students;
    private static Elem schools;

    /*:
      private static specvar reach :: "obj => obj => bool";
      vardefs "reach == (% a b. rtrancl_pt (% x y. x..Elem.next = y) a b)";

      public static specvar ST :: objset;
      vardefs "ST == {x. x ~= null & reach students x}";
      public static specvar SC :: objset;
      vardefs "SC == {x. x ~= null & reach schools x}";

      public invariant "null ~: (ST Un SC)";
      public invariant "ST Int SC = {}";

      invariant "tree [Elem.next]";
      invariant "ALL x. x ~: (ST Un SC Un {null}) -->
                        (ALL y. y ~= null --> y..Elem.next~=x) &
                        x..Elem.next=null";

      invariant "ALL x y. Elem.attends x = y -->
                     (x : ST --> y : SC) &
                     (x ~: ST --> y = null)";
     */

    public static void addStudent(Elem st, Elem sc)
    /*:
      requires "st ~: (ST Un SC Un {null}) & sc : SC"
      modifies ST
      ensures "ST = old ST Un {st}"
     */
    {
        st.attends = sc;
        st.next = students;
        students = st;
    }
}
```

Figure 6-5: Students example

lation invariant I and a derived field f is a formula of the form

$$\mathsf{D}_f \;\equiv\; \forall x\, y.\; f(x) = y \;\rightarrow\; \mathsf{FC}_f(x, y)$$

where FC_f is a formula with two free variables such that (1) $\mathsf{Fields}(\mathsf{FC}_f) \subseteq \mathsf{Fields}(I)$, and (2) FC_f is total with respect to I, i.e. $I \models \forall x.\; \exists\, y\,.\, \mathsf{FC}_f(x, y)$.

We call the constraint D_f deterministic *if FC_f is deterministic with respect to I, i.e.*

$$I \models \forall x\, y\, z.\; \mathsf{FC}_f(x, y) \wedge \mathsf{FC}_f(x, z) \;\rightarrow\; y = z \;\; .$$

We write D for the conjunction of D_f for all derived fields f.

Note that Definition 14 covers arbitrary constraints on a field, because D_f is equivalent to $\forall x.\; \mathsf{FC}_f(x, f(x))$.

The totality condition (2) is not required for the soundness of our approach, only for its completeness, and rules out invariants equivalent to “false”. The condition (2) does not involve derived fields and can therefore be checked automatically using a single call to the underlying decision procedure.

Our goal is to check validity of formulas of the form $I \wedge D \;\rightarrow\; G$, where G is a formula with possible occurrences of derived fields. If G does not contain any derived fields then there is nothing to do, because in that case checking validity immediately reduces to the validity problem without field constraints, as given by the following lemma.

Lemma 15 *Let G be a formula such that $\mathsf{Fields}(G) \subseteq \mathsf{Fields}(I)$.*
Then $I \models G$ iff $I \wedge D \models G$.

Proof. The left-to-right direction follows immediately. For the right-to-left direction assume that $I \wedge D \;\rightarrow\; G$ is valid. Let S be a structure such that $S \models I$. By totality of all field constraints in D there exists a structure S' such that $S' \models I \wedge D$ and S' differs from S only in the interpretation of derived fields. Since $\mathsf{Fields}(G) \subseteq \mathsf{Fields}(I)$ and I contains no derived fields we have that $S' \models G$ implies $S \models G$. ∎

To check validity of $I \wedge D \;\rightarrow\; G$, we therefore proceed as follows. We first obtain a formula G' from G by eliminating all occurrences of derived fields in G. Next, we check validity of G' with respect to I. In the case of a derived field f that is defined by a deterministic field constraint, occurrences of f in G can be eliminated by flattening the formula and substituting each term $f(x) = y$ by $\mathsf{FC}_f(x, y)$. However, in the general case of non-deterministic field constraints such a substitution is only sound for negative occurrences of derived fields, since the field constraint gives an over-approximation of the derived field. Therefore, a more sophisticated elimination algorithm is needed.

Eliminating derived fields. Figure 6-7 presents our algorithm Elim for elimination of derived fields. Consider a derived field f and let $F \equiv \mathsf{FC}_f$. The basic idea of Elim is that we can replace an occurrence $f(x)$ in a formula $G(f(x))$ by a new variable y that satisfies $F(x, y)$, yielding a stronger formula $\forall y.\, F(x, y) \;\rightarrow\; G(y)$. As an improvement, if G contains two occurrences $f(x_1)$ and $f(x_2)$, and if x_1 and x_2 evaluate to the same value, then the algorithm imposes the constraint that the values y_1 and y_2 replacing $f(x_1)$ and $f(x_2)$ are equal.

Define function skolem as follows: 1) $\mathsf{skolem}(\forall x.G) = G$; 2) $\mathsf{skolem}(G_1 \wedge G_2) = \mathsf{skolem}(G_1) \wedge \mathsf{skolem}(G_2)$; and 3) $\mathsf{skolem}(G) = G$ if G is not of the form $\forall x.G$ or $G_1 \wedge G_2$.

$$\mathsf{D}_{nextSub} \equiv \forall v_1\, v_2.\ nextSub(v_1) = v_2 \ \rightarrow\ next^+(v_1, v_2)$$

$$\begin{aligned}
G &\equiv \mathsf{wlp}((e.nextSub := root.nextSub\,\mathbf{;}\ e.next := root), \mathsf{D}_{nextSub}) \\
&\equiv \forall v_1\, v_2.\ \ nextSub[e := nextSub(root)](v_1) = v_2 \ \ \rightarrow\ (next[e := root])^+(v_1, v_2)
\end{aligned}$$

$$\begin{aligned}
G' \equiv\ & \mathsf{skolem}(\mathsf{Elim}(G)) \equiv \\
& x_1 = root \ \rightarrow\ next^+(x_1, y_1) \ \rightarrow \\
& \qquad x_2 = v_1 \ \rightarrow\ next^+[e := y_1](x_2, y_2) \wedge (x_2 = x_1 \ \rightarrow\ y_2 = y_1) \ \rightarrow \\
& \qquad\qquad\qquad y_2 = v_2 \ \rightarrow\ (next[e := root])^+(v_1, v_2)
\end{aligned}$$

Figure 6-6: Elimination of derived fields from a pretty nice formula. The notation $next^+$ denotes the irreflexive transitive closure of predicate $next(x) = y$.

Example 16 The example in Figure 6-6 demonstrates the elimination of derived fields using the Elim algorithm. The example in Figure 6-6 is inspired by the skip list module from Section 6.1. The formula G expresses the preservation of field constraint $\mathsf{D}_{nextSub}$ for updates of fields *next* and *nextSub* that insert e in front of *root*. This formula is valid under the assumption that $\forall x.\ next(x) \neq e$ holds. Elim first replaces the inner occurrence $nextSub(root)$ and then the outer occurrence of *nextSub*. Theorem 25 implies that the resulting formula $\mathsf{skolem}(\mathsf{Elim}(G))$ is valid under the same assumption as the original formula G.

♦

The Elim algorithm uses the set K of triples (x, f, y) to record previously assigned values for $f(x)$. Elim runs in time $O(n^2)$ where n is the size of the formula and produces an at most quadratically larger formula. Elim accepts formulas in negation normal form, where all negation signs apply to atomic formulas (see Figure 4-8 for rules of transformation into negation normal form). We generally assume that each quantifier $Q\, z$ binds a variable z that is distinct from other bound variables and distinct from the free variables of the entire formula. We present Elim as acting on potentially quantified formulas, but Elim is also useful for checking validity of quantifier-free formulas because it only introduces universal quantifiers which can be replaced by Skolem constants. The algorithm is also applicable to multisorted logics, and, by treating sets of elements as a new sort, to MSOL. To make the discussion simpler, we consider a deterministic version of Elim where the non-deterministic choices of variables and terms are resolved by some arbitrary, but fixed, linear ordering on terms. We write $\mathsf{Elim}(G)$ to denote the result of applying Elim to a formula G.

The correctness of Elim is given by Theorem 17. The proof of Theorem 17 relies on the monotonicity of logical operations and quantifiers in negation normal form of a formula.

Theorem 17 (Soundness) *The algorithm* Elim *is sound: if* $I \wedge D \models \mathsf{Elim}(G)$*, then* $I \wedge D \models G$*. What is more,* $I \wedge D \wedge \mathsf{Elim}(G) \models G$*.*

Proof. By induction on the first argument G of elim we prove that, for all finite K,

$$I \wedge D \wedge \mathsf{elim}(G, K) \wedge \bigwedge_{(x_i, f_i, y_i) \in K} \mathsf{FC}_{f_i}(x_i, y_i) \models G$$

S	$-$	a term or a formula
$\mathsf{Terms}(S)$	$-$	terms occurring in S
$\mathsf{FV}(S)$	$-$	variables free in S
$\mathsf{Ground}(S)$	$=$	$\{t \in \mathsf{Terms}(S).\ \mathsf{FV}(t) \subseteq \mathsf{FV}(S)\}$
$\mathsf{Derived}(S)$	$-$	derived function symbols in S

```
proc $\mathsf{Elim}(G) = \mathsf{elim}(G, \emptyset)$
proc $\mathsf{elim}(G$ : formula in negation normal form;
             $K$ : set of (variable,field,variable) triples):
  let $T = \{f(t) \in \mathsf{Ground}(G).\ f \in \mathsf{Derived}(G) \wedge \mathsf{Derived}(t) = \emptyset\}$
  if $T \neq \emptyset$ do
      choose $f(t) \in T$
      choose $x, y$ fresh first-order variables
      let $F = \mathsf{FC}_f$
      let $F_1 = F(x,y) \wedge \bigwedge_{(x_i,f,y_i) \in K} (x = x_i \ \rightarrow \ y = y_i)$
      let $G_1 = G[f(t) := y]$
      return $\forall x.\ x = t \ \rightarrow \ \forall y.\ (F_1 \ \rightarrow \ \mathsf{elim}(G_1, K \cup \{(x,f,y)\}))$
  else case $G$ of
   |   $Qx.\ G_1$ where $Q \in \{\forall, \exists\}$:
         return $Qx.\ \mathsf{elim}(G_1, K)$
   |   $G_1\ op\ G_2$ where $op \in \{\wedge, \vee\}$:
         return $\mathsf{elim}(G_1, K)\ op\ \mathsf{elim}(G_2, K)$
   |   else return $G$
```

Figure 6-7: Derived-field elimination algorithm

For $K = \emptyset$ we obtain $I \wedge D \wedge \mathsf{Elim}(G) \models G$, as desired. In the inductive proof, the cases when $T = \emptyset$ are straightforward. The case $f(t) \in T$ uses the fact that if $M \models G[f(t) := y]$ and $M \models f(t) = y$, then $M \models G$. ∎

Completeness. We now analyze the classes of formulas G for which Elim is *complete.*

Definition 18 *We say that* Elim *is complete for* (D, G) *iff*
$I \wedge D \models G$ *implies* $I \wedge D \models \mathsf{Elim}(G)$.

Note that we cannot hope to achieve completeness for arbitrary constraints D. Indeed, if we let $D \equiv \mathsf{true}$, then D imposes no constraint whatsoever on the derived fields, and reasoning about the derived fields becomes reasoning about uninterpreted function symbols, that is, reasoning in unconstrained predicate logic. Such reasoning is undecidable not only for monadic second-order logic, but also for much weaker fragments of first-order logic [113]. Despite these general observations, we have identified two cases important in practice for which Elim is complete (Theorem 19 and Theorem 25).

Theorem 19 expresses the fact that, in the case where all field constraints are deterministic, Elim is complete (and then it reduces to previous approaches [129, 190] that are restricted to the deterministic case). The proof of Theorem 19 uses the assumption that F is total and functional to conclude $\forall x\ y.\ F(x, y) \ \rightarrow\ f(x) = y$, and then uses an inductive argument similar to the proof of Theorem 17.

Theorem 19 (Completeness for deterministic fields) *Algorithm* Elim *is complete for* (D, G) *when each field constraint in* D *is deterministic.*
What is more, $I \wedge D \wedge G \models \mathsf{Elim}(G)$.

Proof. Consider a field constraint $F \equiv \mathsf{FC}_f$ and let $\bar{x}$ and $\bar{y}$ be such that $F(\bar{x}, \bar{y})$. Because $F(\bar{x}, f(\bar{x}))$ and F is deterministic by assumption, we have $\bar{y} = f(\bar{x})$. It follows that $I \wedge D \wedge F(x, y) \models f(x) = y$. We then prove by induction on the argument G of elim that, for all finite K,

$$I \wedge D \wedge G \wedge \bigwedge_{(x_i, f_i, y_i) \in K} f_i(x_i) = y_i \models \mathsf{elim}(G, K)$$

For $K = \emptyset$ we obtain $I \wedge D \wedge G \models \mathsf{Elim}(G)$, as desired. The inductive proof is similar to the proof of Theorem 17. In the case $f(t) \in T$, we consider a model M such that $M \models I \wedge D \wedge G \wedge \bigwedge_{(x_i, f_i, y_i) \in K} f_i(x_i) = y_i$. Consider any $\bar{x}, \bar{y}$ such that: 1) $M \models x = t$, 2) $M \models F(x, y)$ and 3) $M \models x = x_i \ \rightarrow\ y = y_i$ for all $(x_i, f, y_i) \in K$. To show $M \models \mathsf{elim}(G_1, K \cup \{(x, f, y)\})$, we consider a modified model $M_1 = M[f(\bar{x}) := \bar{y}]$ which is like M except that the interpretation of f at $\bar{x}$ is $\bar{y}$. By $M \models F(x, y)$ we conclude $M_1 \models I \wedge D$. By $M \models x = x_i \ \rightarrow\ y = y_i$, we conclude $M_1 \models \bigwedge_{(x_i, f_i, y_i) \in K} f_i(x_i) = y_i$ as well. Because $I \wedge D \wedge F(x, y) \models f(x) = y$, we conclude $M_1 \models f(x) = y$. Because $M \models x = t$ and $\mathsf{Derived}(t) = \emptyset$, we have $M_1 \models x = t$ so from $M \models G$ we conclude $M_1 \models G_1$ where $G_1 = G[f(t) := y]$. By induction hypothesis we then conclude $M_1 \models \mathsf{elim}(G_1, K \cup \{(x, f, y)\}$. Then also $M \models \mathsf{elim}(G_1, K \cup \{(x, f, y)\}$ because the result of elim does not contain f. Because $\bar{x}, \bar{y}$ were arbitrary, we conclude $M \models \mathsf{elim}(G, K)$. ∎

We next turn to completeness in the cases that admit non-determinism of derived fields. Theorem 25 states that our algorithm is complete for derived fields introduced by the weakest precondition operator to a class of postconditions that includes field constraints. This result is very important in practice. For example, when we used a previous version of an elimination algorithm that was incomplete, we were not able to verify the skip list example in Section 6.1.2. To formalize our completeness result, we introduce two classes of well-behaved formulas: *nice formulas* and *pretty nice formulas*.

Definition 20 (Nice formulas) *A formula G is a* nice formula *if each occurrence of each derived field f in G is of the form $f(t)$, where $t \in \mathsf{Ground}(G)$.*

Nice formulas generalize the notion of quantifier-free formulas by disallowing quantifiers only for variables that are used as arguments to derived fields. Lemma 21 shows that the elimination of derived fields from nice formulas is complete. The intuition behind Lemma 21 is that if $I \wedge D \models G$, then for the choice of y_i such that $F(x_i, y_i)$ we can find an interpretation of the function symbol f such that $f(x_i) = y_i$, and $I \wedge D$ holds, so G holds as well, and $\mathsf{Elim}(G)$ evaluates to the same truth value as G.

Lemma 21 Elim *is complete for (D, G) if G is a nice formula.*

Proof. Let G be a nice formula. To show that $I \wedge D \models G$ implies $I \wedge D \models \mathsf{Elim}(G)$, let $I \wedge D \models G$ and let $f_1(t_1), \ldots, f_n(t_n)$ be the occurrences of derived fields in G. By assumption, $t_1, \ldots, t_n \in \mathsf{Ground}(G)$ and $\mathsf{Elim}(G)$ is of the form

$$\begin{array}{l}\forall x_1\, y_1.\ x_1 = t_1 \ \rightarrow\ (F_1^1 \wedge \\ \quad \forall x_2\, y_2.\ x_2 = t_2' \ \rightarrow\ (F_1^2 \wedge \\ \quad\quad \ldots \\ \quad\quad \forall x_n, y_n.\ x_n = t_n' \ \rightarrow\ (F_1^n \wedge G_0) \ldots))\end{array}$$

where t_i' differs from t_i in that some of its subterms may be replaced by variables y_j for $j < i$. Here $F^i = \mathsf{FC}_{f_i}$ and

$$F_1^i = F^i(x_i, y_i) \wedge \bigwedge_{j<i, f_j = f_i} (x_i = x_j \ \rightarrow\ y_i = y_j).$$

Consider a model M of $I \wedge D$, we show M is a model for $\mathsf{Elim}(G)$. Consider any assignment $\bar{x}_i, \bar{y}_i$ to variables x_i, y_i for $1 \leq i \leq n$. If any of the conditions $x_i = t_i$ or F_1^i are false for this assignment, then $\mathsf{Elim}(G)$ is true because these conditions are on the left-hand side of an implication. Otherwise, conditions $F_1^i(x_i, y_i)$ hold, so by definition of F_1^i, if $\bar{x}_i = \bar{x}_j$, then $\bar{y}_i = \bar{y}_j$. Therefore, for each distinct function symbol f_j there exist a function $\bar{f}_j$ such that $\bar{f}(x_i) = \bar{y}_i$ for $f_j = f_i$. Because $F^i(x_i, y_i)$ holds and each FC_f is total, we can define such $\bar{f}_j$ so that D holds. Let $M' = M[f_j \mapsto \bar{f}_j]_j$ be a model that differs from M only in that f_j are interpreted as $\bar{f}_j$. Then $M' \models I$ because I does not mention derived fields and $M' \models D$ by construction. We therefore conclude $M' \models G$. If $\bar{t}_i$ is the value of t_i in M', then $\bar{x}_i = \bar{t}_i$ because $M \models x_i = t_i$ and $\mathsf{Derived}(t_i) = \emptyset$. Using this fact, as well as $\bar{f}_j(\bar{x}_i) = \bar{y}_i$, by induction on subformulas of G_0 we conclude that G_0 has the same truth value as G in M', so $M' \models G_0$. Because G_0 does not contain derived function symbols, $M \models G_0$ as well. Because $\bar{x}_i$ and $\bar{y}_i$ were arbitrary, we conclude $M \models \mathsf{Elim}(G)$. This completes the proof.

Remark. Note that it is not the case that a stronger statement $I \wedge D \wedge G \models \mathsf{Elim}(G)$ holds. For example, take $D \equiv \mathsf{true}$, and $G \equiv f(a) = b$. Then $\mathsf{Elim}(G)$ is equivalent to $\forall y.y = b$ and it is not the case that $I \wedge f(a) = b \models \forall y.y = b$. ■

Definition 22 (Pretty nice formulas) *The set of* pretty nice formulas *is defined inductively by 1) a nice formula is pretty nice; 2) if G_1 and G_2 are pretty nice, then $G_1 \wedge G_2$ is pretty nice; 3) if G is pretty nice and x is a first-order variable, then $\forall x.G$ is pretty nice.*

Pretty nice formulas therefore additionally admit universal quantification over arguments of derived fields.

Lemma 23 *The following observations hold:*

1. *each field constraint* D_f *is a pretty nice formula;*
2. *if* G *is a pretty nice formula, then* $\mathsf{skolem}(G)$ *is a nice formula and* $H \models G$ *iff* $H \models \mathsf{skolem}(G)$ *for any set of sentences* H*.*

The next Lemma 24 shows that pretty nice formulas are closed under wlp; the lemma follows from the conjunctivity of the weakest precondition operator.

Lemma 24 *Let* c *be a guarded command of the language in Figure 3-3. If* G *is a nice formula, then* $\mathsf{wlp}(c, G)$ *is a nice formula. If* G *is a pretty nice formula, then* $\mathsf{wlp}(c, G)$ *is equivalent to a pretty nice formula.*

Proof. Using the conjunctivity properties of wlp:

$$\mathsf{wlp}(c, \forall x.G) \;\leftrightarrow\; \forall x.\mathsf{wlp}(c, G)$$

and

$$\mathsf{wlp}(c, G_1 \wedge G_2) \;\leftrightarrow\; \mathsf{wlp}(c, G_1) \wedge \mathsf{wlp}(c, G_2)$$

the problem reduces to proving the lemma for the case of nice formulas.

Since we defined wlp recursively on the structure of commands, we prove the statement by structural induction on command c. For $c = (e_1 := e_2)$ and $c = \mathsf{havoc}(x)$ we have that wlp replaces ground terms by ground terms, i.e. in particular all introduced occurrences of derived fields are ground. For $c = \mathsf{assume}(F)$ and $c = \mathsf{assert}(F)$ every occurrence of a derived field introduced by wlp comes from F. Since F is quantifier free, all such occurrences are ground. The remaining cases follow from the induction hypothesis for component commands. ∎

Lemmas 24, 23, 21, and 15 imply our main theorem, Theorem 25. Theorem 25 implies that Elim is a complete technique for checking preservation (over straight-line code) of field constraints, even if they are conjoined with additional pretty nice formulas. Elimination is also complete for data structure operations with loops as long as the necessary loop invariants are pretty nice.

Theorem 25 (Completeness for preservation of field constraints) *Let* G *be a pretty nice formula,* D *a conjunction of field constraints, and* c *a guarded command (Figure 3-3). Then*

$$I \wedge D \models \mathsf{wlp}(c, G \wedge D) \quad \textit{iff} \quad I \models \mathsf{Elim}(\mathsf{wlp}(c, \mathsf{skolem}(G \wedge D))).$$

Proof. Let G be a quite nice formula, D a conjunction of field constraints, and c a guarded command. Since $\mathsf{skolem}(G \wedge D)$ is a nice formula, Lemma 24 implies that $\mathsf{wlp}(c, \mathsf{skolem}(G \wedge D))$ is a nice formula, so we have

$$\begin{array}{ll} I \wedge D \models \mathsf{wlp}(c, G \wedge D) & \\ I \wedge D \models \mathsf{wlp}(c, \mathsf{skolem}(G \wedge D)) & \text{(by Lemma 23)} \\ I \wedge D \models \mathsf{Elim}(\mathsf{wlp}(c, \mathsf{skolem}(G \wedge D))) & \text{(by Lemma 21)} \\ I \models \mathsf{Elim}(\mathsf{wlp}(c, \mathsf{skolem}(G \wedge D))) & \text{(by Lemma 15)} \;\blacksquare \end{array}$$

Limits of completeness. In our implementation, we have successfully used Elim in the context of MSOL, where we encode transitive closure using second-order quantification. Unfortunately, formulas that contain transitive closure of derived fields are often not pretty nice, leading to false alarms after the application of Elim. This behavior is to be expected due to the undecidability of transitive closure logics over general graphs [128]. On the other hand, unlike approaches based on axiomatizations of transitive closure in first-order logic, our use of MSOL enables complete reasoning about reachability over the backbone fields. It is therefore useful to be able to consider a field as part of a backbone whenever possible. For this purpose, it can be helpful to verify conjunctions of constraints using different backbone for different conjuncts.

Verifying conjunctions of constraints. In our skip list example, the field `nextSub` forms an acyclic (sub-)list. It is therefore possible to verify the conjunction of constraints independently, with `nextSub` a derived field in the first conjunct (as in Section 6.1.2) but a backbone field in the second conjunct. Therefore, although the reasoning about transitive closure is incomplete in the first conjunct, it is complete in the second conjunct.

6.3 Using Field Constraint Analysis to Approximate HOL Formulas

We next describe the approximation of HOL formulas using field constraint analysis and monadic second-order logic over trees. The input to this approximation is a HOL formula representing a sequent that results from splitting the verification condition into conjuncts. Jahob replaces the sequent with a stronger formula in MSOL over trees and then attempts to prove the resulting formula using the MONA tool.

Identifying tree backbone. Jahob examines the assumptions of a sequent and pattern-matches the assumptions of the form `tree`$[f_1, \ldots, f_n]$, identifying $f_1, \ldots, f_n$ as the edges in the tree backbone. If there are multiple `tree` declarations, Jahob currently uses only the `tree` constraint occurring in the last assumption of the sequent. The assumption order in a sequent corresponds to the assumption order in the loop-free code fragment from which the sequent is generated, so this policy corresponds to reasoning with respect to the tree backbone in the most recent program state.

Identifying derived fields. Jahob pattern-matches an assumption of the form $\forall x\, y.\; f\, x = y \rightarrow F$ as a field constraint on the field f and extracts F as the formula for approximating the occurrences of the field f. If a field f occurs neither in such a derived field assumption nor in a `tree` invariant, Jahob assumes that f is a derived field with a dummy field constraint `True`.

Eliminating derived fields. Having identified tree backbone fields and derived fields of a data structure, Jahob eliminates the derived fields as described in Section 6.2. The result is a formula where all fields are backbone fields.

Approximating remaining constructs. Jahob conservatively approximates any remaining constructs not directly supported by MSOL over trees by replacing subformulas containing them with `True` or `False`. The unsupported constructs include constraints on integer variables.

Using MSOL over trees. Jahob emits the approximated formula in MONA input format. If the formula contains only one backbone field name, Jahob uses the string mode of MONA, otherwise it uses the tree mode of MONA. To support an unbounded number of disconnected tree data structures, Jahob uses a form of structure simulation [129] with an implicit list that stores all tree data structures in the program. Note that Jahob's semantic model represents fields as total functions and treats `null` as a special object, resulting in many objects pointing to the `null` object. Jahob uses structure simulation mapping that avoids such sharing of `null` by using distinct MSOL structure objects to represent `null` values of different objects of the source structure. To make this modelling faithful to Jahob semantics, Jahob replaces object equality in the input sequent with an equivalence relation that conflates all such null values and acts as the identity relation on the remaining objects.

6.4 Experience with Field Constraint Analysis

We have initially implemented field constraint analysis and deployed it as the "Bohne" analysis plugin of our Hob framework [152, 165]. We have successfully verified singly-linked lists, doubly-linked lists with and without iterators and header nodes (Section 6.1.1), two-level skip lists (Section 6.1.2), and our students example from Section 6.1. Because we have integrated Bohne into the Hob framework, we were able to verify just the parts of programs which require the power of field constraint analysis with the Bohne plugin, while using less detailed analyses for the remainder of the program. We have used the list data structures verified with Bohne as modules of larger examples, such as the 900-line Minesweeper benchmark and the 1200-line web server benchmark. Hob's pluggable analysis approach allowed us to use the typestate plugin [164] and loop invariant inference techniques to efficiently verify client code, while reserving shape analysis for the container data structures.

We have subsequently integrated Bohne and field constraint analysis into the Jahob system as well. This allowed us to largely decouple Bohne's algorithm for inferring universally quantified loop invariants from the decision procedures used to answer queries, both during loop invariant inference and during proving programs with explicitly supplied loop invariants. The combination of additional reasoning procedures (for example, first-order reasoning and Nelson-Oppen style decision procedures) with field constraint analysis and MSOL allowed us to verify programs that require more precise reasoning about uninterpreted fields than supported by field constraint analysis. Such additional precision was needed in particular when verifying the specifications where the definition of a content of a data structure depends on a derived field that has a weak field constraint.

6.5 Further Related Work

We are not aware of any previous work that provides completeness guarantees for analyzing tree-like data structures with non-deterministic cross-cutting fields for expressive constraints such as MSOL. TVLA [226, 174] was initially designed as an analysis framework with user-supplied transfer functions; subsequent work addresses synthesis of transfer functions using finite differencing [219], which is not guaranteed to be complete. Decision procedures [185, 160] are effective at reasoning about local properties, but are not complete for reasoning about reachability. Promising, although still incomplete, approaches include [172] as well as [196, 161]. Some reachability properties can be reduced to first-order properties using hints in the form of ghost fields, as in Chapter 5, and as suggested in [151, 185]. Completeness of analysis can be achieved by representing loop invariants or candidate loop invariants by formulas in a logic that supports transitive closure [190, 262, 157, 259, 261, 251, 211]. These approaches treat decision procedure as a black box and, when applied to MSOL, inherit the limitations of structure simulation [129]. Our work can be viewed as a technique for lifting existing decision procedures into decision procedures that are applicable to a larger class of structures. Therefore, it can be incorporated into all of these previous approaches.

6.6 Conclusion

Shape analysis is one of the most challenging problems in the field of program analysis; its central relevance stems from the fact that it addresses key data structure consistency properties that are 1) important in and of themselves 2) critical for the further verification of other program properties.

Historically, the primary challenge in shape analysis was seen to be dealing effectively with the extremely precise and detailed consistency properties that characterize many (but by no means all) data structures. Perhaps for this reason, many formalisms were built on logics that supported *only* data structures with very precisely defined referencing relationships. This chapter presents an analysis that supports both the extreme precision of previous approaches and the controlled reduction in the precision required to support a more general class of data structures whose referencing relationships may be random, depend on the history of the data structure, or vary for some other reason that places the referencing relationships inherently beyond the ability of previous logics and analyses to characterize. We have deployed this analysis in the context of the Hob program analysis and verification system; our results show that it is effective at 1) analyzing individual data structures to 2) verify interfaces that allow other, more scalable analyses to verify larger-grain data structure consistency properties whose scope spans larger regions of the program.

In a broader context, we view our result as taking an important step towards the practical application of shape analysis. By supporting data structures whose backbone functionally determines the referencing relationships as well as data structures with inherently less structured referencing relationships, it promises to be able to successfully analyze the broad range of data structures that arise in practice.

Chapter 7

Boolean Algebra with Presburger Arithmetic for Data Structure Sizes

In this chapter I present several results about the first-order theory of Boolean Algebra with Presburger arithmetic, interpreted over subsets of a finite set. This chapter is based on [154], whose earlier version appeared in [153] and [158]. In addition, I present a new result that I established in November 2006: an algorithm proving that quantifier-free Boolean Algebra with Presburger Arithmetic is in NP. I also include additional observations about real measures of sets (the end of Section 7.9) and the translations between HOL and BAPA (Figure 7-1 and Figure 7-15).

I have noted already that sets are a useful abstraction of data structure content. The motivation for this chapter is the fact that we often need to reason not only about the content of a data structure, but also about the size of a data structure. For example, we may want to express the fact that the number of elements stored in a data structure is equal to the value of an integer variable that is used to cache the data structure size, or we may want to introduce a decreasing integer measure on the data structure to show program termination. These considerations lead to a natural generalization of the first-order theory of BA of sets, a generalization that allows integer variables in addition to set variables, and allows stating relations of the form $|A| = k$ meaning that the cardinality of the set A is equal to the value of the integer variable k. Once we have integer variables, a natural question arises: which relations and operations on integers should we allow? It turns out that, using only the BA operations and the cardinality operator, we can already define all operations of PA. This leads to the structure BAPA, which properly generalizes both BA and PA.

As I explain in Section 7.1, a version of BAPA was shown decidable already in [90]. Recently, a decision procedure for a fragment of BAPA without quantification over sets was presented in [265], cast as a multi-sorted theory. Starting from [154, 164] as the motivation, I used quantifier elimination in [158] to show the decidability of the full BAPA, which was initially stated as an open question in [265]. A quantifier-elimination algorithm for a single-sorted version of BAPA was presented independently in [221] as a way of evaluating queries in constraint databases; [221] leaves open the complexity of the decision problem.

I give the first formal description of a decision procedure for the full first-order theory of BAPA. Furthermore, I analyze this decision procedure and show that it yields optimal computational complexity for BAPA, identical to the complexity of PA [32]. This solves the question of the computational complexity of BAPA.[1] We have also implemented our decision procedure; I report on our initial experience in using the decision procedure in Jahob.

Contributions. I summarize the contributions of this chapter as follows.

1. As a **motivation** for BAPA, I show in Section 7.2 that BAPA constraints can be used for program analysis and verification by expressing 1) data structure invariants and the correctness of procedures with respect to their specifications,[2] 3) simulation relations between program fragments, 4) termination conditions for programs that manipulate data structures, and 5) projection of formulas onto a desired set of variables, with applications in static analysis, model checking, automated abstraction, and relational query evaluation.

2. I present an **algorithm** α (Section 7.3) that translates BAPA sentences into PA sentences by translating set quantifiers into integer quantifiers and show how it can be used to decide the truth value of PA formulas and to eliminate individual quantifiers (Section 7.5).

3. I analyze the algorithm α and show that its **complexity** matches the lower bound for PA and is therefore **optimal** (Section 7.4). This result solves the question of the complexity of the decision problem for BAPA and is the main technical contribution of this chapter. Our analysis includes showing an alternating time upper bound for PA, parameterized by the number of quantifier alternations.

4. I show that **the quantifier-free fragment of** BAPA can be solved in nondeterministic polynomial time, using a technique that, for the first time, avoids considering an exponential number of Venn regions.

5. I discuss our initial experience in using our **implementation** of BAPA to discharge verification conditions generated in our verification system.

6. I observe the following additional results:

   (a) PA sentences generated by translating BA sentences without cardinalities can be decided in optimal alternating time for BA (Section 7.4.4), which gives an alternative proof of upper **bound for** BA **of sets**;

   (b) Our algorithm extends to **countable sets** with a predicate distinguishing finite and infinite sets (Section 7.8);

   (c) In contrast to the undecidability of monadic second-order logic over strings (MSOL) when extended with equicardinality operator, I identify a decidable combination of **MSOL with** BA (Section 7.8).

[1]In [153] I state only the corresponding space upper bound on BAPA; I thank Dexter Kozen for suggesting to use an alternating time complexity class of [32] to establish the optimal bound.

[2]This motivation was presented first in [158] and was subsequently used in [266].

$$
\begin{array}{rcl}
[\![\_]\!] & : & \textsf{BAPA}\text{ formulas} \rightarrow \textsf{HOL}\text{ formulas} \\
[\![f_1 \land f_2]\!] & = & [\![f_1]\!] \land [\![f_2]\!] \\
[\![f_1 \lor f_2]\!] & = & [\![f_1]\!] \lor [\![f_2]\!] \\
[\![\neg F]\!] & = & \neg [\![F]\!] \\
[\![\exists x. f]\!] & = & \exists x :: \textsf{obj set}. [\![f]\!] \\
[\![\forall x. f]\!] & = & \forall x :: \textsf{obj set}. [\![f]\!] \\
[\![\exists k. f]\!] & = & \exists k :: \textsf{int}. [\![f]\!] \\
[\![\forall k. f]\!] & = & \forall k :: \textsf{int}. [\![f]\!] \\
[\![b_1 = b_2]\!] & = & [\![b_1]\!] = [\![b_2]\!] \\
[\![b_1 \subseteq b_2]\!] & = & [\![b_1]\!] \subseteq [\![b_2]\!] \\
[\![t_1 = t_2]\!] & = & [\![t_1]\!] = [\![t_2]\!] \\
[\![t_1 < t_2]\!] & = & [\![t_1]\!] < [\![t_2]\!] \\
[\![K \,\textsf{dvd}\, t]\!] & = & ([\![K]\!] \,\textsf{mod}\, [\![t]\!] = 0) \\
[\![x]\!] & = & x \\
[\![\mathbf{0}]\!] & = & \emptyset \\
[\![\mathbf{1}]\!] & = & \textsf{Object} \\
[\![b_1 \cup b_2]\!] & = & [\![b_1]\!] \cup [\![b_2]\!] \\
[\![b_1 \cap b_2]\!] & = & [\![b_1]\!] \cap [\![b_2]\!] \\
[\![b^c]\!] & = & \textsf{Object} \setminus [\![b]\!] \\
[\![k]\!] & = & k \\
[\![K]\!] & = & K \\
[\![\textsf{MAXC}]\!] & = & \textsf{cardinality Object} \\
[\![t_1 + t_2]\!] & = & [\![t_1]\!] + [\![t_2]\!] \\
[\![K \cdot t]\!] & = & [\![K]\!] * [\![t]\!] \\
[\![\,|b|\,]\!] & = & \textsf{cardinality}\, [\![b]\!]
\end{array}
$$

Figure 7-1: Embedding of BAPA into HOL

$$
\begin{array}{rcl}
F & ::= & A \mid F_1 \land F_2 \mid F_1 \lor F_2 \mid \neg F \mid \exists x.F \mid \forall x.F \\
A & ::= & B_1 = B_2 \mid B_1 \subseteq B_2 \mid |B| = K \mid |B| \geq K \\
B & ::= & x \mid \mathbf{0} \mid \mathbf{1} \mid B_1 \cup B_2 \mid B_1 \cap B_2 \mid B^c \\
K & ::= & 0 \mid 1 \mid 2 \mid \ldots
\end{array}
$$

Figure 7-2: Formulas of Boolean Algebra (BA)

$$
\begin{array}{rcl}
F & ::= & A \mid F_1 \land F_2 \mid F_1 \lor F_2 \mid \neg F \mid \exists k.F \mid \forall k.F \\
A & ::= & T_1 = T_2 \mid T_1 < T_2 \mid K \,\mathsf{dvd}\, T \\
T & ::= & K \mid T_1 + T_2 \mid K \cdot T \\
K & ::= & \ldots -2 \mid -1 \mid 0 \mid 1 \mid 2 \ldots
\end{array}
$$

Figure 7-3: Formulas of Presburger Arithmetic (PA)

7.1 The First-Order Theory BAPA

Figure 7-4 presents the syntax of Boolean Algebra with Presburger Arithmetic (BAPA), which is the focus of this chapter. We next present some justification for the operations in Figure 7-4. Our initial motivation for BAPA was the use of BA to reason about data structures in terms of sets [152]. Our language for BA (Figure 7-2) allows cardinality constraints of the form $|A| = K$ where K is a *constant* integer. Such constant cardinality constraints are useful and enable quantifier elimination for the resulting language [232, 176]. However, they do not allow stating constraints such as $|A| = |B|$ for two sets A and B, and cannot represent constraints on relationships between sizes of sets and integer variables. Consider therefore the equicardinality relation $A \sim B$ that holds iff $|A| = |B|$, and consider BA extended with relation $A \sim B$. Define the ternary relation $\mathsf{plus}(A, B, C) \iff (|A| + |B| = |C|)$ by the formula $\exists x_1.\ \exists x_2.\ x_1 \cap x_2 = \emptyset\ \land\ A \sim x_1\ \land\ B \sim x_2\ \land\ x_1 \cup x_2 = C$. The relation $\mathsf{plus}(A, B, C)$ allows us to express addition using arbitrary sets as representatives for natural numbers; $\emptyset$ can represent the natural number zero, and any singleton set can represent the natural number one. (The property of A being a singleton is definable using e.g. the first-order formula $A \neq \emptyset \land \forall B. A \cap B = B \rightarrow (B = \emptyset \lor B = A)$.) Moreover, we can represent integers as equivalence classes of pairs of natural numbers under the equivalence relation $(x, y) \approx (u, v) \iff x + v = u + y$; this construction also allows us to express the unary predicate of being non-negative. The quantification over pairs of sets represents quantification over integers, and quantification over integers with the addition operation and the predicate "being non-negative" can express all PA operations, presented in Figure 7-3. Therefore, a natural closure under definable operations leads to our formulation of the language BAPA in Figure 7-4, which contains both sets and integers.

The argument above also explains why we attribute the decidability of BAPA to [90, Section 8], which showed the decidability of BA over sets extended with the equicardinality relation $\sim$, using the decidability of the first-order theory of the addition of cardinal numbers.

The language BAPA has two kinds of quantifiers: quantifiers over integers and quantifiers over sets; we distinguish between these two kinds by denoting integer variables with symbols such as k, l and set variables with symbols such as x, y. We use the shorthand $\exists^+ k.F(k)$ to

$$\begin{array}{rcl} F & ::= & A \mid F_1 \wedge F_2 \mid F_1 \vee F_2 \mid \neg F \mid \exists x.F \mid \forall x.F \mid \exists k.F \mid \forall k.F \\ A & ::= & B_1 = B_2 \mid B_1 \subseteq B_2 \mid T_1 = T_2 \mid T_1 < T_2 \mid K \,\mathsf{dvd}\, T \\ B & ::= & x \mid \mathbf{0} \mid \mathbf{1} \mid B_1 \cup B_2 \mid B_1 \cap B_2 \mid B^c \\ T & ::= & k \mid K \mid \mathsf{MAXC} \mid T_1 + T_2 \mid K \cdot T \mid \ \mathsf{|} B \mathsf{|} \\ K & ::= & \ldots -2 \mid -1 \mid 0 \mid 1 \mid 2 \ldots \end{array}$$

Figure 7-4: Formulas of Boolean Algebra with Presburger Arithmetic (BAPA)

denote $\exists k.k \geq 0 \wedge F(k)$ and, similarly $\forall^+ k.F(k)$ to denote $\forall k.k \geq 0 \rightarrow F(k)$. In summary, the language of BAPA in Figure 7-4 subsumes the language of PA in Figure 7-3, subsumes the language of BA in Figure 7-2, and contains non-trivial combination of these two languages in the form of using the cardinality of a set expression as an integer value.

The semantics of operations in Figure 7-4 is the expected one. We interpret integer terms as integers, and interpret set terms as elements of the powerset of a finite set. The MAXC constant denotes the size of the finite universe $\mathcal{U}$, so we require $\mathsf{MAXC} = |\mathcal{U}|$ in all models. Our results generalize to the Boolean algebra of powersets of a countable set, see Section 7.8.

Figure 7-1 formalizes the semantics of BAPA by giving a translation into our higher-order language presented in Section 4.1, showing that BAPA is a fragment of higher-order logic.

7.2 Applications of BAPA

This section illustrates the importance of BAPA constraints. Section 7.2.1 shows the uses of BAPA constraints to express and verify data structure invariants as well as procedure preconditions and postconditions. Section 7.2.2 shows how a class of simulation relation conditions can be proved automatically using a decision procedure for BAPA. Section 7.2.3 shows how BAPA can be used to express and prove termination conditions for a class of programs. Section 7.2.4 discusses the applications of quantifier elimination, which is relevant to BAPA because our decision procedure is based on quantifier elimination.

7.2.1 Verifying Data Structure Consistency

Figure 7-5 presents a procedure insert in a language that directly manipulates sets. Such languages can either be directly executed [84] or can arise as abstractions of programs in standard languages [152]. The program in Figure 7-5 manipulates a global set of objects content and an integer field size. The program maintains an invariant I that the size of the set content is equal to the value of the variable size. The insert procedure inserts an element e into the set and correspondingly updates the integer variable. The requires clause (precondition) of the insert procedure is that the parameter e is a non-null reference to an object that is not stored in the set content. The ensures clause (postcondition) of the procedure is that the size variable after the insertion is positive. Note that we represent references to objects (such as the procedure parameter e) as sets with at most one element. An empty set represents a null reference; a singleton set $\{o\}$ represents a reference to object o. The value of a variable after procedure execution is indicated by marking the variable name with a prime.

```
**var** content : **set**;
**var** size : **integer**;
**invariant** $I \iff (\mathsf{size} = |\mathsf{content}|)$;

**procedure** insert($e$ : element)
**maintains** $I$
**requires** $|e| = 1 \wedge |e \cap \mathsf{content}| = 0$
**ensures** $\mathsf{size}' > 0$
{
    content := content $\cup$ $e$;
    size := size + 1;
}
```

Figure 7-5: insert Procedure

$$\{|e| = 1 \wedge |e \cap \mathsf{content}| = 0 \wedge \mathsf{size} = |\mathsf{content}|\}$$
$$\mathsf{content} := \mathsf{content} \cup e; \quad \mathsf{size} := \mathsf{size} + 1;$$
$$\{\mathsf{size}' > 0 \wedge \mathsf{size}' = |\mathsf{content}'|\}$$

Figure 7-6: Hoare Triple for insert

$$\begin{array}{l}\forall e.\ \forall \mathsf{content}.\ \forall \mathsf{content}'.\ \forall \mathsf{size}.\ \forall \mathsf{size}'.\\ \quad (|e| = 1 \wedge |e \cap \mathsf{content}| = 0 \wedge \mathsf{size} = |\mathsf{content}| \wedge\\ \quad\ \mathsf{content}' = \mathsf{content} \cup e \wedge \mathsf{size}' = \mathsf{size} + 1) \rightarrow\\ \qquad \mathsf{size}' > 0 \wedge \mathsf{size}' = |\mathsf{content}'|\end{array}$$

Figure 7-7: Verification Condition for Figure 7-6

The insert procedure maintains an invariant, I, which captures the relationship between the size of the set content and the integer variable size. The invariant I is implicitly conjoined with the requires and the ensures clauses of the procedure. The Hoare triple in Figure 7-6 summarizes the resulting correctness condition for the insert procedure. Figure 7-7 presents a verification condition corresponding to the Hoare triple in Figure 7-6. Note that the verification condition contains both set and integer variables, contains quantification over these variables, and relates the sizes of sets to the values of integer variables. Our small example leads to a formula without quantifier alternations; in general, formulas that arise in verification may contain alternations of existential and universal variables over both integers and sets. This chapter shows the decidability of such formulas and presents the complexity of the decision procedure.

7.2.2 Proving Simulation Relation Conditions

BAPA constraints are also useful when proving that a given binary relation on states is a simulation relation between two program fragments. Figure 7-8 shows one such example. The concrete procedure start1 manipulates two sets: a set of running processes and a set of suspended processes in a process scheduler. The procedure start1 inserts a new process x into the set of running processes R, unless there are already too many running processes. The procedure start2 is a version of the procedure that operates in a more abstract state space: it maintains only the union P of all processes and the number k of running processes. Figure 7-8 shows a forward simulation relation r between the transition relations for start1 and start2. The standard simulation relation diagram condition is $\forall s_1. \forall s_1'. \forall s_2. (t_1(s_1, s_1') \wedge r(s_1, s_2)) \rightarrow \exists s_2'. \ (t_2(s_2, s_2') \wedge r(s_1', s_2'))$. In the presence of preconditions, $t_1(s_1, s_1') = (\mathsf{pre}_1(s_1) \rightarrow \mathsf{post}_1(s_1, s_1'))$ and $t_2(s_2, s_2') = (\mathsf{pre}_2(s_2) \rightarrow \mathsf{post}_2(s_2, s_2'))$, and

```
var R : set;
var S : set;

procedure start1($x$)
requires $x \not\subseteq \mathsf{R} \wedge |x| = 1 \wedge |\mathsf{R}| < \mathsf{MAXR}$
ensures $\mathsf{R}' = \mathsf{R} \cup x \wedge \mathsf{S}' = \mathsf{S}$
{
  R := R ∪ $x$;
}
```

```
var P : set;
var k : integer;

procedure start2($x$)
requires $x \not\subseteq \mathsf{P} \wedge |x| = 1 \wedge \mathsf{k} < \mathsf{MAXR}$
ensures $\mathsf{P}' = \mathsf{P} \cup x \wedge \mathsf{k}' = \mathsf{k} + 1$
{
  P := P ∪ $x$;
  k := k + 1;
}
```

Simulation relation r:
$r((\mathsf{R},\mathsf{S}),(\mathsf{P},\mathsf{k})) = (\mathsf{P} = \mathsf{R} \cup \mathsf{S} \wedge \mathsf{k} = |\mathsf{R}|)$

Simulation relation conditions in BAPA:

1. $\forall x, \mathsf{R}, \mathsf{S}, \mathsf{P}, \mathsf{k}.(\mathsf{P} = \mathsf{R} \cup \mathsf{S} \wedge \mathsf{k} = |\mathsf{R}|) \wedge (x \not\subseteq \mathsf{P} \wedge |x| = 1 \wedge \mathsf{k} < \mathsf{MAXR}) \rightarrow (x \not\subseteq \mathsf{R} \wedge |x| = 1 \wedge |\mathsf{R}| < \mathsf{MAXR})$
2. $\forall x, \mathsf{R}, \mathsf{S}, \mathsf{R}', \mathsf{S}', \mathsf{P}, \mathsf{k}.\exists \mathsf{P}', \mathsf{k}'.((\mathsf{P} = \mathsf{R} \cup \mathsf{S} \wedge \mathsf{k} = |\mathsf{R}|) \wedge (\mathsf{R}' = \mathsf{R} \cup x \wedge \mathsf{S}' = \mathsf{S}) \wedge (x \not\subseteq \mathsf{P} \wedge |x| = 1 \wedge \mathsf{k} < \mathsf{MAXR})) \rightarrow (\mathsf{P}' = \mathsf{P} \cup x \wedge \mathsf{k}' = \mathsf{k} + 1) \wedge (\mathsf{P}' = \mathsf{R}' \cup \mathsf{S}' \wedge \mathsf{k}' = |\mathsf{R}'|)$

Figure 7-8: Proving simulation relation in BAPA

sufficient conditions for simulation relation are:

1. $\forall s_1.\forall s_2.r(s_1, s_2) \wedge \mathsf{pre}_2(s_2) \rightarrow \mathsf{pre}_1(s_1)$
2. $\forall s_1.\forall s_1'.\forall s_2.\exists s_2'.\ r(s_1, s_2) \wedge \mathsf{post}_1(s_1, s_1') \wedge \mathsf{pre}_2(s_2) \ \rightarrow\ \mathsf{post}_2(s_2, s_2') \wedge r(s_1', s_2')$

Figure 7-8 shows BAPA formulas that correspond to the simulation relation conditions in this example. Note that the second BAPA formula has a quantifier alternation, which illustrates the relevance of quantifiers in BAPA.

7.2.3 Proving Program Termination

We next show that BAPA is useful for proving program termination. A standard technique for proving termination of a loop is to introduce a ranking function f that maps program states to non-negative integers, then prove that the value of the function decreases at each loop iteration. In other words, if $t(s, s')$ denotes the relationship between the state at the beginning and the state at the end of each loop iteration, then the condition $\forall s.\forall s'.t(s, s') \rightarrow f(s) > f(s')$ holds. Figure 7-9 shows an example program that processes each element of the initial value of set iter; this program can be viewed as manipulating an iterator over a data structure that implements a set. Using the the ability to take cardinality of a set allows us to define a natural ranking function for this program. Figure 7-10 shows the termination proof based on such ranking function. The resulting termination condition can be expressed as a formula that belongs to BAPA, and can be discharged using our decision procedure. In general, we can reduce the termination problem of programs that manipulate both sets and integers to showing a simulation relation with a fragment of a terminating program that manipulates only integers, which can be proved terminating using techniques [210]. The simulation relation condition can be proved correct using our BAPA decision procedure whenever the simulation relation is expressible with a BAPA formula. While one could, in principle, use finite sets directly to describe certain ranking functions, the ability to abstract sets into integers allows us to use existing tools and techniques developed for

```
var iter : set;

procedure iterate()
{
  while iter ≠ ∅ do
    var e : set;
    e := choose iter;
    iter := iter \ e;
    process(e);
  done
}
```

Figure 7-9: Terminating procedure

Ranking function:
$f(s) = |s|$

Transition relation:
$t(\mathsf{iter}, \mathsf{iter}') = (\exists e.\ |e| = 1 \wedge e \subseteq \mathsf{iter} \wedge \mathsf{iter}' = \mathsf{iter} \setminus e)$

Termination condition in BAPA:
$$\forall \mathsf{iter}.\forall \mathsf{iter}'.\ (\exists e.|e| = 1 \wedge e \subseteq \mathsf{iter} \wedge \mathsf{iter}' = \mathsf{iter} \setminus e) \\ \rightarrow |\mathsf{iter}'| < |\mathsf{iter}|$$

Figure 7-10: Termination proof for Figure 7-9

integer ranking functions.

7.2.4 Quantifier Elimination

The fact that BAPA admits quantifier elimination enables applications that involve hiding certain variables from a BAPA formula. Hiding a variable x in a formula means existentially quantifying over x and then eliminating the quantifier $\exists x$. This process can also be viewed as a projection of a formula onto variables other than x. As an example, Figure 7-11 shows the transition relation inspired by the procedure insert in Figure 7-6. The transition relation mentions a variable e that is local to the procedure and not meaningful outside it. In the public description of the transition relation the variable e is existentially quantified. Our quantifier elimination algorithm (Section 7.3, Section 7.5) removes the quantifier from the formula and obtains an equivalent formula without quantifiers, such as the one shown in the lower part of Figure 7-11.

In general, variable hiding is useful in projecting state transitions and invariants onto a desired set of variables, computing relational composition of transition relations, and computing the image of a set under a transition relation. Such symbolic computation of transition relations, with appropriate widenings, can be used to generalize static analyses such as [164] and model checking approaches such as [44]. Quantifier elimination here ensures that the transition relation remains represented by a quantifier-free formula throughout the analysis.

Quantifier elimination is also useful for query evaluation in constraint databases [221], and loop invariant inference [137].

7.3 Decision Procedure for BAPA

This section presents our algorithm, denoted α, which decides the validity of BAPA sentences. The algorithm reduces a BAPA sentence to an equivalent PA sentence with the

$$\exists e.\ |e| = 1 \wedge \mathsf{content}' = \mathsf{content} \cup e$$
$$\Downarrow$$
$$\mathsf{content} \subseteq \mathsf{content}' \ \wedge\ |\mathsf{content}' \setminus \mathsf{content}| \leq 1 \ \wedge\ |\mathsf{content}'| \geq 1$$

Figure 7-11: Eliminating a local variable from a transition relation

same number of quantifier alternations and an exponential increase in the total size of the formula. This algorithm has several desirable properties:

1. Given an optimal algorithm for deciding PA sentences, the algorithm α is optimal for deciding BAPA sentences and shows that the complexity of BAPA is the same as the complexity of PA (Section 7.4).

2. The algorithm α does not eliminate integer variables, but instead produces an equivalent quantified PA sentence. The resulting PA sentence can therefore be decided using *any* decision procedure for PA, including the decision procedures based on automata [143, 36].

3. The algorithm α can eliminate set quantifiers from any extension of PA. We thus obtain a technique for adding a particular form of set reasoning to every extension of PA, and the technique preserves the decidability of the extension. One example of decidable theory that extends PA is MSOL over strings, see Section 7.8.

4. For simplicity we present the algorithm α as a decision procedure for formulas with no free variables, but the algorithm can be used to transform and simplify formulas with free variables as well, because it transforms one quantifier at a time starting from the innermost one. We explore this version of our algorithm in Section 7.5.

We next describe the algorithm α for transforming a BAPA sentence F_0 into a PA sentence. As the first step of the algorithm, transform F_0 into prenex form

$$Q_p v_p. \dots Q_1 v_1.\ F(v_1, \dots, v_p)$$

where F is quantifier-free, and each quantifier $Q_i v_i$ is of one the forms $\exists k$, $\forall k$, $\exists y$, $\forall y$ where k denotes an integer variable and y denotes a set variable.

The next step of the algorithm is to separate F into a BA part and a PA part. To achieve this, replace each formula $b_1 = b_2$ where b_1 and b_2 are set expressions, with the conjunction $b_1 \subseteq b_2 \wedge b_2 \subseteq b_1$, and replace each formula $b_1 \subseteq b_2$ with the equivalent formula $|b_1 \cap b_2^c| = 0$. In the resulting formula, each set variable x occurs in some term $|t(x)|$. Next, use the same reasoning as when generating disjunctive normal form for propositional logic to write each set expression $t(x)$ as a union of cubes (regions in the Venn diagram [243]). The cubes have the form $\bigcap_{i=1}^{n} x_i^{\alpha_i}$ where $x_i^{\alpha_i}$ is either x_i or x_i^c; there are $m = 2^n$ cubes $s_1, \dots, s_m$. Suppose that $t(x) = s_{j_1} \cup \dots \cup s_{j_a}$; then replace the term $|t(x)|$ with the term $\sum_{i=1}^{a} |s_{j_i}|$. In the resulting formula, each set x appears in an expression of the form $|s_i|$ where s_i is a cube. For each s_i introduce a new variable l_i. The resulting formula is then equivalent to

$$\begin{aligned} & Q_p v_p. \dots Q_1 v_1. \\ & \quad \exists^+ l_1, \dots, l_m.\ \bigwedge_{i=1}^{m} |s_i| = l_i \ \wedge\ G_1 \end{aligned} \tag{7.1}$$

where G_1 is a PA formula. Formula (7.1) is the starting point of the main phase of algorithm α. The main phase of the algorithm successively eliminates quantifiers $Q_1 v_1, \dots, Q_p v_p$ while maintaining a formula of the form

$$\begin{aligned} & Q_p v_p \dots Q_r v_r. \\ & \quad \exists^+ l_1 \dots l_q.\ \bigwedge_{i=1}^{q} |s_i| = l_i \ \wedge\ G_r \end{aligned} \tag{7.2}$$

where G_r is a PA formula, r grows from 1 to $p+1$, and $q = 2^e$ where e for $0 \leq e \leq n$ is the number of set variables among $v_p, \dots, v_r$. The list $s_1, \dots, s_q$ is the list of all 2^e partitions formed from the set variables among $v_p, \dots, v_r$.

We next show how to eliminate the innermost quantifier $Q_r v_r$ from the formula (7.2). During this process, the algorithm replaces the formula G_r with a formula G_{r+1} which has more integer quantifiers. If v_r is an integer variable then the number of sets q remains the same, and if v_r is a set variable, then q reduces from 2^e to 2^{e-1}. We next consider each of the four possibilities $\exists k$, $\forall k$, $\exists y$, $\forall y$ for the quantifier $Q_r v_r$.

Case $\exists$k: Consider first the case $\exists k$. Because k does not occur in $\bigwedge_{i=1}^{q} |s_i| = l_i$, simply move the existential quantifier to G_r and let $G_{r+1} = \exists k.G_r$, which completes the step.

Case $\forall$k: For universal quantifiers, it suffices to let $G_{r+1} = \forall k.G_r$, again because k does not occur in $\bigwedge_{i=1}^{q} |s_i| = l_i$.

Case $\exists$y: We next show how to eliminate an existential set quantifier $\exists y$ from

$$\exists y.\ \exists^+ l_1 \dots l_q.\ \bigwedge_{i=1}^{q} |s_i| = l_i \ \wedge \ G_r \tag{7.3}$$

which is equivalent to $\exists^+ l_1 \dots l_q.\ (\exists y. \bigwedge_{i=1}^{q} |s_i| = l_i) \ \wedge \ G_r$. This is the key step of the algorithm and relies on the following lemma.

Lemma 26 *Let $b_1, \dots, b_n$ be finite disjoint sets, and $l_1, \dots, l_n, k_1, \dots, k_n$ be natural numbers. Then the following two statements are equivalent:*

1. *There exists a finite set y such that*
$$\bigwedge_{i=1}^{n} |b_i \cap y| = k_i \ \wedge \ |b_i \cap y^c| = l_i \tag{1}$$

2. $$\bigwedge_{i=1}^{n} |b_i| = k_i + l_i \tag{2}$$

Proof. ($\rightarrow$) Suppose that there exists a set y satisfying (1). Because $b_i \cap y$ and $b_i \cap y^c$ are disjoint, $|b_i| = |b_i \cap y| + |b_i \cap y^c|$, so $|b_i| = k_i + l_i$.

($\leftarrow$) Suppose that (2) holds, so $|b_i| = k_i + l_i$ for each of the pairwise disjoint sets $b_1, \dots, b_n$. For each b_i choose a subset $y_i \subseteq b_i$ such that $|y_i| = k_i$. Because $|b_i| = k_i + l_i$, we have $|b_i \cap y_i^c| = l_i$. Having chosen $y_1, \dots, y_n$, let $y = \bigcup_{i=1}^{n} y_i$. For $i \neq j$ we have $b_i \cap y_j = \emptyset$ and $b_i \cap y_j^c = b_i$, so $b_i \cap y = y_i$ and $b_i \cap y^c = b_i \cap y_i^c$. By the choice of y_i, we conclude that y is the desired set for which (1) holds. ∎

In the quantifier elimination step, assume without loss of generality that the set variables $s_1, \dots, s_q$ are numbered such that $s_{2i-1} \equiv s_i' \cap y^c$ and $s_{2i} \equiv s_i' \cap y$ for some cube s_i'. Then apply Lemma 26 and replace each pair of conjuncts

$$|s_i' \cap y^c| = l_{2i-1} \ \wedge \ |s_i' \cap y| = l_{2i}$$

with the conjunct $|s_i'| = l_{2i-1} + l_{2i}$, yielding the formula

$$\exists^+ l_1 \dots l_q.\ \bigwedge_{i=1}^{q'} |s_i'| = l_{2i-1} + l_{2i} \ \wedge \ G_r \tag{7.4}$$

$$
\begin{aligned}
&\alpha_1\,(\exists y.\ \exists^+ l_1 \ldots l_{2q'}.\ \textstyle\bigwedge_{i=1}^{q'} |s_i \cap y^c| = l_{2i-1} \wedge |s_i \cap y| = l_{2i} \wedge G_r) = \\
&\qquad \exists^+ l'_1 \ldots l'_{q'}.\ \textstyle\bigwedge_{i=1}^{q'} |s_i| = l'_i \wedge \exists^+ l_1 \ldots l_{2q'}.\textstyle\bigwedge_{i=1}^{q'} .l'_i = l_{2i-1} + l_{2i} \wedge G_r \\
&\alpha_1\,(\forall y.\ \exists^+ l_1 \ldots l_{2q'}.\ \textstyle\bigwedge_{i=1}^{q'} |s_i \cap y^c| = l_{2i-1} \wedge |s_i \cap y| = l_{2i} \wedge G_r) = \\
&\qquad \exists^+ l'_1 \ldots l'_{q'}.\ \textstyle\bigwedge_{i=1}^{q'} |s_i| = l'_i \wedge \forall^+ l_1 \ldots l_{2q'}.\textstyle\bigwedge_{i=1}^{q'} .l'_i = l_{2i-1} + l_{2i} \rightarrow G_r \\
&\alpha_1\,(\exists k.\ \exists^+ l_1 \ldots l_{2q'}.\ \textstyle\bigwedge_{i=1}^{q'} |s_i \cap y^c| = l_{2i-1} \wedge |s_i \cap y| = l_{2i} \wedge G_r) = \\
&\qquad \exists^+ l_1 \ldots l_{2q'}.\ \textstyle\bigwedge_{i=1}^{q'} |s_i \cap y^c| = l_{2i-1} \wedge |s_i \cap y| = l_{2i} \wedge \exists k.G_r \\
&\alpha_1\,(\forall k.\ \exists^+ l_1 \ldots l_{2q'}.\ \textstyle\bigwedge_{i=1}^{q'} |s_i \cap y^c| = l_{2i-1} \wedge |s_i \cap y| = l_{2i} \wedge G_r) = \\
&\qquad \exists^+ l_1 \ldots l_{2q'}.\ \textstyle\bigwedge_{i=1}^{q'} |s_i \cap y^c| = l_{2i-1} \wedge |s_i \cap y| = l_{2i} \wedge \forall k.G_r
\end{aligned}
$$

$$\alpha_F(G(|\mathbf{1}|)) = G(\mathsf{MAXC})$$

$$\alpha = \mathsf{prenex}\ ;\ \mathsf{separate}\ ;\ \alpha_1^*\ ;\ \alpha_F$$

Figure 7-12: Algorithm α for translating BAPA sentences to PA sentences

for $q' = 2^{e-1}$. Finally, to obtain a formula of the form (7.2) for $r+1$, introduce fresh variables l'_i constrained by $l'_i = l_{2i-1} + l_{2i}$, rewrite (7.4) as

$$\exists^+ l'_1 \ldots l'_{q'}.\ \bigwedge_{i=1}^{q'} |s'_i| = l'_i \ \wedge\ (\exists^+ l_1 \ldots l_q.\ \bigwedge_{i=1}^{q'} l'_i = l_{2i-1} + l_{2i} \ \wedge\ G_r)$$

and let

$$G_{r+1} \equiv \exists^+ l_1 \ldots l_q.\ \bigwedge_{i=1}^{q'} l'_i = l_{2i-1} + l_{2i} \ \wedge\ G_r \tag{$\exists$-step}$$

This completes the description of the elimination of an existential set quantifier $\exists y$.

Case $\forall$y: To eliminate a set quantifier $\forall y$, observe that

$$\neg(\exists^+ l_1 \ldots l_q.\ \bigwedge_{i=1}^{q} |s_i| = l_i \ \wedge\ G_r)$$

is equivalent to $\exists^+ l_1 \ldots l_q.\ \bigwedge_{i=1}^{q} |s_i| = l_i \ \wedge\ \neg G_r$, because existential quantifiers over l_i together with the conjuncts $|s_i| = l_i$ act as definitions for l_i, so we may first substitute all values l_i into G_r, then perform the negation, and then extract back the definitions of all values l_i. By expressing $\forall y$ as $\neg\exists y \neg$, we can show that the elimination of $\forall y$ is analogous to elimination of $\exists y$: introduce fresh variables $l'_i = l_{2i-1} + l_{2i}$ and let

$$G_{r+1} \equiv \forall^+ l_1 \ldots l_q.\ (\bigwedge_{i=1}^{q'} l'_i = l_{2i-1} + l_{2i}) \ \rightarrow\ G_r \tag{$\forall$-step}$$

Final step: After eliminating all quantifiers by repeatedly applying the step of the algorithm, we obtain a formula of the form $\exists^+ l.\ |\mathbf{1}| = l \wedge G_{p+1}(l)$. Namely, in the step when we have only one set variable y and its negation y^c, we can write $|y|$ and $|y^c|$ as $|\mathbf{1} \cap y$ and $|\mathbf{1} \cap y^c|$ and apply the algorithm one more time. We then define the result of the algorithm, denoted $\alpha(F_0)$, to be the PA sentence $G_{p+1}(\mathsf{MAXC})$.

We summarize the algorithm in Figure 7-12. We use $f ; g$ to denote the function composition $g \circ f$, and we use f^* to denote iterative application of function f. The prenex function transforms a formula into the prenex form, whereas the separate function transforms it into

$$
\begin{array}{l}
\forall^+ l_1.\forall^+ l_0.\ \mathsf{MAXC} = l_1 + l_0 \rightarrow \\
\forall^+ l_{11}.\forall^+ l_{01}.\forall^+ l_{10}.\forall^+ l_{00}. \\
l_1 = l_{11} + l_{01} \wedge l_0 = l_{10} + l_{00} \rightarrow \\
\quad \forall^+ l_{111}.\ \forall^+ l_{011}.\ \forall^+ l_{101}.\ \forall^+ l_{001}. \\
\quad \forall^+ l_{110}.\ \forall^+ l_{010}.\ \forall^+ l_{100}.\ \forall^+ l_{000}. \\
\quad\quad l_{11} = l_{111} + l_{011}\ \wedge l_{01} = l_{101} + l_{001}\ \wedge \\
\quad\quad l_{10} = l_{110} + l_{010}\ \wedge l_{00} = l_{100} + l_{000} \rightarrow \\
\quad\quad\quad \forall size.\forall size'. \\
\quad\quad\quad\quad (l_{111} + l_{011} + l_{101} + l_{001} = 1 \wedge \\
\quad\quad\quad\quad l_{111} + l_{011} = 0 \wedge \\
\quad\quad\quad\quad l_{111} + l_{011} + l_{110} + l_{010} = size \wedge \\
\quad\quad\quad\quad l_{100} = 0 \wedge \\
\quad\quad\quad\quad l_{011} + l_{001} + l_{010} = 0 \wedge \\
\quad\quad\quad\quad size' = size + 1) \rightarrow \\
\quad\quad\quad\quad\quad (0 < size' \wedge \\
\quad\quad\quad\quad\quad l_{111} + l_{101} + l_{110} + l_{100} = size')
\end{array}
$$

Figure 7-13: The translation of the BAPA sentence from Figure 7-7 into a PA sentence

general relationship:

$$l_{i_1,\ldots,i_k} = |\mathsf{set}_q^{i_1} \cap \mathsf{set}_{q+1}^{i_2} \cap \ldots \cap \mathsf{set}_S^{i_k}|$$
$$q = S - (k-1)$$
$$(S \text{ is number of set variables})$$

in this example:

$$
\begin{array}{l}
\mathsf{set}_1 = \mathsf{content}' \\
\mathsf{set}_2 = \mathsf{content} \\
\mathsf{set}_3 = e
\end{array}
$$

$$
\left.
\begin{array}{l}
l_{000} = |\mathsf{content}'^c \cap \mathsf{content}^c \cap e^c| \\
l_{001} = |\mathsf{content}'^c \cap \mathsf{content}^c \cap e| \\
l_{010} = |\mathsf{content}'^c \cap \mathsf{content} \cap e^c| \\
l_{011} = |\mathsf{content}'^c \cap \mathsf{content} \cap e| \\
l_{100} = |\mathsf{content}' \cap \mathsf{content}^c \cap e^c| \\
l_{101} = |\mathsf{content}' \cap \mathsf{content}^c \cap e| \\
l_{110} = |\mathsf{content}' \cap \mathsf{content} \cap e^c| \\
l_{111} = |\mathsf{content}' \cap \mathsf{content} \cap e|
\end{array}
\right\} (H)
$$

Figure 7-14: The correspondence, denoted H, between integer variables in Figure 7-13 and set variables in Figure 7-7

form (7.1). We have argued above that each of the individual steps of the algorithm is equivalence preserving, so we have the following lemma.

Lemma 27 *The transformations* prenex*,* separate*, α_1, α_F are all equivalence preserving (with respect to the* BAPA *interepretation).*

By induction we obtain the correctness of our algorithm.

Theorem 28 *The algorithm α in Figure 7-12 maps each* BAPA*-sentence F_0 into an equivalent* PA*-sentence $\alpha(F_0)$.*

The validity of PA sentences is decidable [212]. In combination with a decision procedure for PA such as [215, 36, 119], the algorithm α is a decision procedure for BAPA sentences.

7.3.1 Example Run of Algorithm α

As an illustration, we show the result of runing the algorithm α on the BAPA formula in Figure 7-7. The result is the PA formula in Figure 7-13. Note that the structure of the resulting formula mimics the structure of the original formula: every set quantifier is replaced by the corresponding block of quantifiers over non-negative integers constrained to partition the previously introduced integer variables. Figure 7-14 presents the correspondence between the set variables of the BAPA formula and the integer variables of the translated PA formula. Note that the relationship $\mathsf{content}' = \mathsf{content} \cup e$ translates into the conjunction of the constraints $|\mathsf{content}' \cap (\mathsf{content} \cup e)^c| = 0 \wedge |(\mathsf{content} \cup e) \cap \mathsf{content}'^c| = 0$, which reduces to the conjunction $l_{100} = 0 \wedge l_{011} + l_{001} + l_{010} = 0$ using the translation of set expressions into the disjoint union of partitions, and the correspondence in Figure 7-14.

7.4 Complexity of BAPA

In this section we analyze the algorithm α from Section 7.3 and use it to show that the computational complexity of BAPA is identical to the complexity of PA, which is

$\mathsf{STA}(*, 2^{2^{n^{O(1)}}}, n)$ [32], that is, alternating doubly exponential time with a linear number of alternations.

An alternating Turing machine [52] is a generalization of a non-deterministic Turing machine that, in addition to making non-deterministic (existential) choices, can make universal choices. A universal choice step succeeds iff the computation succeeds for all the chosen values. We then have the following definition.

Definition 29 [32]. *$\mathsf{STA}(s(n), t(n), a(n))$ denotes the class of languages computable using an alternating Turing machine that uses $s(n)$ cells on its tape, runs in $t(n)$ steps, and performs $a(n)$ alternations.*

In the remainder of this section we first show the lower bound on the complexity of BAPA. We then use the algorithm α in the previous section and a parameterized upper bound on PA to establish the matching upper bound on BAPA. We finally show that the algorithm α can also decide BA formulas using optimal resources $\mathsf{STA}(*, 2^n, n)$, which is the complexity established in [145]. Moreover, by construction, our procedure reduces to the procedure for PA formulas if there are no set quantifiers. Therefore, the algorithm α can be used as a component for an optimal algorithm for BAPA but also for the special cases of BA and PA.

7.4.1 Lower Bound on the Complexity of Deciding BAPA

The lower bound of PA follows by showing how to encode full integer arithmetic in which quantifiers are bounded by 2^{2^n} using formulas of size n [146, Lecture 23].

Fact 1 [32, Page 75]. *The truth value of PA formulas of size $O(n)$ can encode the acceptance of an alternating Turing machine with n alternations running in 2^{2^n} steps.*

Because BAPA contains PA, the lower bound directly applies to BAPA.

Lemma 30 *The truth value of BAPA formulas of size $O(n)$ can encode the acceptance of an alternating Turing machine with n alternations running in 2^{2^n} steps.*

7.4.2 Parameterized Upper Bound on PA

As a step towards establishing the upper bound for BAPA, we show a parameterized upper bound on PA. We use the following result.

Fact 2 [217, Page 324]. *If F is a closed PA formula $(Q_1 x_1) \ldots (Q_r x_r) G(x_1, \ldots, x_r)$ of size $n > 4$ with m quantifier alternations, where $Q_1, \ldots, Q_r$ are quantifiers and G is quantifier-free, then F is true iff the formula*

$$\left(Q_1 x_1 \le 2^{2^{(c+1)n^{m+3}}}\right) \ldots \left(Q_r x_r \le 2^{2^{(c+r)n^{m+3}}}\right) G(x_1, \ldots, x_r)$$

with bounded quantifiers is true for some $c > 0$.

Fact 2 allows us to establish the following parameterized upper bound on PA.

Theorem 31 *A PA sentence of size n with m quantifier alternations can be decided in $\mathsf{STA}(*, 2^{n^{O(m)}}, m)$.*

Proof. Analogous to the proof of the space upper bound in [217, Page 325]. To decide formulas with m quantifier alternations, use an alternating Turing machine that does m alternations, mimicking the quantifier alternations. The Turing machine guesses the assignments to variables $x_1, \ldots, x_r$ in ranges given by Fact 2. This requires guessing

$$2^{(c+1)n^{m+3}} + \ldots + 2^{(c+r)n^{m+3}} \leq 2^{c_1 n^{m+4}}$$

bits (because $r \leq n$), and then evaluating the formula in time proportional to $n2^{c_1 n^{m+4}} \leq 2^{c_2 n^{m+4}}$, which is in $2^{n^{O(m)}}$. ∎

7.4.3 Upper Bound on the Complexity of Deciding BAPA

We next show that the algorithm in Section 7.3 transforms a BAPA sentence F_0 into a PA sentence whose size is at most exponential and which has the same number of quantifier alternations. Theorem 31 will then give us the upper bound on BAPA that matches the lower bound of Lemma 30.

If F is a formula in prenex form, let $\mathsf{size}(F)$ denote the size of F, and let $\mathsf{alts}(F)$ denote the number of quantifier alternations in F.

Lemma 32 *For the algorithm α from Section 7.3 there is a constant $c > 0$ such that $\mathsf{size}(\alpha(F_0)) \leq 2^{c \cdot \mathsf{size}(F_0)}$ and $\mathsf{alts}(\alpha(F_0)) = \mathsf{alts}(F_0)$. Moreover, the algorithm α runs in $2^{O(\mathsf{size}(F_0))}$ deterministic time.*

Proof. To gain some intuition on the size of $\alpha(F_0)$ compared to the size of F_0, compare first the formula in Figure 7-13 with the original formula in Figure 7-7. Let n denote the size of the initial formula F_0 and let S be the number of set variables. Note that the following operations are polynomially bounded in time and space: 1) transforming a formula into prenex form, 2) transforming relations $b_1 = b_2$ and $b_1 \subseteq b_2$ into the form $|b| = 0$. Introducing set variables for each partition and replacing each $|b|$ with a sum of integer variables yields formula G_1 whose size is bounded by $O(n2^S S)$ (the last S factor is because representing a variable from the set of K variables requires space $\log K$). The subsequent transformations introduce the existing integer quantifiers, whose size is bounded by n, and introduce additionally $2^{S-1} + \ldots + 2 + 1 = 2^S - 1$ new integer variables along with the equations that define them. Note that the defining equations always have the form $l'_i = l_{2i-1} + l_{2i}$ and have size bounded by S. We therefore conclude that the size of $\alpha(F_0)$ is $O(nS(2^S + 2^S))$ and therefore $O(nS2^S)$, which is in $2^{O(n)}$. Note that we have obtained a more precise bound $O(nS2^S)$ indicating that the exponential explosion is caused only by set variables. Finally, the fact that the number of quantifier alternations is the same in F_0 and $\alpha(F_0)$ is immediate because the algorithm replaces one set quantifier with a block of corresponding integer quantifiers. ∎

Combining Lemma 32 with Theorem 31 we obtain the upper bound for BAPA.

Lemma 33 *The validity of a BAPA sentence of size n and the number of quantifier alternations m can be decided in $\mathsf{STA}(*, 2^{2^{O(mn)}}, m)$.*

Proof. The algorithm first applies the algorithm α and then applies the algorithm from Theorem 31. Consider a BAPA formula of size n with m quantifier alternations. By Lemma 32 applying α takes $2^{O(n)}$ steps and produces a formula of size $2^{O(n)}$ with m quantifier alternations. Theorem 31 then allows deciding such formula in $\mathsf{STA}(*, 2^{2^{(2^{O(n)})^{O(m)}}}, m)$, which is $\mathsf{STA}(*, 2^{2^{O(mn)}}, m)$. ∎

Summarizing Lemma 30 and Lemma 33, taking into account that $2^{O(mn)}$ is in $2^{n^{O(1)}}$ for $m \leq n$, we establish the desired theorem.

Theorem 34 *The validity of* BAPA *formulas is* $\mathsf{STA}(*, 2^{2^{n^{O(1)}}}, n)$*-complete.*

7.4.4 Deciding BA as a Special Case of BAPA

We next analyze the result of applying the algorithm α to a pure BA sentence F_0. By a pure BA sentence we mean a BA sentence without cardinality constraints, containing only the standard operations $\cap, \cup, ^c$ and the relations $\subseteq, =$. At first, it might seem that the algorithm α is not a reasonable approach to deciding BA formulas given that the complexity of PA is worse than the corresponding of BA. However, we show that the sentences $\mathsf{PA}_{\mathsf{BA}} = \{\alpha(F_0) \mid F_0 \text{ is in } \mathsf{BA}\}$ generated by applying α to pure BA sentences can be decided using resources optimal for BA [145]. The key observation is that if a PA formula is in $\mathsf{PA}_{\mathsf{BA}}$, then we can use the bounds for quantified variables that are smaller than the bounds in Fact 2.

Let F_0 be a pure BA formula and let S be the number of set variables in F_0 (the set variables are the only variables in F_0). Let $l_1, \ldots, l_q$ be the free variables of the formula $G_r(l_1, \ldots, l_q)$ in the algorithm α. Then $q = 2^e$ for $e = S + 1 - r$. Let $w_1, \ldots, w_q$ be integers specifying the values of $l_1, \ldots, l_q$. We then have the following lemma.

Lemma 35 *For each r where $1 \leq r \leq S$, formula $G_r(w_1, \ldots, w_q)$ is equivalent to formula $G_r(\bar{w}_1, \ldots, \bar{w}_q)$ where $\bar{w}_i = \min(w_i, 2^{r-1})$.*

Proof. We prove the claim by induction. For $r = 1$, observe that the translation of a quantifier-free part of the pure BA formula yields a PA formula F_1 whose all atomic formulas are of the form $l_{i_1} + \ldots + l_{i_k} = 0$, which are equivalent to $\bigvee_{j=1}^{k} l_{i_j} = 0$. Therefore, the truth-value of F_1 depends only on whether the integer variables are zero or non-zero, which means that we may restrict the variables to interval $[0, 1]$.

For the inductive step, consider the elimination of a set variable, and assume that the property holds for G_r and for all q-tuples of non-negative integers $w_1, \ldots, w_q$. Let $q' = q/2$ and $w'_1, \ldots, w'_{q'}$ be a tuple of non-negative integers. We show that $G_{r+1}(w'_1, \ldots, w'_{q'})$ is equivalent to $G_{r+1}(\bar{w}'_1, \ldots, \bar{w}'_{q'})$. We consider the case when G_{r+1} is obtained by eliminating an existential set quantifier, so

$$G_{r+1} \equiv \exists^+ l_1 \ldots l_q. \bigwedge_{i=1}^{q'} l'_i = l_{2i-1} + l_{2i} \;\wedge\; G_r$$

The case for the universal quantifier can be obtained from the proof for the existential one by expressing $\forall$ as $\neg\exists\neg$. We show both directions of the equivalence.

Suppose first that $G_{r+1}(\bar{w}'_1, \ldots, \bar{w}'_{q'})$ holds. Then for each $\bar{w}'_i$ there are u_{2i-1} and u_{2i} such that $\bar{w}'_i = u_{2i-1} + u_{2i}$ and $G_r(u_1, \ldots, u_q)$. We define the witnesses $w_1, \ldots, w_q$ for existential quantifiers as follows. If $w'_i \leq 2^r$, then $\bar{w}'_i = w'_i$, so we use the same witnesses, letting $w_{2i-1} = u_{2i-1}$ and $w_{2i} = u_{2i}$. Let $w'_i > 2^r$, then $\bar{w}'_i = 2^r$. Because $u_{2i-1} + u_{2i} = 2^r$, we have $u_{2i-1} \geq 2^{r-1}$ or $u_{2i} \geq 2^{r-1}$. Suppose, without loss of generality, $u_{2i} \geq 2^{r-1}$. Then let $w_{2i-1} = u_{2i-1}$ and let $w_{2i} = w'_i - u_{2i-1}$. Because $w_{2i} \geq 2^{r-1}$ and $u_{2i} \geq 2^{r-1}$, by induction hypothesis we have

$$G_r(\ldots, w_{2i}, \ldots) \iff G_r(\ldots, u_{2i}, \ldots) \iff G_r(\ldots, 2^{r-1}, \ldots)$$

For $w_1, \ldots, w_q$ chosen as above we therefore have $w'_i = w_{2i-1} + w_{2i}$ and $G_r(w_1, \ldots, w_q)$, which by definition of G_{r+1} means that $G_{r+1}(w'_1, \ldots, w'_{q'})$ holds.

Conversely, suppose that $G_{r+1}(w'_1, \ldots, w'_{q'})$ holds. Then there are $w_1, \ldots, w_q$ such that $G_r(w_1, \ldots, w_q)$ and $w'_i = w_{2i-1} + w_{2i}$. If $w_{2i-1} \leq 2^{r-1}$ and $w_{2i} \leq 2^{r-1}$ then $w'_i \leq 2^r$ so let $u_{2i-1} = w_{2i-1}$ and $u_{2i} = w_{2i}$. If $w_{2i-1} > 2^{r-1}$ and $w_{2i} > 2^{r-1}$ then let $u_{2i-1} = 2^{r-1}$ and $u_{2i} = 2^{r-1}$; we have $u_{2i-1} + u_{2i} = 2^r = \bar{w}'_i$ because $w'_i > 2^r$. If $w_{2i-1} > 2^{r-1}$ and $w_{2i} \leq 2^{r-1}$ then let $u_{2i-1} = 2^r - w_{2i}$ and $u_{2i} = w_{2i}$. By induction hypothesis we have $G_r(u_1, \ldots, u_q) \iff G_r(w_1, \ldots, w_q)$. Furthermore, $u_{2i-1} + u_{2i} = \bar{w}'_i$, so $G_{r+1}(\bar{w}'_1, \ldots, \bar{w}'_{q'})$ by definition of G_{r+1}. ∎

Theorem 36 *The decision problem for* $\mathsf{PA}_{\mathsf{BA}}$ *is in* $\mathsf{STA}(*, 2^{O(n)}, n)$.

Proof. Consider a formula F_0 of size n with S variables. Then $\alpha(F_0) = G_{S+1}$. By Lemma 32, $\mathsf{size}(\alpha(F_0))$ is in $2^{O(n)}$ and $\alpha(F_0)$ has at most S quantifier alternations. By Lemma 35, it suffices for the outermost quantified variable of $\alpha(F_0)$ to range over the integer interval $[0, 2^S]$, and the range of subsequent variables is even smaller. Therefore, the value of each of the $2^{O(S)}$ variables is given by $O(S)$ bits. The values of all bound variables in $\alpha(F_0)$ can therefore be guessed in alternating time $2^{O(S)}$ using S alternations. The truth value of a PA formula for given values of variables can be evaluated in time polynomial in the size of the formula, so deciding $\alpha(F_0)$ can be done in $\mathsf{STA}(*, 2^{O(n)}, S)$, which is bounded by $\mathsf{STA}(*, 2^{O(n)}, n)$. ∎

7.5 Eliminating Individual Variables from a Formula

Section 7.3 described an algorithm for BAPA that reduces all set quantifiers in a BAPA sentence to integer quantifiers. The advantage of such an algorithm is that it can be applied to combinations of BA with extensions of PA that need not support quantifier elimination (see Section 7.8.2). Moreover, this version of the algorithm made the complexity analysis in Section 7.4 easier. However, as we discussed in Section 7.2.4, there are uses for an algorithm that eliminates a quantified variable from a formula with free variables, yielding an equivalent quantifier-free formula. In this section we explain that the individual step α_1 of the algorithm α can be used for this purpose.

Quantifier elimination based on our algorithm is possible because PA itself admits quantifier elimination. Therefore, after transforming a set quantifier into an integer quantifier, we can remove the resulting integer quantifier and substitute back the variables constrained by $l_i = |s_i|$. Denote this elimination of integer quantifiers by PAelim. Then the algorithm α', given by

$$\mathsf{separate}\ ;\ \alpha_1\ ;\ \mathsf{PAelim}$$

eliminates one set or integer quantifier from a BAPA formula $Qv.F$, even if F contains free variables (see also [158] for details). Lemma 27 again implies the correctness of this approach.

Example. We illustrate the algorithm α' on the example in Figure 7-11. We use the naming convention given by the formula H for cardinalities of Venn regions from Figure 7-14. After applying separate we obtain the formula

$$\begin{array}{l} \exists e.\ \exists^+ l_{000}, \ldots, l_{111}.\ H \land \\ \quad l_{111} + l_{011} + l_{101} + l_{001} = 1 \land \\ \quad l_{100} = 0 \ \land\ l_{011} + l_{001} + l_{010} = 0 \end{array}$$

After applying α_1 we obtain the formula

$$\begin{aligned}\exists l_{00}, l_{01}, l_{10}, l_{11}.\ & l_{00} = |\mathsf{content}'^c \cap \mathsf{content}^c| \ \wedge\ l_{01} = |\mathsf{content}'^c \cap \mathsf{content}| \ \wedge \\ & l_{10} = |\mathsf{content}' \cap \mathsf{content}^c| \wedge\ l_{11} = |\mathsf{content}' \cap \mathsf{content}| \ \wedge\ G\end{aligned}$$

where G is the PA formula

$$\begin{aligned}\exists^+ l_{000}, \ldots, l_{111}.\ \ & l_{00} = l_{000} + l_{001} \ \wedge l_{01} = l_{010} + l_{011} \ \wedge \\ & l_{10} = l_{100} + l_{101} \ \wedge l_{11} = l_{110} + l_{111} \ \wedge \\ & l_{111} + l_{011} + l_{101} + l_{001} = 1 \wedge \\ & l_{100} = 0 \ \wedge\ l_{011} + l_{001} + l_{010} = 0\end{aligned}$$

Applying quantifier elimination for PA and simplifications of the quantifier-free formula, we reduce G to

$$l_{01} = 0 \ \wedge\ l_{10} \leq 1 \ \wedge\ l_{11} + l_{10} \geq 1$$

After substituting back the definitions of $l_{00}, \ldots, l_{11}$ we obtain

$$\begin{aligned}& |\mathsf{content}'^c \cap \mathsf{content}| = 0 \ \wedge\ |\mathsf{content}' \cap \mathsf{content}^c| \leq 1 \wedge \\ & |\mathsf{content}' \cap \mathsf{content}| + |\mathsf{content}' \cap \mathsf{content}^c| \geq 1\end{aligned}$$

which can indeed be simplified to the result in Figure 7-11.

7.5.1 Reducing the Number of Integer Variables

The approaches for deciding BAPA described so far always introduce 2^S integer variables where S is the number of set variables in the formula. We next describe observations that, although not an improvement in the worst case, may be helpful for certain classes of formulas.

First, as pointed out in [202], if the input formula entails any BA identities (which can be represented as $|b| = 0$), then the number of non-empty Venn regions decreases, which reduces the number of integer variables in the resulting PA formula, and eliminates such atomic formulas b from further considerations.

Second, when eliminating a particular variable y, we can avoid considering Venn regions with respect to all variables of the formula, and only consider those variables x for which there exists an expression $b_i(x, y)$ where both x and y occur. This requires using a version $\mathsf{separate}_0$ of separation that introduces integer variables only for those terms $|b|$ that occur in the BAPA formula that results from reducing any remaining BA atomic formulas to the form $|b| = 0$. Like the form (7.1) obtained by separate in Section 7.3, the result of $\mathsf{separate}_0$ contains a conjunction of formulas $|b_i| = l_i$ with a PA formula, with two important differences 1) in the case of $\mathsf{separate}_0$ the resulting formula is polynomial in the original size and 2) b_i are arbitrary BA expressions as opposed to Venn regions. Let $a_1(y), \ldots, a_q(y)$ be those terms b_i that contain y, and let $x_1, \ldots, x_{S_1}$ be the free variables in $a_1(y), \ldots, a_q(y)$. When eliminating quantifier Qy, it suffices to introduce 2^{S_1} integer variables corresponding to the the partitions with respect to $x_1, \ldots, x_{S_1}$, which may be an improvement because $S_1 \leq S$.

The final observation is useful if the number q of terms $a_1(y), \ldots, a_q(y)$ satisfies the property $2q < S_1$, i.e. there is a large number of variables, but a small number of BA terms containing them. In this case, consider all Boolean combinations $t_1, \ldots, t_u$ of the $2q$ expressions $a_1(\mathbf{0}), a_1(\mathbf{1}), a_2(\mathbf{0}), a_2(\mathbf{1}), \ldots, a_q(\mathbf{0}), a_q(\mathbf{1})$. For each a_i, we have

$$a_i(y) = (y \cap a_i(\mathbf{0})) \cup (y^c \cap a_i(\mathbf{1}))$$

Each $a_i(\mathbf{0})$ and each $a_i(\mathbf{1})$ is a disjoint union of cubes over the BA expressions $t_1, \ldots, t_u$, so each $a_i(y)$ is a disjoint union of cubes over the BA expressions $y, t_1, \ldots, t_u$. It therefore suffices to introduce 2^{2q} integer variables denoting all terms of the form $y \cap t_i$ and $y^c \cap t_i$, as opposed to 2^{S_1} integer variables.

7.6 Approximating HOL formulas by BAPA formulas

To make BAPA is applicable to a broader range of verification tasks, show how to approximate higher-order logic formulas by BAPA formulas. As suggested in Section 4.2.3, our system splits formulas into independent conjuncts, then attempts to approximates each conjuct with a formula in a known fragment of logic.

Figure 7-15 presents one such approximation that maps HOL formulas to BAPA formulas. This approximation is sound for validity: when the resulting BAPA formula is valid, so is the original BAPA formula. (The converse need not be the case because HOL formulas are more expressive, allowing, in particular reasoning about relations.) The translation function $[\![f]\!]^b$ in Figure 7-15 translates an HOL formula or an expression f into a BAPA formula, a BAPA set term or a BAPA integer term $[\![f]\!]^b$. The boolean parameter p keeps track of the polarity of formulas to ensure sound approximation. We define $\overline{0} = 1$ and $\overline{1} = 0$. The translation conservatively approximates formulas outside the scope of BAPA with the truth value p. The correctness of the transformations is given by the relatioships

$$[\![f]\!]^0 \ \models \ f \ \models \ [\![f]\!]^1$$

which follows by induction, using the semantics of BAPA in Figure 7-15 and the monotonicity of logical operations. The translation uses type information to disambiguate the equality operator, which is polymorphic in our HOL notation and monomorphic in the BAPA notation. The translation assumes that the formula has been type checked, and computes the type $\mathsf{typeof}(f)$ of a subformula f using the types reconstructed by Hindley-Milner type inference. The translation assumes that the input formula is flat, that is, that a previous formula transformation pass flattened complex set and integer expressions by introducing fresh variables. The translation changes the role of object variables, replacing them with identically named variables that denote sets of cardinality one. This explains, for example, the translation of atomic formulas $x = \{y\}$ as $x = y$ and $x \in y$ as $x \subseteq y$. More robust translations from the one in Figure 7-15 are possible. Nevertheless, the current translation does show that BAPA can be easily integrated as a component of a reasoning procedure for higher-order logic formulas arising in verification.

7.7 Experience Using Our Decision Procedure for BAPA

Using Jahob, we generated verification conditions for several Java program fragments that require reasoning about sets and their cardinalities, for example, to prove the equality between the set representing the number of elements in a list and the integer field size after they have been updated, as well as several other examples from Section 7.2. We found that the existing automated techniques were able to deal with some of the formulas involving only sets or only integers, but not with the formulas that relate cardinalities of operations on sets to the cardinalities of the individual sets. These formulas can be proved in Isabelle, but require user interaction in terms of auxiliary lemmas. On the other hand,

$$
\begin{array}{rcl}
[\![\_]\!]^{\_} & : & \mathsf{HOL}\ \text{formulas} \to \mathsf{BAPA}\ \text{formulas} \\
[\![f_1 \wedge f_2]\!]^p & \equiv & [\![f_1]\!]^p \wedge [\![f_2]\!]^p \\
[\![f_1 \vee f_2]\!]^p & \equiv & [\![f_1]\!]^p \vee [\![f_2]\!]^p \\
[\![\neg f]\!]^p & \equiv & \neg [\![f]\!]^{\overline{p}} \\
[\![f_1 \rightarrow f_2]\!]^p & \equiv & [\![f_1]\!]^{\overline{p}} \rightarrow [\![f_2]\!]^p \\
[\![\forall k :: \mathsf{int}.\, f]\!]^p & \equiv & \forall k. [\![f]\!]^p \\
[\![\exists k :: \mathsf{int}.\, f]\!]^p & \equiv & \exists k. [\![f]\!]^p \\
[\![\forall x :: \mathsf{obj\ set}.\, f]\!]^p & \equiv & \forall x. [\![f]\!]^p \\
[\![\exists x :: \mathsf{obj\ set}.\, f]\!]^p & \equiv & \exists x. [\![f]\!]^p \\
[\![\forall x :: \mathsf{obj}.\, f]\!]^p & \equiv & \forall x. |x| = 1 \rightarrow [\![f]\!]^p \\
[\![\exists x :: \mathsf{obj}.\, f]\!]^p & \equiv & \exists x. |x| = 1 \wedge [\![f]\!]^p \\
[\![x = \mathsf{cardinality}\, y]\!]^p & \equiv & x = |y| \\
[\![x < y]\!]^p & \equiv & x < y \\
[\![x = y]\!]^p & \equiv & x = y,\ \mathsf{typeof}(x) = \mathsf{int} \\
[\![x = c \cdot t]\!]^p & \equiv & x = c \cdot t \\
[\![x = y + z]\!]^p & \equiv & x = y + z \\
[\![x = y - z]\!]^p & \equiv & x = y + (-1) \cdot z \\
[\![c\, \mathsf{dvd}\, x]\!]^p & \equiv & \exists k.\ x = c \cdot k \\
[\![x = y]\!]^p & \equiv & x = y,\ \mathsf{typeof}(x) = \mathsf{obj\ set} \\
[\![x = y]\!]^p & \equiv & x = y,\ \mathsf{typeof}(x) = \mathsf{obj} \\
[\![x = \{y\}]\!]^p & \equiv & x = y,\ \mathsf{typeof}(y) = \mathsf{obj} \\
[\![x = y \cup z]\!]^p & \equiv & x = y \cup z,\ \mathsf{typeof}(x) = \mathsf{obj\ set} \\
[\![x = y \cap z]\!]^p & \equiv & x = y \cap z,\ \mathsf{typeof}(x) = \mathsf{obj\ set} \\
[\![x = y \setminus z]\!]^p & \equiv & x = y \cap z^c,\ \mathsf{typeof}(x) = \mathsf{obj\ set} \\
[\![x \subseteq y]\!]^p & \equiv & x \subseteq y,\ \mathsf{typeof}(x) = \mathsf{obj\ set} \\
[\![x \in y]\!]^p & \equiv & x \subseteq y,\ \mathsf{typeof}(x) = \mathsf{obj} \\
[\![x = \emptyset]\!]^p & \equiv & x = \mathbf{0},\ \mathsf{typeof}(x) = \mathsf{obj\ set} \\
[\![x = \mathcal{U}]\!]^p & \equiv & x = \mathbf{1},\ \mathsf{typeof}(x) = \mathsf{obj\ set} \\
[\![f]\!]^0 & \equiv & \mathsf{false},\ \text{if none of the previous cases apply} \\
[\![f]\!]^1 & \equiv & \mathsf{true},\ \text{if none of the previous cases apply}
\end{array}
$$

Figure 7-15: Approximation of HOL by BAPA

our implementation of the decision procedure automatically discharges these formulas.

Our initial experience indicates that a straightforward implementation of the algorithm α works fast as long as the number of set variables is small; typical timings are fractions of a second for 4 or less set variables and less than 10 seconds for 5 variables. More than 5 set variables cause the PA decision procedure to run out of memory. On the other hand, the decision procedure is much less sensitive to the number of integer variables in BAPA formulas, because they translate into the same number of integer variables in the generated PA formula. We used the Omega Calculator to decide PA formulas because we found that it outperforms LASH on the formulas generated from our examples. For quantifier-free BAPA formulas we have also successfully used CVC Lite to prove the resulting quantifier-free PA formulas.

Our current implementation makes use of certain formula transformations to reduce the size of the generated PA formula. We found that eliminating set variables by substitution of equals for equals is an effective optimization. We also observed that lifting quantifiers to the top level noticeably improves the performance of the Omega Calculator. These transformations extend the range of formulas that the current implementation can handle. A possible alternative to the current approach is to interleave the elimination of integer variables with the elimination of the set variables, and to perform formula simplifications during this process. Finally, once we obtain a formula with only existential quantifiers, we can decide its satisfiability more efficiently using the recent improvements for quantifier-free formulas (Section 7.9).

7.8 Further Observations

We next sketch some further observations about BAPA. First we show that the restriction of BAPA to finite sets is not necessary. We then clarify key differences and connections between BAPA and monadic second-order logic over trees.

7.8.1 BAPA of Countably Infinite Sets

Note that our results also extend to the generalization of BAPA where set variables range over subsets of an arbitrary (not necessarily finite) set, which follows from the decidability of the first-order theory of the addition of cardinals [90, Page 78]; see [266, Appendix A] for the complexity of the quantifier-free case. For simplicity of formula semantics, we here consider only the case of all subsets of a countable set, and argue that the complexity results we have developed above for the finite sets still apply. We first generalize the language of BAPA and the interpretation of BAPA operations, as follows. Introduce a function $\mathsf{inf}(b)$ which returns 0 if b is a finite set and 1 if b is an infinite set. Define $|b|$ to be some arbitrary integer (for concreteness, zero) if b is infinite, and the cardinality of b if b is finite. A countable or a finite cardinal can therefore be represented in PA using a pair (k, i) of an integer k and an infinity flag i, where we put $k = 0$ when $i = 1$. The relation representing the addition of cardinals $(k_1, i_1) + (k_2, i_2) = (k_3, i_3)$ is then definable by formula

$$(i_1 = 0 \land i_2 = 0 \land i_3 = 0 \land k_1 + k_2 = k_3) \ \lor \ ((i_1 = 1 \lor i_2 = 1) \land i_3 = 1 \land k_3 = 0)$$

Note that $\mathsf{inf}(x) = \max(\mathsf{inf}(x \cap y),\ \mathsf{inf}(x \cap y^c))$, which allows the reduction of all occurrences of $\mathsf{inf}(b)$ expressions to occurrences where b is a Venn region. Moreover, we have the following generalization of Lemma 26.

Lemma 37 *Let $b_1, \ldots, b_n$ be disjoint sets, $l_1, \ldots, l_n, k_1, \ldots, k_n$ be natural numbers, and $p_1, \ldots, p_n, q_1, \ldots, q_n \in \{0, 1\}$ for $1 \leq i \leq n$. Then the following two statements are equivalent:*

1. *There exists a set y such that*

$$\bigwedge_{i=1}^{n} |b_i \cap y| = k_i \wedge \mathsf{inf}(b_i \cap y) = p_i \ \wedge \ |b_i \cap y^c| = l_i \wedge \mathsf{inf}(b_i \cap y^c) = q_i$$

2. $$\bigwedge_{i=1}^{n} \begin{pmatrix} (p_i = 0 \wedge q_i = 0 \rightarrow |b_i| = k_i + l_i) \wedge \\ (\mathsf{inf}(b_i) = 0 \leftrightarrow (p_i = 0 \wedge q_i = 0)) \wedge \\ (p_i = 1 \rightarrow k_i = 0) \ \wedge \ (q_i = 1 \rightarrow l_i = 0) \end{pmatrix}$$

Proof. The case when $p_i = 0$ and $q_i = 0$ follows as in the proof of Lemma 26. When $p_i = 1$ or $q_i = 1$ then b_i contains an infinite set as a subset, so it must be infinite. Conversely, an infinite set can always be split into a set of the desired size and another infinite set, or into two infinite sets. ■

The BAPA decision procedure for the case of countable set then uses Lemma 37 and generalizes the algorithm α in a natural way. The resulting PA formulas are at most polynomially larger than for the case of finite sets, so we obtain a generalization of Theorem 34 to subsets of a countable set.

7.8.2 BAPA and MSOL over Strings

The weak monadic second-order logic (MSOL) over strings is a decidable logic [235, 119] that can encode Presburger arithmetic by encoding addition using one successor symbol and quantification over sets of elements. There are two important differences between MSOL over strings and BAPA: (1) BAPA can express relationships of the form $|A| = k$ where A is a set variable and k is an integer variable; such relation is not definable in MSOL over strings; (2) when MSOL over strings is used to represent PA operations, the sets contain binary integer digits whereas in BAPA the sets contain uninterpreted elements. Note also that MSOL extended with a construct that takes a set of elements and returns an encoding of the size of that set is undecidable, because it could express MSOL with equicardinality, which is undecidable by a reduction from Post correspondence problem. Despite this difference, the algorithm α gives a way to combine MSOL over strings with BA yielding a decidable theory. Namely, α does not impose any upper bound on the complexity of the theory for reasoning about integers. Therefore, α can decide an extension of BAPA where the constraints on cardinalities of sets are expressed using relations on integers definable in MSOL over strings; these relations go beyond PA [236, Page 400], [43].

7.9 Quantifier-Free BAPA is NP-complete

This section shows that the satisfiability problem for quantifier-free fragment of BAPA, denoted QFBAPA, is NP-complete. QFBAPA satisfiability is clearly NP-hard, because QFBAPA has propositional operators on formulas. Moreover, QFBAPA contains Boolean algebra of sets that has its own propositional structure. The challenge is therefore to prove membership in NP, given formulas such as $|A \setminus B \cup C| = 10000$, which can force the sizes of sets to be exponential, leading to a doubly exponential number of interpretations of set variables.

$$
\begin{array}{rcl}
F & ::= & A \mid F_1 \wedge F_2 \mid F_1 \vee F_2 \mid \neg F \\
A & ::= & B_1 = B_2 \mid B_1 \subseteq B_2 \mid T_1 = T_2 \mid T_1 < T_2 \mid K \,\mathsf{dvd}\, T \\
B & ::= & x \mid \mathbf{0} \mid \mathbf{1} \mid B_1 \cup B_2 \mid B_1 \cap B_2 \mid B^c \\
T & ::= & k \mid K \mid \mathsf{MAXC} \mid T_1 + T_2 \mid K \cdot T \mid \ \mathsf{|}B\mathsf{|} \\
K & ::= & \ldots -2 \mid -1 \mid 0 \mid 1 \mid 2 \ldots
\end{array}
$$

Figure 7-16: Quantifier-Free Boolean Algebra with Presburger Arithmetic (QFBAPA)

Motivation for QFBAPA. Note that a consequence of the quantifier elimination property of BAPA is that QFBAPA formulas define the same class of relations on sets and integers as BAPA formulas, so they essentially have the same expressive power. In general, QFBAPA formulas may be exponentially larger than the corresponding quantified BAPA formulas. However, it is often the case that the proof obligation (or other problem of interest) is already expressed in quantifier-free form. It is therefore interesting to consider the complexity of the satisfiability problem for QFBAPA.

Quantifier-free Presburger arithmetic. Quantifier-free PA is in NP because it has a small model property implying that satisfiable formulas have solutions whose binary representation is polynomial. The small model property for quantifier-free PA follows from the small model property for conjunctions of atomic formulas, which in turn follows from bounds on solutions of integer linear programming problems [206]. In practice, quantifier-free PA formulas can be solved using implementations such as CVC Lite [30] and UCLID [162].

Previous algorithms for QFBAPA. Existing algorithms for QFBAPA [266, 221, 202] run in non-deterministic exponential time, because they explicitly introduce a variable for each Venn region. The same exponential explosion occurs in our algorithm α [153, 154, 158] described in Section 7.3: when applied to a QFBAPA formula F, the algorithm α produces an exponentially large quantifier-free PA formula $\alpha(F)$. Quantifier-free PA is NP-complete, so checking satisfiability of $\alpha(F)$ will take non-deterministic exponential time in the size of F.

My result. I have previously used a divide-and-conquer algorithm to show that it is not necessary to simultaneously generate all Venn region variables, proving that QFBAPA is in PSPACE [180, Section 3]. I here give a stronger result, which shows that QFBAPA is in NP. In the process, I identify a natural encoding of QFBAPA formulas into polynomially-sized quantifier-free PA formulas. In my argument I use a recent result [86] that if an element is in an integer cone generated by a set of vectors X, then it is also in an integer cone generated by a "small" subset of X. This result implies that a system of equations with bounded coefficients, if satisfiable, has a *sparse solution* with only polynomially many non-zero variables, even if the number of variables in the system is exponential. As a result, instead of using exponentially many Venn region cardinality variables to encode relationships between sets, we can use polynomially many "generic" variables along with polynomially many indices that determine which Venn region cardinality each generic variable represents. In other words, every satisfiable QFBAPA formula has a witness of polynomial size, which indicates the values of integer variables in the original QFBAPA formula, lists the Venn regions that are non-empty, and indicates the cardinalities of these regions.

7.9.1 Constructing Small Presburger Arithmetic Formulas

Given a QFBAPA formula, this section shows how to construct a small quantifier-free Presburger Arithmetic formula. Section 7.9.2 then proves that this formula is equisatisfiable with the original one.

Figure 7-16 shows a context-free grammar for QFBAPA, which simply omits the quantifiers from the full BAPA grammar in Figure 7-4. We first separate PA and BA parts as in Section 7.3, by replacing $b_1 = b_2$ with $b_1 \subseteq b_2 \wedge b_2 \subseteq b_1$, replacing $b_1 \subseteq b_2$ with $|b_1 \cap b_2^c| = 0$, and then introducing integer variables k_i for all cardinality expressions $|b_i|$ occurring in the formula. With a constant increase in size, we obtain an equisatisfiable QFBAPA formula of the form $G \wedge F$ where G is a quantifier-free PA formula and F is of the form

$$\bigwedge_{i=0}^{p} |b_i| = k_i \tag{7.5}$$

We assume that $b_0 = \mathbf{1}$ and $k_0 = \mathsf{MAXC}$, i.e., that the first constraint is $|\mathbf{1}| = \mathsf{MAXC}$ and establishes the relationship between the universal set $\mathbf{1}$ and the maximal cardinality variable MAXC.

Let $y_1, \ldots, y_e$ be the set variables in $b_1, \ldots, b_p$. If we view each Boolean algebra formula b_i as a propositional formula, then for $\beta = (p_1, \ldots, p_e)$ where $p_i \in \{0,1\}$ let $[\![b_i]\!]_\beta \in \{0,1\}$ denote the truth value of b_i under the propositional valuation assigning the truth value p_i to the variable y_i. Let further s_β denote the Venn region associated with β, given by $s_\beta = \cap_{j=1}^{e} y_j^{p_j}$ where $y_j^0 = y_j^c$ is set complement and $y_j^1 = y_j$. We then have $|b_i| = \sum_{\beta \models b_i} |s_\beta|$. For the sake of analysis, for each $\beta \in \{0,1\}^e$ introduce a non-negative integer variable l_β denoting $|s_\beta|$. Then (7.5) is equisatisfiable with the exponentially larger PA formula

$$\bigwedge_{i=0}^{p} \sum \left\{ l_\beta \mid \beta \in \{0,1\}^e \wedge [\![b_i]\!]_\beta{=}1 \right\} = k_i \tag{7.6}$$

Instead of this exponentially large formula where β ranges over all 2^e propositional assignments, we will check the satisfiability of a smaller formula

$$G \;\wedge\; \bigwedge_{i=0}^{p} \sum \left\{ l_\beta \mid \beta \in \{\beta_1, \ldots, \beta_N\} \wedge [\![b_i]\!]_\beta{=}1 \right\} = k_i \tag{7.7}$$

where β ranges over a set of N assignments $\beta_1, \ldots, \beta_N$ for $\beta_i = (p_{i1}, \ldots, p_{ie})$ and p_{ij} are fresh free variables ranging over $\{0,1\}$. Let $d = p+1$. We are interested in the best upper bound $N(d)$ on the number of non-zero Venn regions over all possible systems of equations. In the sequel we show that $N(d)$ is polynomial and therefore polynomial in the size of the original QFBAPA formula. This result will prove that QFBAPA is in NP and give an effective bound on how to construct a quantifier-free PA formula for checking the satisfiability of a given QFBAPA formula.

Some details on PA encoding of QFBAPA. We next provide some details on the encoding of the formula (7.7) in quantifier-free PA, to convince the reader that the resulting formula is indeed polynomially large as long as N is polynomial in d. Let $c_{ij} = [\![b_i]\!]_{\beta_j}$ for $1 \leq i \leq p$ and $1 \leq j \leq N$. Then we need to express in quantifier-free PA the sum $\sum_{j=1}^{N} c_{ij} l_{\beta_j} = k_i$. It suffices to show how to efficiently express sums with boolean variable (as opposed to constant) coefficients. We illustrate this encoding for our particular example. Introduce a variable s_{ij} whose purpose is to store the value of the partial sum $s_{ij} = \sum_{k=1}^{j} c_{ik} l_{\beta_k}$. We introduce formula $s_{i0} = 0$ as well as

$$\begin{aligned}&(p \leftrightarrow [\![\lceil b_i \rceil]\!]_{\beta_j}) \wedge \\ &(p \;\rightarrow\; s_{ij} = s_{i(j-1)} + l_{\beta_j}) \wedge \\ &(\neg p \;\rightarrow\; s_{ij} = s_{i(j-1)})\end{aligned} \qquad (D_{ij})$$

where $[\![\lceil b_i \rceil]\!]_{\beta_j}$ denotes the propositional formula corresponding to b_i with propositional variables of β_j substituted for the corresponding sets. We therefore obtain dN polynomially sized expressions (D_{ij}), so if N is polynomial in d, the entire formula (7.7) is polynomial.

7.9.2 Upper Bound on the Number of Non-Zero Venn Regions

We next prove that the number of non-zero Venn regions can be assumed to be polynomial in d. Let $\mathbb{Z}$ denote the set of integers and $\mathbb{Z}_{\geq 0}$ denote the set of non-negative integers. We write $\sum X$ for $\sum\limits_{y \in X} y$.

Definition 38 *For $X \subseteq \mathbb{Z}^d$ a set of integer vectors, let*

$$\text{int\_cone}(X) = \{\lambda_1 x_1 + \ldots + \lambda_t x_t \mid t \geq 0 \wedge x_1, \ldots, x_t \in X \wedge \lambda_1, \ldots, \lambda_n \in \mathbb{Z}_{\geq 0}\}$$

denote the set of all non-negative integer linear combination of vectors from X.

To prove the bound on the number N of non-empty Venn regions from Section 7.9.1, we use a variation of the following result, established as Theorem 1(ii) in [86].

Fact 3 (Eisenbrand, Shmonina (2005)) *Let $X \subseteq \mathbb{Z}^d$ be a finite set of integer vectors and $M = \max\{(\max_{i=1}^{d} |x_j^i|) \mid (x_j^1, \ldots, x_j^d) \in X\}$ be the bound on the coordinates of vectors in X. If $b \in \text{int\_cone}(X)$, then there exists a subset $\tilde{X} \subseteq X$ such that $b \in \text{int\_cone}(\tilde{X})$ and $|\tilde{X}| \leq 2d \log(4dM)$.*

To apply Fact 3 to formula (7.6), let $X = \{x_\beta \mid \beta \in \{0,1\}^e\}$ where $x_\beta \in \{0,1\}^e$ is given by

$$x_\beta = ([\![b_0]\!]_\beta, [\![b_1]\!]_\beta, \ldots, [\![b_e]\!]_\beta).$$

Fact 3 implies is that if $(k_0, k_1, \ldots, k_p) \in \text{int\_cone}(X)$ where k_i are as in formula (7.6), then $(k_0, k_1, \ldots, k_p) \in \text{int\_cone}(\tilde{X})$ where $|\tilde{X}| = 2d \log(4d)$ (note that $M = 1$ because x_β are $\{0,1\}$-vectors). The subset $\tilde{X}$ corresponds to selecting a polynomial subset of N Venn region cardinality variables l_β and assuming that the remaining ones are zero. This implies that formulas (7.6) and (7.7) are equisatisfiable.

A direct application of Fact 3 yields $N = 2d \log(4d)$ bound, which is sufficient to prove that QFBAPA is in NP. However, because this bound is not tight, in the sequel we prove results that slightly strengthen the bound and provide additional insight into the problem.

7.9.3 Properties of Nonredundant Integer Cone Generators

Definition 39 *Let X be a set of integer vectors. We say that X is a* nonredundant integer cone generator *for b, and write $NICG(X, b)$, if $b \in \text{int\_cone}(X)$, and for every $y \in X$, $b \notin \text{int\_cone}(X \setminus \{y\})$.*

Lemma 40 says that if NICG(X,b) for some b, then the sums of vectors $\sum Y$ for $Y \subseteq X$ are uniquely generated elements of int_cone(X).

Lemma 40 *Suppose NICG(X,b). If $\lambda_1, \lambda_2 : X \to \mathbb{Z}_{\geq 0}$ are non-negative integer coefficients for vectors in X such that*

$$\sum_{x\in X} \lambda_1(x)x = \sum_{x\in X} \lambda_2(x)x \tag{7.8}$$

and $\lambda_1(x) \in \{0,1\}$ for all $x \in X$, then $\lambda_2 = \lambda_1$.

Proof. Suppose NICG(X,b), $\lambda_1, \lambda_2 : X \to \mathbb{Z}_{\geq 0}$ are such that (7.8) holds and $\lambda_1(x) \in \{0,1\}$ for all $x \in X$, but $\lambda_2 \neq \lambda_1$. If there are vectors x on the left-hand side of (7.8) that also appear on the right-hand side, we can cancel them. We obtain an equality of the form (7.8) for distinct λ_1', λ_2' with the additional property that $\lambda_1'(x) = 1$ implies $\lambda_2'(x) = 0$. Moreover, not all $\lambda_1'(x)$ are equal to zero. By $b \in$ int_cone(X), let $\lambda : X \to \mathbb{Z}_{\geq 0}$ be such that $b = \sum_{x\in X} \lambda(x)x$. Let x_0 be such that $\lambda_1'(x_0) = \min\{\lambda(x) \mid \lambda_1'(x) = 1\}$. By construction, $\lambda_1'(x_0) = 1$ and $\lambda_2'(x_0) = 0$. We then have, with x in sums ranging over X:

$$\begin{aligned} b &= \sum_{\lambda_1'(x)=1} \lambda(x)x + \sum_{\lambda_1'(x)=0} \lambda(x)x \\ &= \sum_{\lambda_1'(x)=1} (\lambda(x) - \lambda(x_0))x + \lambda(x_0) \sum_{\lambda_1'(x)=1} x + \sum_{\lambda_1'(x)=0} \lambda(x)x \\ &= \sum_{\lambda_1'(x)=1} (\lambda(x) - \lambda(x_0))x + \lambda(x_0) \sum \lambda_2'(x)x + \sum_{\lambda_1'(x)=0} \lambda(x)x \end{aligned}$$

In the last sum, the coefficient next to x_0 is zero in all three terms. We conclude $b \in$ int_cone$(X \setminus \{x_0\})$, contradicting NICG(X,b). ■

We write NICG(X) as a shorthand for NICG$(X, \sum X)$. Theorem 41 gives several equivalent characterizations of NICG(X).

Theorem 41 *Let $X \subseteq \{0,1\}^d$. The following statements are equivalent:*

1) *there exists a vector $b \in \mathbb{Z}^d_{\geq 0}$ such that NICG(X,b);*

2) *If $\lambda_1, \lambda_2 : X \to \mathbb{Z}_{\geq 0}$ are non-negative integer coefficients for vectors in X such that*

$$\sum_{x\in X} \lambda_1(x)x = \sum_{x\in X} \lambda_2(x)x$$

   and $\lambda_1(x) \in \{0,1\}$ for all $x \in X$, then $\lambda_2 = \lambda_1$.

3) *For $\{x_1, \ldots, x_n\} = X$ (for $x_1, \ldots, x_n$ distinct), the system of d equations expressed in vector form as*

$$\lambda(x_1)x_1 + \ldots + \lambda(x_n)x_n = \sum X \tag{7.9}$$

   has $(\lambda(x_1), \ldots, \lambda(x_n)) = (1, \ldots, 1)$ as the unique solution in $\mathbb{Z}^n_{\geq 0}$.

4) *NICG(X).*

Proof.

1) $\to$ 2): This is Lemma 40.

2) $\to$ 3): Assume 2) and let $\lambda_1(x_i) = 1$ for $1 \leq i \leq n$. For any solution λ_2 we then have $\sum_{x\in X} \lambda_1(x)x = \sum_{x\in X} \lambda_2(x)x$, so $\lambda_2 = \lambda_1$. Therefore, λ_1 is the unique solution.

3) $\rightarrow$ 4): Assume 3). Clearly $\sum X \in \mathrm{int\_cone}(X)$; it remains to prove that X is minimal. Let $y \in X$. For the sake of contradiction, suppose $\sum X \in \mathrm{int\_cone}(X \setminus \{y\})$. Then there exists a solution $\lambda(x)$ for (7.9) with $\lambda(y) = 0 \neq 1$, a contradiction with the uniqueness of the solution.

4) $\rightarrow$ 1): Take $b = \sum X$. ∎

Corollary 42 is used in [86] to establish the bound on the size of X with NICG(X). We obtain it directly from Lemma 40 taking $\lambda_2(x) \in \{0,1\}$.

Corollary 42 *If NICG(X) then for $Y_1, Y_2 \subseteq X$, $Y_1 \neq Y_2$ we have $\sum Y_1 \neq \sum Y_2$.*

The following lemma says that it suffices to establish bounds on the cardinality of X such that NICG(X), because they give bounds on all X.

Lemma 43 *If $b \in \mathrm{int\_cone}(X)$, then there exists a subset $\tilde{X} \subseteq X$ such that $b \in \mathrm{int\_cone}(\tilde{X})$ and NICG($\tilde{X}, b$).*

Proof. If $b \in \mathrm{int\_cone}(X)$ then by definition $b \in \mathrm{int\_cone}(X_0)$ for a finite $X_0 \subseteq X$. If not NICG(X_0, b), then $b \in \mathrm{int\_cone}(X_1)$ where X_1 is a proper subset of X_0. Continuing in this fashion we obtain a sequence $X_0 \supset X_1 \supset \ldots \supset X_k$ where $k \leq |X_0|$. The last element X_k satisfies NICG(X_k, b). ∎

Moreover, the property NICG(X) is hereditary, i.e. it applies to all subsets of a set that has it.[3]

Lemma 44 *If NICG(X) and $Y \subseteq X$, then NICG(Y).*

Proof. Suppose that NICG(X) and $Y \subseteq X$ but not NICG($Y, \sum Y$). Because $\sum Y \in \mathrm{int\_cone}(X)$, there is $z \in Y$ such that $\sum Y \in \mathrm{int\_cone}(Y \setminus \{z\})$. Then also $\sum Y \in \mathrm{int\_cone}(X \setminus \{z\})$, contradicting Lemma 40. ∎

The following theorem gives our bounds on $|X|$. As in [86], we only use Corollary 42 instead of the stronger Lemma 40, suggesting that the bound is not tight.

Theorem 45 *Let $X \subseteq \{0,1\}^d$ and NICG(X). Then*

$$|X| \leq (1 + \varepsilon(d))(d \log d) \tag{7.10}$$

where $\varepsilon(d) \leq 1$ for all $d \geq 1$, and $\lim_{d \to \infty} \varepsilon(d) = 0$.

Proof. Let $X \subseteq \{0,1\}^d$, NICG(X) and $N = |X|$. We first prove $2^N \leq (N+1)^d$. Suppose that, on the contrary, $2^N > (N+1)^d$. If $\sum Y = (x^1, \ldots, x^d)$ for $Y \subseteq X$, then $0 \leq x^j \leq N$ because $Y \subseteq \{0,1\}^d$ and $|Y| \leq N$. Therefore, there are only $(N+1)^d$ possible sums $\sum Y$. Because there are 2^N subsets $Y \subseteq X$, there exist two distinct subsets $U, V \in 2^X$ such that $\sum U = \sum V$. This contradicts Corollary 42. Therefore, $2^N \leq (N+1)^d$, so $N \leq d \log(N+1)$.

We first show that this implies $N \leq 2d \log(2d)$. We show the contrapositive. Suppose $N > 2d \log(2d)$. Then $\frac{N}{2d} > \log(2d)$ from which we have:

$$1 < \frac{2^{\frac{N}{2d}}}{2d} \tag{7.11}$$

[3] The reader familiar with matroids [250] might be interested to know that, for $d \geq 4$, the family of sets $\{X \subseteq \{0,1\}^d \mid \mathrm{NICG}(X)\}$ is not a matroid, because it contains multiple subset-maximal elements of different cardinality.

Moreover, $d \geq 1$ so $\frac{N}{2d} > \log(2d) \geq \log 2 = 1$, which implies

$$\log(1 + \frac{N}{2d}) \leq \frac{N}{2d} \tag{7.12}$$

From (7.11) and (7.12) we have, similarly to [86],

$$\begin{aligned} d\log(N+1) &< d\log(N\frac{2^{\frac{N}{2d}}}{2d}+1) = d\log(2^{\frac{N}{2d}}(\tfrac{N}{2d}+2^{-\frac{N}{2d}})) < d\log(2^{\frac{N}{2d}}(\tfrac{N}{2d}+1)) \\ &= d(\tfrac{N}{2d}+\log(1+\tfrac{N}{2d})) < d(\tfrac{N}{2d}+\tfrac{N}{2d}) = N. \end{aligned}$$

By contraposition, from $N \leq d\log(N+1)$ we conclude $N \leq 2d\log(2d)$. Substituting this bound on N back into $N \leq d\log(N+1)$ we obtain

$$\begin{aligned} N &\leq d\log(N+1) \leq d\log(2d\log(2d)+1) = d\log(2d(\log(2d)+\tfrac{1}{2d})) \\ &= d(1+\log d+\log(\log(2d)+\tfrac{1}{2d})) = d\log d(1+\tfrac{1+\log(\log(2d)+\frac{1}{2d})}{\log d}) \end{aligned}$$

so we can let

$$\varepsilon(d) = \frac{1+\log(\log d+1+\frac{1}{2d})}{\log d}.$$

It may be of interest for problems arising in practice that, for $d \leq 23170$ we have $\varepsilon(d) \leq \frac{5}{\log d}$ and thus $N \leq d(\log d + 5)$. ■

We can now define the function whose bounds we are interested in computing.

Definition 46 $N(d) = \max\{|X| \mid X \subseteq \{0,1\}^d, \mathit{NICG}(X)\}$

Theorem 45 implies $N(d) \leq (1+\varepsilon(d))(d\log d)$.

7.9.4 Notes on Lower Bounds and Set Algebra with Real Measures

While we currently do not have a tight lower bound on $N(d)$, in this section we show, in sequence, the following:

1. $d \leq N(d)$ for all d;
2. $N_R(d) = d$ if we use real variables instead of integer variables;
3. $N(d) = d$ for $d \in \{1,2,3\}$;
4. for $d + \lfloor \frac{d}{4} \rfloor \leq N(d)$ for $4 \leq d$.

We first show $d \leq N(d)$.

Lemma 47 *Let* $X = \{(x_i^1, \ldots, x_i^d) \mid 1 \leq i \leq n\}$ *and*

$$X^+ = \{(x_i^1, \ldots, x_i^d, 0) \mid 1 \leq i \leq n\} \cup \{(0, \ldots, 0, 1)\}$$

Then $\mathit{NICG}(X)$ *if and only if* $\mathit{NICG}(X^+)$.

Corollary 48 $N(d) + 1 \leq N(d+1)$ *for all* $d \geq 1$.

Proof. Let $X \subseteq \{0,1\}^d$, NICG(X), and $|X| = N(d)$. Then NICG(X^+) by Lemma 47 and $|X^+| = N(d)+1$, which implies $N(d+1) \geq N(d)+1$. ■

Note that we have $N(1) = 1$ because there is only one non-zero $\{0,1\}$ vector in one dimension. From Corollary 48 we obtain our lower bound, with standard basis as NICG.

Lemma 49 $d \leq N(d)$. *Specifically, NICG*$(\{e_1,\ldots,e_d\})$.

Note that for $X = \{e_1,\ldots,e_d\}$ we have int_cone$(X) = \mathbb{Z}^d_{\geq 0}$, which implies that X is a *maximal* NICG, in the sense that no proper superset $W \supset X$ for $W \subseteq \{0,1\}^d$ has the property NICG(W).

Real-valued relaxation of QFBAPA. It is interesting to observe that, for a variation of the QFBAPA problem over *real numbers*, which we call QFBALA (Quantifier-Free Boolean Algebra with Linear Arithmetic), we have $N'(d) = d$ as a lower *and upper* bound for every d.

We define QFBALA similarly as QFBAPA, but we use real (or rational) linear arithmetic instead of integer linear arithmetic and we interpret $|A|$ is some real-valued measure of the set A. A possible application of QFBALA are generalizations of probability consistency problems such as [34, Page 385, Example 8.3]. Set algebra operations then correspond to the σ-algebra of events, and the measure of the set is the probability of the event. Another model of QFBALA is to interpret sets as finite disjoint unions of intervals contained in $[0,1]$, and let $|A|$ be the sum of the lengths of the disjoint intervals making up A.

The conditions we are using on the models is a version of Lemma 26: 1) for two disjoint sets A, B, we have $|A \cup B| = |A| + |B|$, and 2) if $|C| = p$ and $0 \leq q \leq p$, then there exists $B \subseteq C$ such that $|B| = q$. (In addition, if the model allows $|A| = 0$ for $A \neq \emptyset$, then we introduce an additional propositional variable for each Venn region variable to track its emptiness, similarly to the set infinity flags from Section 7.8.1.)

We can reduce the satisfiability of QFBALA to the satisfiability of a quantifier-free linear arithmetic formula over reals and a formula of the form (7.6) but with l_β non-negative real values instead of integer values. We then reduce formula (7.6) to a formula of the form (7.6). The question is then, what can we use as the bound $N'(d)$ for QFBALA problems? This question reduces to following. Define convex cone generated by a set of vectors by

$$\text{cone}(X) = \{\lambda_1 x_1 + \ldots + \lambda_t x_t \mid t \geq 0 \land x_1,\ldots,x_t \in X \land \lambda_1,\ldots,\lambda_n \geq 0\}$$

where $\lambda_1,\ldots,\lambda_n \in \mathbb{R}$ are non-negative real coefficients. If $b \in \text{cone}(X)$, what bound can we put on the cardinality of a subset $\tilde{X} \subseteq X$ such that $X \in \text{cone}(\tilde{X})$? Note that d is a lower bound, using the same example of unit vectors as X. In the case of real numbers, Carathéodory's theorem [63] states that d is an upper bound as well: $b \in \text{cone}(\tilde{X})$ for some $\tilde{X}$ of cardinality at most d. We can also explain that $N'(d) = d$ using the terminology of linear programming [227]. The equations (7.6) along with $l_\beta \geq 0$ for $\beta \in \{0,1\}^e$ determine a polytope in $\mathbb{R}^{2^e}$, so if they have a solution, they have a solution that is a vertex of the polytope. The vertex in $\mathbb{R}^{2^e}$ is the intersection of 2^e hyperplanes, of which at most d are given by (7.6), so the remaining ones must be hyperplanes of the form $l_\beta = 0$. This implies that at least $2^e - d$ coordinates of the vertex are zero and at most d of them can be non-zero.

Note that the linear arithmetic formula in Figure 7-13 is valid not only over integers but also over real numbers. The negation of this formula is a linear programming problem that is unsatisfiable over not only integers but also over reals. In that sense, QFBALA is a relaxation of QFBAPA, and can be used as a sound (but incomplete) method for proving the absence of solutions of a QFBAPA formula.

***$N(d) = d$* for *$d \in \{1, 2, 3\}$*.** We next show that for $d \in \{1, 2, 3\}$ not only $d \le N(d)$ but also $N(d) \le d$.

Lemma 50 *$N(d) = d$ for $d \in \{1, 2, 3\}$.*

Proof. By Corollary 48, if $N(d+1) = d+1$, then $N(d)+1 \le d+1$ so $N(d) \le n$. Therefore, $N(d) = 3$ implies $N(2) = 2$ as well, so we can take $d = 3$.

If $N(d) > d$, then there exists a set X with NICG(X) and $|X| > d$. From Lemma 44, a subset $X_0 \subseteq X$ with $|X| = d+1$ also satisfies NICG(X_0). Therefore, $N(3) = 3$ is equivalent to showing that there is no set $X \subseteq \{0, 1\}^3$ with NICG(X) and $|X| = 4$.

Consider a possible counterexample $X = \{x_1, x_2, x_3, x_4\} \subseteq \{0, 1\}^3$ with $b \in X$. By previous argument on real-value relaxation, $N'(3) = 3$, so b is in convex cone of some three vectors from X, say $b \in \text{cone}(\{x_1, x_3, x_3\})$. On the other hand, $b \notin \text{int\_cone}(\{x_1, x_3, x_3\})$. If we consider a system $\lambda_1 x_1 + \lambda_2 x_2 + \lambda_3 x_3 = b$ this implies that such system has solution over non-negative reals, but not over non-negative integers. This can only happen if in the process of Gaussian elimination we obtain coefficients whose absolute value is more than 1. The only set of three vectors for which this can occur is $X_1 = \{(0, 1, 1), (1, 0, 1), (1, 1, 0)\}$ We then consider all possibilities for the fourth vector in X, which, modulo symmetry of coordinates are $(0, 0, 0)$, $(1, 1, 1)$, $(1, 1, 0)$, and $(1, 0, 0)$. However, adding any of these vectors violates the uniqueness of the solution to $\lambda_1 x_1 + \lambda_2 x_2 + \lambda_3 x_3 + \lambda_4 x_4 = \sum X$, so NICG$(X)$ does not hold by Theorem 41, condition 3). ∎

***$N = \frac{5}{4}d - \frac{3}{4}$* lower bound.** I next show that there exists an example $X_5 \subseteq \{0, 1\}^4$ with NICG(X_5) and $|X_5| = 5$. From this it follows that $N(d) > d$ for all $d \ge 4$.

Consider the following system of 4 equations with 5 variables, where all variable coefficients are in $\{0, 1\}$. (I found this example by narrowing down the search using the observations on minimal counterexamples in the proof of Lemma 50.)

$$\begin{array}{lcl}
\lambda_1 + \lambda_2 + \lambda_3 & = & 3 \\
\quad\;\; \lambda_2 + \lambda_3 + \lambda_4 & = & 3 \\
\lambda_1 \quad\;\; + \lambda_3 + \lambda_4 + \lambda_5 & = & 4 \\
\lambda_1 + \lambda_2 \quad\;\; + \lambda_4 + \lambda_5 & = & 4
\end{array} \tag{7.13}$$

Performing Gaussian elimination yields an equivalent upper-triangular system

$$\begin{array}{rcl}
\lambda_1 + \lambda_2 + \lambda_3 \qquad\qquad & = & 3 \\
\lambda_2 + \lambda_3 + \lambda_4 \qquad & = & 3 \\
\lambda_3 + 2\lambda_4 + \lambda_5 & = & 4 \\
3\lambda_4 + 2\lambda_5 & = & 5
\end{array}$$

From this form it easy to see that the system has $(\lambda_1, \lambda_2, \lambda_3, \lambda_4, \lambda_5) = (1, 1, 1, 1, 1)$ as *the only solution* in the space of non-negative integers. Note that all variables are non-zero in this solution. (In contrast, as discussed above, because the system is satisfiable, it must have a solution in non-negative reals where at most 4 coordinates are non-zero; an example of such solution is $(\lambda_1, \lambda_2, \lambda_3, \lambda_4, \lambda_5) = (0, 1.5, 1.5, 0, 2.5)$.) The five columns of the system (7.13) correspond to the set of vectors $X_5 = \{(1,0,1,1), (1,1,0,1), (1,1,1,0), (0,1,1,1), (0,0,1,1)\}$ such that NICG(X_5). The set X_5 is also a maximal NICG, because adding any of the remaining 9 non-zero vectors in $\{0,1\}^4 \setminus X_5$ results in a set that is not NICG.

Using k identical copies of X_5 (with 4 equations in a group mentioning a disjoint set of 5 variables) we obtain systems of $4k$ equations with $5k$ variables such that the only solution is a vector $(1,\ldots,1)$ of all ones. By adding p unit vector columns for $1 \leq p \leq 3$, we also obtain systems of $4k+p$ equations with $5k+p$ variables, with

$$N = \frac{5}{4}d - \frac{p}{4} = d + \left\lfloor \frac{d}{4} \right\rfloor \geq \frac{5}{4}d - \frac{3}{4}$$

which, in particular, shows that $N = d$ upper bound is invalid for all $d \geq 4$.

This argument shows that there exist maximal NICG of size larger than d for $d \geq 4$. As we have remarked before, the set of d unit vectors is a maximal NICG for every d, which means that, unlike linearly independent sets of vectors over a field or other independent sets in a matroid [250], there are maximal NICG sets of different cardinality.

Note also that X_5 is not a Hilbert basis [229]. Namely $(1,1,1,1) \in \text{cone}(X_5) \setminus \text{int\_cone}(X_5)$ because $(1,1,1,1) = 1/3((1,0,1,1)+(1,1,0,1)+(1,1,1,0)+(0,1,1,1))$. This illustrates why previous results on Hilbert bases do not directly apply to the notion of NICG.

7.9.5 A decision procedure for QFBAPA

Using Theorem 45 we obtain a non-deterministic polynomial-time algorithm for checking QFBAPA satisfiability. For formulas generated from verification, it is likely that a QFBAPA decision procedure implementation can effectively use bounds smaller than $(1+\varepsilon(d))d\log d$ to find counterexamples and to prove their absence, as follows.

1. Attempt to find counterexamples for small N. If a counterexample for any N is found, it is a valid counterexample. One could expect that such counterexamples would often be found by "small scope hypothesis" [131] for typical formulas arising in software verification.

2. If no counterexample is found for small N, then the decision procedure can use the bound $N = d$ with real linear arithmetic and try to prove the absence of solutions. No solutions found means that the original QFBAPA problem has no solutions either. The example in Figure 7-13 and the experience from [82, Section 8] suggest that this approach would often succeed in proving the absence of solutions for unsatisfiable QFBAPA formulas.

3. Finally, if a solution is found in real numbers but not for small N in integers, then the system can use the bound $N = (1+\varepsilon(d))d\log d$, which gives a definite answer thanks to Theorem 45.

The first two steps can be viewed as heuristics for finding the answer faster in common cases; their usefulness remains to be experimentally evaluated.

7.10 Related Work

Our result in Section 7.4 is the first complexity bound for the first-order theory of BAPA. A preliminary version of this result appears in [153, 158] and most of the content of this chapter is presented in [154]. The decidability for BAPA, presented as BA with equicardinality constraints was shown in [90], see Section 7.1. A decision procedure for a special case of BAPA was presented in [265], which allows only quantification over *elements* but not over *sets* of elements. [202] also examines quantifier-free formulas and show how to combine quantifier-free BA constraints with additional constraints using "bridging functions". Bridging functions satisfy homomorphism-like properties and generalize the cardinality operator; we expect that the quantifier-elimination techniques of this chapter can be generalized in this direction as well. An interesting case of a bridging function is a probability measure of a set, which arises in probability consistency problems such as [34, Page 385, Example 8.3]. Our observations in Section 7.9 imply that a very general version of a probability constency problem that allows arbitrary expressions for intersection, union, and completement of events is in NP and therefore NP-complete.

[221] presents quantifier elimination and the decidability of a single-sorted version of BAPA that only contains the set sort. Note that bound integer variables can be simulated using bound set variables, but there are notational and clear efficiency reasons to allow integer variables in BAPA.

To our knowledge, our result in Section 7.9 is the only decision procedure for a logic with sets and cardinality constraints that does not explicitly construct all set partitions. Using a new form of small model property, the "small number of non-zero variables property", we obtained a non-deterministic polynomial-time algorithm that can be solved by producing polynomially large quantifier-free Presburger arithmetic formulas. A polynomial bound sufficient for our result can be derived from [86]. In addition to slight improvements in the bounds, in Section 7.9 we suggest that there is an interesting structure behind these bounds. We introduced the notion of nonredundant integer cone generators and proved additional results that may help us understand their properties and eventually establish tight bounds on their size. We note that previous results such as [229] consider matroids and Hilbert bases. In contrast, nonredundant integer cone generators are the natural notion for our problem. As we remark in Section 7.9.4, the sets of vectors X with $\text{NICG}(X)$ do not form a matroid, and maximal $\text{NICG}(X)$ need not be a Hilbert basis. Note also that the equations generated from QFBAPA problems are more difficult than set packing and set partitioning problems [22] because integer variables are not restricted to be $\{0, 1\}$.

Presburger arithmetic. The original result on decidability of PA is [212]. The space bound for PA was shown in [91]. The matching lower and upper bounds for PA were shown in [32], see also [145, Lecture 24]. An analysis parameterized by the number of quantifier alternations is presented in [217]. Our implementation uses quantifer-elimination based Omega test [215]. Among the decision procedures for full PA, [50] is the only proof-generating version, and is based on [64]. Decidable fragments of arithmetic that go beyond PA are described in [42, 43].

Reasoning about Sets. The first results on decidability of BA of sets are from [176], [3, Chapter 4] and use quantifier elimination, from which one can derive small model property. [145] gives the complexity of the satisfiability problem for arbitrary BA. [182] study unification in Boolean rings. The quantifier-free fragment of BA is shown NP-complete in [181]; see [160] for a generalization of this result using the parameterized complexity of the Bernays-Schönfinkel-Ramsey class of first-order logic [37, Page 258]. [48] gives an overview of several fragments of set theory including theories with quantifiers but no cardinality constraints and theories with cardinality constraints but no quantification over sets. The decision procedure for quantifier-free fragment with cardinalities in [48, Chapter 11] introduces exponentially many integer variables to reduce the problem to PA. Among the systems for interactively reasoning about richer theories of sets are Isabelle [201], HOL [110], PVS [204]. First-order frameworks such as Athena [15] can use axiomatizations of sets along with calls to resolution-based theorem provers [244, 248] to reason about sets.

Combinations of Decidable Theories. The techniques for combining *quantifier-free* theories [197, 223] and their generalizations such as [239, 263, 266, 264, 241] are of great importance for program verification. In this chapter we mostly focused on quantified formulas, which add additional expressive power in writing concise specifications. Among the general results for quantified formulas are the Feferman-Vaught theorem for products [90] and term powers [155, 156]. Description logics [18] also support sets with cardinalities as well as relations, but do not support quantification over sets. While we have found quantifier elimination to be useful, many problems can be encoded in quantifier-free formulas, which motivates the ongoing work in Section 7.9.

Analyses of Linked Data Structures. In addition to the new technical results, one of the contributions of this chapter is to identify the uses of our decision procedure for verifying data structure consistency. We have shown how BAPA enables the verification tools to reason about sets and their sizes. This capability is particularly important for analyses that handle dynamically allocated data structures where the number of objects is statically unbounded [190, 259, 226]. Recently, these approaches were extended to handle the combinations of the constraints representing data structure contents and constraints representing numerical properties of data structures [224, 55]. Our result provides a systematic mechanism for building precise and predictable versions of such analyses. Among other constraints used for data structure analysis, BAPA is unique in being a complete algorithm for an expressive theory that supports arbitrary quantifiers.

7.11 Conclusion

Motivated by static analysis and verification of relations between data structure content and size, I presented an algorithm for deciding the first-order theory of Boolean algebra with Presburger arithmetic (BAPA), established the precise complexity of BAPA, showing that it is identical to the complexity of PA, implemented the algorithm and applied it to discharge verification conditions. Our experience indicates that the algorithm is useful as a component of a data structure verification system. I established that the quantifier-free fragment of BAPA is in NP and presented an efficient encoding into quantifier-free Presburger arithmetic. I believe that this encoding can be a basis for scalable decision procedures.

Chapter 8

Conclusions

This dissertation has presented the Jahob verification system. Jahob is based on a subset of the commonly used implementation language Java, with specifications written in the language of the popular interactive theorem prover Isabelle. I have shown that, in this general setup, it is possible to implement a verification system that effectively proves strong properties of data structure implementations and data structure uses. My colleagues and I have used the system to verify imperative data structures such as hash tables, linked lists, trees, simplified skip lists, and purely functional versions of sorted binary search trees (including the remove operation). We have verified not only global but also instantiable versions of several of these data structures, supporting the typical form of usage of containers in Java programs. While we have not aimed at necessarily verifying full functional correctness, we succeeded in proving key correctness properties, such as the fact that removal from a sorted binary search tree or a hash table indeed removes precisely the given (key,value)-binding from the relation stored in the data structure.

This dissertation focuses on the reasoning engine within Jahob, which is based on a combination of several different reasoning techniques. The key to deploying these techniques is the idea of approximating complex formulas in higher-order logic with formulas in a simpler logic.

First-order resolution-based theorem provers are among the most robust techniques I have used in Jahob. These provers turned out to be relatively fast and their resource control strategies proved extremely useful for software verification. The ability to reason about general graphs (and not just about trees, for example) made it possible to naturally implement assume/guarantee reasoning by inlining procedure specifications. This in turn enabled data structure operations to be written in a recursive way, making their verification simpler.

The use of monadic second-order logic over trees with field constraint analysis offered a very high degree of automation for reasoning about reachability in the presence of imperative data structure updates. This technique was particularly useful whenever we were able to use the string mode of the MONA tool; we are exploring techniques for enabling its use for a wider range of data structures. In tree mode and in the presence of larger proof obligations, we have found the performance of MONA to decrease significantly, suggesting that a combination of symbolic and automata-based decision procedures would be beneficial in the future.

Finally, I deployed a new decision procedure, for Boolean Algebra with Presburger Arithmetic (BAPA). I first formalized a decision procedure for BAPA by reduction to Presburger arithmetic and established the complexity for the BAPA decision problem, showing that it is alternating doubly exponential with a linear number of alternations, thus matching the complexity of Presburger arithmetic. Our preliminary experience of using BAPA shows that, despite its high complexity, BAPA is effective on small formulas, which made it useful in combination with other reasoning procedures. The high complexity of BAPA led me to examine the quantifier-free fragment of BAPA. I proved that quantifier-free BAPA (which is trivially NP-hard) belongs to NP. I showed the membership in NP as a consequence of bounds on the size of nonredundant integer cone generators. This proof gives an effective algorithm for the quantifier-free BAPA fragment.

Overall, these results confirm my hypothesis that different techniques have different strengths and that their synergistic use enables the verification of a broader class of properties than attainable using any single technique in isolation. An approach based on splitting proof obligations, although simple, turned out to be very successful on small methods. A more sophisticated splitting strategy is likely to be needed for larger proof obligations, and may require a tighter integration of reasoning procedures.

I was initially surprised by several discoveries that my coauthors and I made in this project.

- The idea of using Isabelle notation as a semantic and syntactic basis for Jahob formulas helped me avoid making arbitrary design decisions, and helped us debug the system. Using Isabelle's automated provers worked better than I initially expected given the interactive nature of Isabelle as a prover, and was especially helpful in the initial stages of the project, before translations to more tractable logic fragments were available.

- Our experience with first-order theorem provers contained many pleasant surprises. Simple arithmetic axioms turned out to be sufficient in our examples, despite incompleteness in comparison to Nelson-Oppen style approaches. The use of ghost fields and recursion allowed us to verify recursive data structures, pointing to cases where simple hints from programmers can eliminate the need to use transitive closure. (Yet providing these hints can be fairly tricky for the cases where a local data structure operation results in non-local changes to ghost specification fields of multiple objects.) In the translation of higher-order logic to first-order logic, the simplification that results from omitting sorts seems to have outweighed any advantage of using sorts expressed as unary predicates to cut down the branching factor in search. Perhaps most surprising was the fact that omitting sorts in the translation is not only complete but also sound, the only requirement being that the conflated sorts are of equal cardinality. Finally, simple assumption filtering heuristics seem to work surprisingly well in choosing relevant assumptions, despite the fact that they essentially ignore the semantics of formulas and only look at their lexical structure, and despite the fact that theorem provers have built-in heuristics for selection of clauses.

- Field constraint analysis initially grew out of a desire to eliminate the restrictions on systems like the Pointer Assertion Logic Engine [190], where non-tree fields must be uniquely determined by their edges. This was even perceived as a fundamental limitation of the approach. I confess that even our first idea ignored the polarity of fields and would have resulted in an unsound translation. Taking into account the polarity of fields made the approach sound, but it was clear that it was an approximation in general. Thomas Wies made a remarkable claim that field constraint elimination could be complete under certain restrictions; my proof attempt led to a small correction to the algorithm which made this claim actually true. The result was a technique complete for proving the preservation of field constraints themselves. On the other hand, this technique, although powerful for proving many structural invariants, seems too weak for reasoning about uninterpreted functions themselves. Indeed, the technique turned out to be insufficient for proving postconditions that characterize container operations where the abstraction function traverses non-tree fields, or for proving the injectivity of otherwise unconstrained fields. This is one of the justifications for combining field constraint analysis with first-order and Nelson-Oppen style provers.

- The fact that BAPA is decidable may come somewhat as a surprise (see [265, Section 6]), but is a fact described in [90]. The fact that the asymptotic complexity ends up being the same as for Presburger arithmetic is a consequence of the fact that the algorithm I present preserves the number of quantifier alternations, but also depends on the way in which we measure very high complexity classes. In practice, the difference between Presburger arithmetic and BAPA is significant and our experiments provided a convincing demonstration of how doubly-exponential lower bounds can effectively result in algorithms that work up to inputs of certain size and then suddenly stop working. Therefore, although my work has delivered algorithms with as good worst-case complexity on quantified BAPA as we can hope for, it is more realistic to focus on using quantifier-free BAPA fragments in verification. My subsequent work shows membership of quantifier-free BAPA in NP, showing for the first time how to avoid the exponential explosion in the number of integer variables. I expect this result to lead to substantially more efficient algorithms for BAPA, generalizing previous uses of Presburger arithmetic in many verification systems [198, 44].

I expect these results to be a good starting point for the automated verification of complex software systems. The algorithms for checking formula validity address the problem of propagating the descriptions of reachable states through loop-free code and therefore address verification in the presence of loop invariants and procedure contracts. Moreover, specification inference techniques such as [253] can use the algorithms in this dissertation to synthesize loop invariants and therefore propagate reachable states across arbitrary regions of code.

8.1 Future Work

As I am writing the final lines of my dissertation, I am witnessing a number of interesting results in verifying complex properties of software. The research community is adopting new specification languages for reasoning about software [130], unifying the interests of type theorists and program analysis researchers. New results demonstrate the verification of larger classes of properties and the ability to scale the existing techniques to larger code bases. Previously disjoint research communities are finding a common language in building tools that increase software reliability, product groups are using analysis tools that require the use of specifications [74], and commercial static analysis products are deployed to detect software defects [70]. These developments are shaping software analysis and verification as a field with a sound theoretical basis and concrete results that help improve software productivity in practice.

By using general purpose implementation and specification languages, Jahob identifies techniques for reasoning about expressive formulas as the core technology for building future verification tools. This view unifies existing analyses and enables us to build new sophisticated analyses for complex properties. Jahob identifies particular procedures for reasoning about formulas, shows that these procedures can verify interesting classes of expressive properties, and proposes a mechanism for combining these procedures using a unified specification language. Jahob aims to improve both the precision and the scalability of verification, by analyzing properties whose automation was beyond the reach of previous tools, and, at the same time, embracing modular reasoning that makes such precise analyses feasible. I therefore believe that Jahob's basic architecture is a good starting point for future developments. The following paragraphs summarize some of the possible future research directions. Although inspired by my experience with Jahob, I believe that they also apply to other verification systems of similar architecture.

Encapsulation. The notion of encapsulation in object-oriented languages has been explored from many angles, with the Boogie methodology [27, 29] and ownership types [58, 39, 57] being among the most promising approaches. However, it is not yet clear that the community has identified a solution that is both amenable to verification and sufficiently flexible. Jahob's encapsulation mechanisms are still part of an ongoing work. I expect that the result will be a useful compromise between flexibility, verifiability, and conceptual simplicity. Such a solution can contribute to the design of future languages with verifiable imperative data structures.

Aliasing. Jahob encodes aliasing in a sound way, modelling memory as a set of functions. I believe that approaches in this spirit provide an appropriate logical foundation for reasoning about imperative languages. They are also compatible with notations such as role logic [159, 151] and separation logic [130], which can be embedded into the Isabelle notation by defining appropriate shorthands. Incorporating such approaches could make specification and verification easier by taking advantage of common cases (bounded number of aliases and disjointness of data structures), without restricting the set of analyzable programs.

Executing and constraint solving for specifications. Most specifications that we have encountered in software verification are executable, and their execution is extremely useful in identifying bugs in both the specification and the implementation. Modular execution of set specifications requires constraint solving to synthesize states that satisfy procedure preconditions [179, 40, 140]. The generality of HOL makes it difficult to execute and perform constraint solving directly on HOL specifications [246]. Moreover, it opens up the question of the semantics of encapsulation in the presence of executable specifications, because the specification language is more expressive than the implementation language, which affects the notion of observability. The use of bounded quantification as in [150] appears to be a promising approach to identify specifications that are easier to execute and model check. This approach supports the expression of a wide range of safety properties, and yet does not make the construction of proofs much more difficult compared to the use of unbounded quantifiers.

Language for correctness arguments. A useful feature of a verification system is allowing users to manually specify correctness proofs of the properties of interest. Users should be able to specify proofs of arbitrarily complex properties, such specifications should be compatible with the way users reason about the system informally, and the system should be able to check the supplied proofs efficiently. Jahob partially achieves this goal using its interface to interactive theorem provers (Isabelle itself supports a natural deduction proof style [249], and so does the Athena system [15], which could also be connected to Jahob).

However, this approach somewhat obscures the verification problem because the interactive theorem proving environment is independent from Jahob. The user critically relies on the names of identifiers and subformulas to map the intuition about the specified program into the intuition needed to complete the interactive proof. Among the possible solutions for bridging this gap are allowing the program verifier to be interactive and embedding programs into the logic of the theorem prover. Jahob's use of `noteThat...from` statements presents a simple step towards incorporating a proof presentation system into a program verifier. However, for completeness, Jahob would need additional specification constructs that correspond to natural-deduction proof steps. Such constructs would lead to a high-level proof system for reasoning about software that serves as a platform for both manually and automatically constructed correctness arguments for imperative programs.

Statistical and heuristic reasoning. Once a platform for sound reasoning is established, it is possible to use statistical and heuristic techniques to infer possible invariants and program annotations, relying on the soundness of the platform to detect incorrect inferences. Statistical techniques have been applied to the inference of finite state machines [7, 88, 147], and to the inference of likely invariants [89] that, once inferred, can be statically verified [194]. Techniques that more tightly integrate dynamic and static inference have been proposed [258] and might ultimately be necessary to make this approach feasible.

Decision procedures. Given that propositional logic, one of the simplest logics, is NP-complete, it is natural to ask how expressive a logic can be and still belong to NP. Such a logic can then be encoded into SAT [162], or decided using a generalization of satisfiability solving [30]. The fact that I have found QFBAPA to be in NP is promising in this regard. Other expressive logics are likely to have useful NP fragments that yet remain to be identified. NP logics are unlikely to support quantifiers, but often program structure can be used to direct quantifier instantiation [254].

Proof search. Efficient decision procedures are useful for reasoning about well-understood domains, but it is clear that they are not sufficient by themselves for reasoning about software. Ultimately, the space of software systems that we can reason about is characterized by our ability to construct correctness arguments, which suggests that we should explore proof systems and automated proof search. Proof search in first-order logic is a mature research topic, yet proof construction in the presence of more specific axioms is an active area of research [213, 103].

New interfaces between verifiers and provers. Experience with assumption filtering in Jahob (Section 5.4) and other systems [186] suggests that closer integration between theorem provers and program verifiers is necessary, where, for example, the verifier indicates which assumptions are likely to be relevant, or which quantifier instantiations are likely to be fruitful. Ultimately, we can view a program verifier as a theorem prover tuned for reasoning about certain kinds of state transforming functions [183].

Loop invariant synthesis. Loop invariant synthesis can in principle be cast as a theorem proving problem, but it would require theorem provers that can deal with mathematical induction, which is challenging and supported in a small number of theorem proving systems [139, 45]. Static analysis can be used to synthesize loop invariants for particular domains [35]. Among the most promising approaches for deriving complex program-dependent properties are predicate abstraction [24], as well as symbolic shape analysis [254] developed by Thomas Wies and deployed in Jahob. The need for the analysis to adapt to the program suggests that a form of demand-driven approach will be useful [120, 168].

Formula approximation in program analysis. I believe that formula approximation techniques have a great potential in automated reasoning about software. Sections 4.3 and 4.4 describe the use of formula approximation to enable specialized reasoning procedures to communicate using a common expressive language. Formula approximations are also potentially useful for synthesizing loop invariants and transfer functions. This is a natural consequence of viewing static analysis as an approximation process [69] while representing the analysis domain elements as formulas [114, 260, 220]. I believe that this idea can be used to focus the effort of theorem provers to small formulas, avoiding the encoding of entire domain elements.

Application-specific properties. One of my future goals is to explore the verification of application-specific properties of software that contains complex data structures. In this context, the results on data structure verification are significant because they show that the verification of complex user-specified properties is possible, and because they enable the verification of applications to view data structures as sets and relations, avoiding the need to reason about data structure internals. Ultimately, automated verification of deep application-specific properties is likely to require systems that have knowledge of the particular domain. One approach to building such systems is to develop techniques that enable domain experts to customize the verifier. Customized systems have proven to be very useful [226, 219, 87] but need to be made more accessible and need to work at a higher level of abstraction.

Interactive deployment of verification tools. Given the potential of integrated development environments such as Eclipse [100] to improve programmer productivity, it is interesting to consider the use of verification tools in such environments. So far, we have primarily used Jahob as a command-line tool, but Peter Schmitt from Freiburg University implemented a simple Eclipse plugin that makes it possible to invoke Jahob from Eclipse. Jahob's modular verification approach is in principle appropriate for interactive use because it allows the verification on a per method basis. The integration of the Spec# verifier Boogie [27] into an interactive deployment environment suggests that such an approach is indeed feasible, at least for verifying simple program properties. To make Jahob verification sufficiently fast for interactive use, it would be desirable to make the granularity of verification tasks even smaller, to employ more efficient reasoning techniques, and to make the entire verification task more demand-driven and amenable to incremental computation. These directions lead to challenging algorithmic questions, especially in the context of verifying complex program properties. One promising step in this direction is the use of caching to avoid similar invocations of decision procedures, which is incorporated in Jahob's Bohne plugin [254].

Supporting program transformations. An important class of tasks in integrated development environments is different types of program refactoring [240]. Verifying the correctness of such transformations may require proving properties such as equivalence of program fragments, which is difficult in general but feasible for loop-free code. What makes verification of factoring easier is that transformations are not arbitrary and that the type and the parameters of a refactoring can provide proof hints on why the transformation is valid.

In the context of interactive transformations supported by sophisticated program analyses, it may be interesting to revisit programming by refinement, whose wide practical use has so far been limited [2, 1, 47, 49, 20], but appears cost-effective when full specification is desired. One challenge is supporting refinements that introduce loops while keeping the program development model intuitive for programmers.

More expressive specifications. The use of verification and analysis tools in interactive program development suggests another avenue for future research: extending Jahob's specification language and the underlying analysis. First, it is useful to allow specifications that mention sequences of method invocations and other statements. Higher-order logic in Jahob is sufficiently expressive, but its particular interpretation in terms of program states is limited to reasoning about a finite number of program points. On the other hand, queries in an interactive context are likely to cross-cut larger pieces of code. It would therefore be useful for a specification language to support temporal operators that can refer to past and future events in program execution, as in temporal logic [209], and to support invocations of methods within specifications, as in algebraic specifications [116], as in approaches based on embedding of programs into the specification logic [139], as in the rich formalism of the KeY system [5], or, for pure methods, as in JML [46].

An even more ambitious step in increasing the expressive power of specification languages is embedding not only the semantics but also the syntax of the language into logical formulas. This approach would allow checking queries that refer to program structure and would be useful for program understanding.

Beyond logic. An orthogonal aspect of specification language expressiveness is its robustness with respect to specification errors, which are even more likely to occur in an interactive context than in more stable program annotations. Mechanisms for detecting and avoiding errors in specifications are essential for making specifications concise and closer to natural language and are likely to require domain knowledge. This direction is therefore important for making the specification process accessible to software developers and software designers. Ultimately, its use in both implementation and specification languages can blur the distinction between software developers and users, holding out the hope to eliminate the software development bottleneck and unlocking the continuously growing computing potential of our society's infrastructure.

8.2 Final Remarks

More than ten years ago, I became fascinated by the field of automated software verification, only to be subsequently disillusioned by the tremendous gap between informal reasoning and its formalization in verification tools. Years later, the success of static analyses and lightweight verification systems convinced me that software verification, when approached appropriately, is not an unreachable utopia but an exciting field capable of delivering many concrete results. The depth of the software verification problem ensures that it will remain an ambitious area in the future, an area that cross-cuts the entire field of computer science, and an area that can both benefit from and have an impact on many other computer science disciplines. I am therefore thrilled to have had an opportunity to experience some of the challenges of this field and expect to continue exploring it in the future.

Bibliography

[1] J-R Abrial. *The B-Book*. Cambridge University Press, 1996. 146

[2] Jean-Raymond Abrial, Matthew K. O. Lee, Dave Neilson, P. N. Scharbach, and Ib Sørensen. The B-method. In *Proceedings of the 4th International Symposium of VDM Europe on Formal Software Development-Volume 2*, pages 398–405. Springer-Verlag, 1991. 39, 146

[3] W. Ackermann. *Solvable Cases of the Decision Problem*. North Holland, 1954. 139

[4] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. The KeY tool. *Software and System Modeling*, 4:32–54, 2005. 39

[5] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Martin Giese, Elmar Habermalz, Reiner Haehnle, Wolfram Menzel, and Peter H. Schmitt. The KeY approach: Integrating object oriented design and formal verification. In *Proceedings, 8th European Workshop on Logics in AI (JELIA), Malaga, Spain*, 2000. 147

[6] Alexander Aiken, Edward L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *Proc. 21st ACM POPL*, pages 163–173, New York, NY, 1994. 9

[7] Glenn Ammons, Rastislav Bodik, and James R. Larus. Mining specifications. In *Proc. 29th ACM POPL*, 2002. 145

[8] C. Scott Ananian, Krste Asanović, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded transactional memory. *IEEE Micro, Special Issue: Top Picks from Computer Architecture Conferences*, 26(1), 2006. 27

[9] C. Scott Ananian and Martin Rinard. Efficient object-based software transactions. In *Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*, 2005. 27

[10] P. Andrews, M. Bishop, S. Issar, D. Nesmith, F. Pfenning, and H. Xi. TPS: A theorem proving system for classical type theory. *Journal of Automated Reasoning*, 16(3):321–353, June 1996. 63

[11] Peter B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*. Springer (Kluwer), 2nd edition, 2002. 43, 45, 46

[12] Konstantine Arkoudas. *Denotational Proof Languages*. PhD thesis, Massachusetts Institute of Technology, 2000. 63

[13] Konstantine Arkoudas. Specification, abduction, and proof. In *Second International Symposium on Automated Technology for Verification and Analysis*, Taiwan, October 2004. 64

[14] Konstantine Arkoudas, Sarfraz Khurshid, Darko Marinov, and Martin Rinard. Integrating model checking and theorem proving for relational reasoning. In *7th International Seminar on Relational Methods in Computer Science (RelMiCS 2003)*, 2003. 64

[15] Konstantine Arkoudas, Karen Zee, Viktor Kuncak, and Martin Rinard. Verifying a file system implementation. In *Sixth International Conference on Formal Engineering Methods (ICFEM'04)*, volume 3308 of *LNCS*, Seattle, Nov 8-12, 2004 2004. 53, 63, 64, 89, 140, 144

[16] Alessandro Armando, Silvio Ranise, and Michaël Rusinowitch. A rewriting approach to satisfiability procedures. *Information and Computation*, 183(2):140–164, 2003. 64, 88

[17] David Aspinall. Proof general. `http://proofgeneral.inf.ed.ac.uk/`. Last visited December 9, 2006. 47

[18] Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation and Applications*. CUP, 2003. 140

[19] Leo Bachmair and Harald Ganzinger. Resolution theorem proving. In *Handbook of Automated Reasoning (Volume 1)*, chapter 2. Elsevier and The MIT Press, 2001. 76

[20] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus*. Springer-Verlag, 1998. 38, 45, 52, 146

[21] Ittai Balaban, Amir Pnueli, and Lenore Zuck. Shape analysis by predicate abstraction. In *VMCAI'05*, 2005. 91

[22] Egon Balas and Manfred W. Padberg. Set partitioning: A survey. *SIAM Review*, 18(4):710–760, 1976. 139

[23] Thomas Ball, Shuvendu Lahiri, and Madanlal Musuvathi. Zap: Automated theorem proving for software analysis. Technical Report MSR-TR-2005-137, Microsoft Research, 2005. 88

[24] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *Proc. ACM PLDI*, 2001. 9, 145

[25] M. Balser, W. Reif, G. Schellhorn, K. Stenzel, and A. Thums. Formal system development with KIV. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering*, number 1783 in LNCS. Springer, 2000. 39

[26] Henk P. Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science, Vol. II.* Oxford University Press, 2001. 45, 73

[27] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO*, 2005. 10, 144, 146

[28] Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004. 39, 40

[29] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *CASSIS: Int. Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart devices*, 2004. 88, 144

[30] Clark Barrett and Sergey Berezin. CVC Lite: A new implementation of the cooperating validity checker. In *Proc. 16th Int. Conf. on Computer Aided Verification (CAV '04)*, volume 3114 of *Lecture Notes in Computer Science*, pages 515–518, 2004. 9, 58, 64, 65, 88, 130, 145

[31] David Basin and Stefan Friedrich. Combining WS1S and HOL. In D.M. Gabbay and M. de Rijke, editors, *Frontiers of Combining Systems 2*, volume 7 of *Studies in Logic and Computation*, pages 39–56. Research Studies Press/Wiley, Baldock, Herts, UK, February 2000. 54, 64

[32] Leonard Berman. The complexity of logical theories. *Theoretical Computer Science*, 11(1):71–77, 1980. 109, 110, 121, 139

[33] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development–Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004. 48, 58, 63

[34] Dimitris Bertsimas and John N. Tsitsiklis. *Introduction to Linear Optimization*. Athena Scientific, Belmont, Massachusetts, 1997. 136, 139

[35] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Mine, D. Monniaux, and X. Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In *Essays Dedicated to Neil D. Jones*, volume 2566 of *LNCS*, 2002. 9, 145

[36] Bernard Boigelot, Sébastien Jodogne, and Pierre Wolper. An effective decision procedure for linear arithmetic over the integers and reals. *ACM Trans. Comput. Logic*, 6(3):614–633, 2005. 117, 120

[37] Egon Börger, Erich Grädel, and Yuri Gurevich. *The Classical Decision Problem*. Springer-Verlag, 1997. 139

[38] Charles Bouillaguet, Viktor Kuncak, Thomas Wies, Karen Zee, and Martin Rinard. On using first-order theorem provers in a data structure verification system. Technical Report MIT-CSAIL-TR-2006-072, MIT, November 2006. `http://hdl.handle.net/1721.1/34874`. 65

[39] Chandrasekhar Boyapati. *SafeJava: A Unified Type System for Safe Programming*. PhD thesis, MIT, 2004. 9, 32, 39, 144

[40] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: Automated testing based on Java predicates. In *Proc. International Symposium on Software Testing and Analysis*, pages 123–133, July 2002. 9, 144

[41] R. S. Boyer and J S. Moore. Integrating decision procedures into heuristic theorem provers: A case study of linear arithmetic. In *Machine Intelligence*, volume 11, pages 83–124. Oxford University Press, 1988. 64

[42] M. Bozga and R. Iosif. On decidability within the arithmetic of addition and divisibility. In *FOSSACS'05*, 2005. 139

[43] V. Bruyére, G. Hansel, C. Michaux, and R. Villemaire. Logic and p-recognizable sets of integers. *Bull. Belg. Math. Soc. Simon Stevin*, 1:191–238, 1994. 129, 139

[44] Tevfik Bultan, Richard Gerber, and William Pugh. Model-checking concurrent systems with unbounded integer variables: symbolic representations, approximations, and experimental results. *ACM Trans. Program. Lang. Syst.*, 21(4):747–789, 1999. 116, 143

[45] Alan Bundy, David Basin, Dieter Hutter, and Andrew Ireland. *Rippling: Meta-Level Guidance for Mathematical Reasoning*. Cambridge University Press, 2005. 145

[46] Lilian Burdy, Yoonsik Cheon, David Cok, Michael D. Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. Technical Report NII-R0309, Computing Science Institute, Univ. of Nijmegen, March 2003. 147

[47] Michael Butler, Jim Grundy, Thomas Langbacka, Rimvydas Ruksenas, and Joakim von Wright. The refinement calculator: Proof support for program refinement. In *Proc. Formal Methods Pacific '97*, 1997. 146

[48] Domenico Cantone, Eugenio Omodeo, and Alberto Policriti. *Set Theory for Computing*. Springer, 2001. 139

[49] D. Carrington, I. Hayes, R. Nickson, G. Watson, and J. Welsh. A review of existing refinement tools. Technical report, University of Queensland, Australia, 1994. 146

[50] Amine Chaieb and Tobias Nipkow. Generic proof synthesis for Presburger arithmetic. Technical report, Technische Universität München, October 2003. 139

[51] Patrice Chalin, Clément Hurlin, and Joe Kiniry. Integrating static checking and interactive verification: Supporting multiple theories and provers in verification. In *Proceedings of Verified Software: Tools, Technologies, and Experiences (VSTTE)*, 2005. 39

[52] Ashok K. Chandra, Dexter C. Kozen, and Larry J. Stockmeyer. Alternation. *J. ACM*, 28(1):114–133, 1981. 121

[53] Yoonsik Cheon. *A Runtime Assertion Checker for the Java Modelling Language*. PhD thesis, Iowa State University, April 2003. 39

[54] David R. Cheriton and Michael E. Wolf. Extensions for multi-module records in conventional programming languages. In *ACM PLDI*, pages 296–306. ACM Press, 1987. 32

[55] Wei-Ngan Chin, Siau-Cheng Khoo, and Dana N. Xu. Extending sized types with with collection analysis. In *ACM PEPM'03*, 2003. 140

[56] Koen Claessen and Niklas Sörensson. New techniques that improve MACE-style model finding. In *Model Computation*, 2003. 53, 55

[57] Dave Clarke and Sophia Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *Proceedings of the 17th ACM conference on Object-oriented programming, systems, languages, and applications*, pages 292–310. ACM Press, 2002. 144

[58] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *Proc. 13th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 1998. 144

[59] Paul J. Cohen. *Set Theory and the Continuum Hypothesis*. New York: Benjamin, 1966. 53

[60] David R. Cok. Reasoning with specifications containing method calls and model fields. *Journal of Object Technology*, 4(8):77–103, September–October 2005. 88

[61] David R. Cok and Joseph R. Kiniry. Esc/java2: Uniting ESC/Java and JML. In *CASSIS: Construction and Analysis of Safe, Secure and Interoperable Smart devices*, 2004. 10, 88

[62] Robert L. Constable, Stuart F. Allen, H. M. Bromley, Walter Rance Cleaveland, J. F. Cremer, Robert W. Harper, Douglas J. Howe, Todd B. Knoblock, Nax P. Mendler, Prakash Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing Mathematics with the Nuprl Development System*. Prentice-Hall, 1986. 63

[63] W. J. Cook, J. Fonlupt, and A. Schrijver. An integer analogue of Carathéodory's theorem. *Journal of Combinatorial Theory, Series B*, 40(63–70), 1986. 136

[64] D. C. Cooper. Theorem proving in arithmetic without multiplication. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 7, pages 91–100. Edinburgh University Press, 1972. 139

[65] Thierry Coquand and Gérard P. Huet. The calculus of constructions. *Inf. Comput.*, 76(2/3):95–120, 1988. 63

[66] James Corbett, Matthew Dwyer, John Hatcliff, Corina Pasareanu, Robby, Shawn Laubach, and Hongjun Zheng. Bandera: Extracting finite-state models from Java source code. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE)*, June 2000. 39

[67] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Cliff Stein. *Introduction to Algorithms (Second Edition)*. MIT Press and McGraw-Hill, 2001. 49

[68] J.-F. Couchot, F. Dadeau, D. Déharbe, A. Giorgetti, and S. Ranise. Proving and debugging set-based specifications. In *Proc. of the 6th Workshop on Formal Methods*, 2003. 89

[69] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. 4th POPL*, 1977. 54, 146

[70] Coverity, Incorporated. http://www.coverity.com/, 2007. Last visited January 14, 2007. 143

[71] H.B. Curry, J. R. Hindley, and J. P. Seldin. *Combinatory Logic II*. North-Holland, 1972. 13

[72] Dennis Dams and Kedar S. Namjoshi. Shape analysis through predicate abstraction and model checking. In *VMCAI'03*, volume 2575 of *LNCS*, pages 310–323, 2003. 91

[73] Bent Dandanell. Rigorous development using RAISE. In *Proceedings of the conference on Software for citical systems*, pages 29–43. ACM Press, 1991. 39

[74] Manuvir Das. Formal specifications on industrial-strength code–From myth to reality. In *CAV*, page 1, 2006. 10, 143

[75] Manuvir Das, Sorin Lerner, and Mark Seigle. ESP: Path-sensitive program verification in polynomial time. In *Proc. ACM PLDI*, 2002. 9

[76] Rowan Davies. *Practical Refinement-Type Checking*. PhD thesis, CMU, 2005. 89

[77] Willem-Paul de Roever and Kai Engelhardt. *Data Refinement: Model-oriented proof methods and their comparison*. Cambridge University Press, 1998. 11, 17, 31, 39, 69

[78] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *Proc. ACM PLDI*, 2001. 9

[79] Brian Demsky, Cristian Cadar, Daniel Roy, and Martin C. Rinard. Efficient specification-assisted error localization. In *Second International Workshop on Dynamic Analysis*, 2004. 39

[80] Greg Dennis, Felix Chang, and Daniel Jackson. Modular verification of code with SAT. In *ISSTA*, 2006. 9, 10, 39, 89

[81] L. A. Dennis, G. Collins, M. Norrish, R. Boulton, K. Slind, G. Robinson, M. Gordon, and T. Melham. The PROSPER toolkit. In S. Graf and M. Schwartbach, editors, *Tools and Algorithms for Constructing Systems (TACAS 2000)*, number 1785 in Lecture Notes in Computer Science, pages 78–92. Springer-Verlag, 2000. 64

[82] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Laboratories Palo Alto, 2003. 64, 88, 138

[83] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Technical Report 159, COMPAQ Systems Research Center, 1998. 39

[84] Robert K. Dewar. Programming by refinement, as exemplified by the SETL representation sublanguage. *ACM TOPLAS*, July 1979. 113

[85] Jonathan Edwards, Daniel Jackson, and Emina Torlak. A type system for object models. In *Foundations of Software Engineering*, 2004. 63

[86] Friedrich Eisenbrand and Gennady Shmonina. Carathéodory bounds for integer cones. *Operations Research Letters*, 34(5):564–568, September 2006. http://dx.doi.org/10.1016/j.orl.2005.09.008. 130, 132, 134, 135, 139

[87] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proc. 4th USENIX OSDI*, 2000. 146

[88] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proc. 18th ACM Symposium on Operating Systems Principles*, 2001. 145

[89] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 2006. 145

[90] S. Feferman and R. L. Vaught. The first order properties of products of algebraic systems. *Fundamenta Mathematicae*, 47:57–103, 1959. 64, 109, 112, 128, 139, 140, 143

[91] Jeanne Ferrante and Charles W. Rackoff. *The Computational Complexity of Logical Theories*, volume 718 of *Lecture Notes in Mathematics*. Springer-Verlag, 1979. 139

[92] Jean-Christophe Filliatre. Verification of non-functional programs using interpretations in type theory. *Journal of Functional Programming*, 13(4):709–745, 2003. 39

[93] Cormac Flanagan, Rajeev Joshi, Xinming Ou, and James B. Saxe. Theorem proving using lazy proof explication. In *CAV*, pages 355–367, 2003. 64

[94] Cormac Flanagan, Rajeev Joshi, and James B. Saxe. An explicating theorem prover for quantified formulas. Technical Report HPL-2004-199, HP Laboratories Palo Alto, 2004. 9, 51

[95] Cormac Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for esc/java. In *FME '01: Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity*, pages 500–517, London, UK, 2001. Springer-Verlag. 39

[96] Cormac Flanagan, K. Rustan M. Leino, Mark Lilibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended Static Checking for Java. In *ACM Conf. Programming Language Design and Implementation (PLDI)*, 2002. 10, 11, 15, 21, 29, 39

[97] Cormac Flanagan and Shaz Qadeer. Predicate abstraction for software verification. In *Proc. 29th ACM POPL*, 2002. 39

[98] Cormac Flanagan and James B. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *Proc. 28th ACM POPL*, 2001. 37

[99] Darren Foulger and Steve King. Using the SPARK toolset for showing the absence of run-time errors in safety-critical software. In *Ada-Europe 2001*, pages 229–240, 2001. 39

[100] The Eclipse Foundation. `http://www.eclipse.org/`, 2007. Last visited January 14, 2007. 146

[101] Pascal Fradet and Daniel Le Métayer. Shape types. In *Proc. 24th ACM POPL*, 1997. 10, 91

[102] Andreas Franke, Stephan M. Hess, Christoph G. Jung, Michael Kohlhase, and Volker Sorge. Agent-oriented integration of distributed mathematical services. *J. Universal Computer Science*, 5(3):156–187, 1999. 64

[103] H. Ganzinger and K. Korovin. Theory Instantiation. In *Proceedings of the 13 Conference on Logic for Programming Artificial Intelligence Reasoning (LPAR'06)*, Lecture Notes in Computer Science. Springer, 2006. 145

[104] Silvio Ghilardi. Model theoretic methods in combined constraint satisfiability. *Journal of Automated Reasoning*, 33(3-4):221–249, 2005. 64

[105] Silvio Ghilardi, Enrica Nicolini, and Daniele Zucchelli. A comprehensive framework for combined decision procedures. In *FroCos*, pages 1–30, 2005. 64

[106] Rakesh Ghiya and Laurie Hendren. Is it a tree, a DAG, or a cyclic graph? In *Proc. 23rd ACM POPL*, 1996. 10, 91

[107] Patrice Godefroid. Model checking for programming languages using verisoft. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 174–186, New York, NY, USA, 1997. ACM Press. 39

[108] Kurt Gödel. *On Formally Undecidable Propositions of Principia Mathematica and Related Systems (reprint)*. Dover, 1992. 53

[109] Donald I. Good, Robert L. Akers, and Lawrence M. Smith. Report on Gypsy 2.05. Technical report, University of Texas at Austin, February 1986. 39

[110] M. J. C. Gordon and T. F. Melham. *Introduction to HOL, a theorem proving environment for higher-order logic*. Cambridge University Press, Cambridge, England, 1993. 45, 63, 140

[111] Michael Gordon. Notes on PVS from a HOL perspective. `http://www.cl.cam.ac.uk/users/mjcg/PVS.html`, 1995. Last visited January 17, 2007. 63

[112] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Sun Microsystems, Inc., 2001. 11, 27

[113] Erich Grädel. Decidable fragments of first-order and fixed-point logic. From prefix-vocabulary classes to guarded logics. In *Proceedings of Kalmár Workshop on Logic and Computer Science, Szeged*, 2003. 103

[114] Susanne Graf and Hassen Saidi. Construction of abstract state graphs with PVS. In *Proc. 9th CAV*, pages 72–83, 1997. 146

[115] Dan Grossman. Existential types for imperative languages. In *Proc. 11th ESOP*, 2002. 9

[116] John Guttag and James Horning. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, 1993. 39, 147

[117] John V. Guttag, Ellis Horowitz, and David R. Musser. Abstract data types and software validation. *Communications of the ACM*, 21(12):1048–1064, 1978. 88

[118] K. Havelund and T. Pressburger. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer*, 2000. 39

[119] J.G. Henriksen, J. Jensen, M. Jørgensen, N. Klarlund, B. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In *TACAS '95, LNCS 1019*, 1995. 120, 129

[120] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. Abstractions from proofs. In *31st POPL*, 2004. 145

[121] Benjamin Hindman and Dan Grossman. Atomicity via source-to-source translation. In *ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, 2006. 27

[122] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969. 34

[123] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281, 1971. 17

[124] Marieke Huisman. *Java program verification in higher order logic with PVS and Isabelle*. PhD thesis, University of Nijmegen, 2001. 88

[125] Joe Hurd. Integrating Gandalf and HOL. In *Proc. 12th International Conference on Theorem Proving in Higher Order Logics*, volume 1690 of *Lecture Notes in Computer Science*, pages 311–321. Springer, September 1999. 64

[126] Joe Hurd. An LCF-style interface between HOL and first-order logic. In *CADE-18*, 2002. 13, 64, 89

[127] Neil Immerman. *Descriptive Complexity*. Springer-Verlag, 1998. 47, 92

[128] Neil Immerman, Alexander Moshe Rabinovich, Thomas W. Reps, Shmuel Sagiv, and Greta Yorsh. The boundary between decidability and undecidability for transitive-closure logics. In *Computer Science Logic (CSL)*, pages 160–174, 2004. 106

[129] Neil Immerman, Alexander Moshe Rabinovich, Thomas W. Reps, Shmuel Sagiv, and Greta Yorsh. Verification via structure simulation. In *CAV*, pages 281–294, 2004. 92, 98, 103, 107, 108

[130] Samin Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. In *Proc. 28th ACM POPL*, 2001. 89, 143, 144

[131] Daniel Jackson. *Software Abstractions: Logic, Language, & Analysis*. MIT Press, 2006. 39, 53, 55, 63, 138

[132] Daniel Jackson, Ilya Shlyakhter, and Manu Sridharan. A micromodularity mechanism. In *Proc. ACM SIGSOFT Conf. Foundations of Software Engineering / European Software Engineering Conference (FSE/ESEC '01)*, 2001. 9, 53

[133] Bart P. F. Jacobs and Erik Poll. Java program verification at Nijmegen: Developments and perspective. Technical Report NIII-R0318, Nijmegen Institute of Computing and Information Sciences, September 2003. 39

[134] Jacob L. Jensen, Michael E. Jørgensen, Nils Klarlund, and Michael I. Schwartzbach. Automatic verification of pointer programs using monadic second order logic. In *Proc. ACM PLDI*, Las Vegas, NV, 1997. 91

[135] Cliff B. Jones. *Systematic Software Development using VDM*. Prentice Hall International (UK) Ltd., 1986. `http://www.vdmbook.com/jones90.pdf`. 39

[136] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. *Information and Computing*, 98(2):142–170, 1992. 9

[137] Deepak Kapur. Automatically generating loop invariants using quantifier elimination. In *IMACS Intl. Conf. on Applications of Computer Algebra*, 2004. 116

[138] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore, editors. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Publishers, 2000. 53, 63

[139] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore, editors. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000. 63, 145, 147

[140] Sarfraz Khurshid and Darko Marinov. TestEra: Specification-based testing of java programs using SAT. *Autom. Softw. Eng.*, 11(4):403–434, 2004. 89, 144

[141] James Cornelius King. *A Program Verifier*. PhD thesis, CMU, 1970. 39

[142] H. Kirchner, S. Ranise, C. Ringeissen, and D. K. Tran. Automatic combinability of rewriting based satisfiability procedures. In *13th International Conference on Logic Programming and Artificial Intelligence and Reasoning (LPAR)*, 2006. 64

[143] Nils Klarlund, Anders Møller, and Michael I. Schwartzbach. MONA implementation secrets. In *Proc. 5th International Conference on Implementation and Application of Automata*. LNCS, 2000. 9, 23, 58, 92, 94, 117

[144] Nils Klarlund and Michael I. Schwartzbach. Graph types. In *Proc. 20th ACM POPL*, Charleston, SC, 1993. 91

[145] Dexter Kozen. Complexity of boolean algebras. *Theoretical Computer Science*, 10:221–247, 1980. 121, 123, 139

[146] Dexter Kozen. *Theory of Computation*. Springer, 2006. 121

[147] Ted Kremenek, Paul Twohey, Godmar Back, Andrew Y. Ng, and Dawson Engler. From uncertainty to belief: Inferring the specification within. In *Seventh USENIX Symposium on Operating Systems Design and Implementation (OSDI 2006)*, 2006. 145

[148] Viktor Kuncak. Binary search trees. The Archive of Formal Proofs, `http://afp.sourceforge.net/`, April 2004. 88

[149] Viktor Kuncak and Daniel Jackson. On relational analysis of algebraic datatypes. Technical Report 985, MIT, April 2005. 55

[150] Viktor Kuncak and Daniel Jackson. Relational analysis of algebraic datatypes. In *Joint 10th European Software Engineering Conference (ESEC) and 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, 2005. 53, 144

[151] Viktor Kuncak, Patrick Lam, and Martin Rinard. Role analysis. In *Annual ACM Symp. on Principles of Programming Languages (POPL)*, 2002. 10, 66, 107, 144

[152] Viktor Kuncak, Patrick Lam, Karen Zee, and Martin Rinard. Modular pluggable analyses for data structure consistency. *IEEE Transactions on Software Engineering*, 32(12), December 2006. 10, 49, 59, 91, 94, 107, 112, 113

[153] Viktor Kuncak, Hai Huu Nguyen, and Martin Rinard. An algorithm for deciding BAPA: Boolean Algebra with Presburger Arithmetic. In *20th International Conference on Automated Deduction, CADE-20*, Tallinn, Estonia, July 2005. 109, 110, 130, 138

[154] Viktor Kuncak, Hai Huu Nguyen, and Martin Rinard. Deciding Boolean Algebra with Presburger Arithmetic. *J. of Automated Reasoning*, 2006. `http://dx.doi.org/10.1007/s10817-006-9042-1`. 109, 130, 139

[155] Viktor Kuncak and Martin Rinard. On the theory of structural subtyping. Technical Report 879, Laboratory for Computer Science, Massachusetts Institute of Technology, 2003. 140

[156] Viktor Kuncak and Martin Rinard. Structural subtyping of non-recursive types is decidable. In *Eighteenth Annual IEEE Symposium on Logic in Computer Science*, 2003. 64, 140

[157] Viktor Kuncak and Martin Rinard. Boolean algebra of shape analysis constraints. In *Proc. 5th International Conference on Verification, Model Checking and Abstract Interpretation*, 2004. 108

[158] Viktor Kuncak and Martin Rinard. The first-order theory of sets with cardinality constraints is decidable. Technical Report 958, MIT CSAIL, July 2004. 109, 110, 124, 130, 138

[159] Viktor Kuncak and Martin Rinard. Generalized records and spatial conjunction in role logic. In *11th Annual International Static Analysis Symposium (SAS'04)*, Verona, Italy, August 26–28 2004. 144

[160] Viktor Kuncak and Martin Rinard. Decision procedures for set-valued fields. In *1st International Workshop on Abstract Interpretation of Object-Oriented Languages (AIOOL 2005)*, 2005. 65, 89, 107, 139

[161] Shuvendu K. Lahiri and Shaz Qadeer. Verifying properties of well-founded linked lists. In *POPL'06*, 2006. 107

[162] Shuvendu K. Lahiri and Sanjit A. Seshia. The UCLID decision procedure. In *CAV'04*, 2004. 9, 130, 145

[163] Patrick Lam, Viktor Kuncak, and Martin Rinard. Cross-cutting techniques in program specification and analysis. In *4th International Conference on Aspect-Oriented Software Development (AOSD'05)*, 2005. 32, 40

[164] Patrick Lam, Viktor Kuncak, and Martin Rinard. Generalized typestate checking for data structure consistency. In *6th Int. Conf. Verification, Model Checking and Abstract Interpretation*, 2005. 10, 107, 109, 116

[165] Patrick Lam, Viktor Kuncak, and Martin Rinard. Hob: A tool for verifying data structure consistency. In *14th International Conference on Compiler Construction (tool demo)*, April 2005. 91, 107

[166] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML. Technical Report 96-06p, Iowa State University, 2001. 88

[167] Oukseh Lee, Hongseok Yang, and Kwangkeun Yi. Automatic verification of pointer programs using grammar-based shape analysis. In *ESOP*, 2005. 10, 89, 91

[168] K. Rustan M. Leino and F. Logozzo. Loop invariants on demand. In *Proceedings of the the 3rd Asian Symposium on Programming Languages and Systems (APLAS'05)*, volume 3780 of *LNCS*, 2005. 145

[169] K. Rustan M. Leino and Peter Müller. A verification methodology for model fields. In *ESOP'06*, 2006. 88

[170] Xavier Leroy. *The Objective Caml system, release 3.08*, July 2004. 40

[171] Xavier Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *POPL*, 2006. 53

[172] T. Lev-Ami, N. Immerman, T. Reps, M. Sagiv, S. Srivastava, and G. Yorsh. Simulating reachability using first-order logic with applications to verification of linked data structures. In *CADE-20*, 2005. 107

[173] Tal Lev-Ami. TVLA: A framework for Kleene based logic static analyses. Master's thesis, Tel-Aviv University, Israel, 2000. 89

[174] Tal Lev-Ami, Thomas Reps, Mooly Sagiv, and Reinhard Wilhelm. Putting static analysis to work for verification: A case study. In *Int. Symp. Software Testing and Analysis*, 2000. 89, 107

[175] Barbara Liskov and John Guttag. *Program Development in Java*. Addison-Wesley, 2001. 11

[176] L. Loewenheim. Über Mögligkeiten im Relativkalkül. *Math. Annalen*, 76:228–251, 1915. 112, 139

[177] Maria Manzano. *Extensions of First-Order Logic*. Cambridge University Press, 1996. 74, 87

[178] C. Marché, C. Paulin-Mohring, and X. Urbain. The Krakatoa tool for certification of JAVA/JAVACARD programs annotated in JML. *Journal of Logic and Algebraic Programming*, 2003. 10, 39, 89

[179] Darko Marinov. *Automatic Testing of Software with Structurally Complex Inputs*. PhD thesis, MIT, 2005. 39, 89, 144

[180] Bruno Marnette, Viktor Kuncak, and Martin Rinard. On algorithms and complexity for sets with cardinality constraints. Technical report, MIT CSAIL, August 2005. 130

[181] Kim Marriott and Martin Odersky. Negative boolean constraints. Technical Report 94/203, Monash University, August 1994. 139

[182] Ursula Martin and Tobias Nipkow. Boolean unification: The story so far. *Journal of Symbolic Computation*, 7(3):275–293, 1989. 139

[183] John Matthews, J. Strother Moore, Sandip Ray, and Daron Vroon. Verification condition generation via theorem proving. In *LPAR*, pages 362–376, 2006. 145

[184] Sean McLaughlin, Clark Barrett, and Yeting Ge. Cooperating theorem provers: A case study combining HOL-Light and CVC Lite. In *Proc. 3^{rd} Workshop on Pragmatics of Decision Procedures in Automated Reasoning (PDPAR '05)*, volume 144(2) of *Electronic Notes in Theoretical Computer Science*, pages 43–51. Elsevier, January 2006. 64

[185] Scott McPeak and George C. Necula. Data structure specifications via local equality axioms. In *CAV*, pages 476–490, 2005. 66, 91, 107

[186] Jia Meng and L. C. Paulson. Lightweight relevance filtering for machine-generated resolution problems. In *ESCoR: Empirically Successful Computerized Reasoning*, 2006. 64, 66, 78, 89, 145

[187] Jia Meng and L. C. Paulson. Translating higher-order problems to first-order clauses. In *ESCoR: Empir. Successful Comp. Reasoning*, pages 70–80, 2006. 13, 64, 89

[188] Jia Meng and Lawrence C. Paulson. Experiments on supporting interactive proof using resolution. In *IJCAR*, 2004. 13, 54, 64, 89

[189] Robin Milner. An algebraic definition of simulation between programs. In *Proceedings of the 2nd international joint Artificial intelligence Conference*, 1971. 17

[190] Anders Møller and Michael I. Schwartzbach. The Pointer Assertion Logic Engine. In *Programming Language Design and Implementation*, 2001. 10, 91, 92, 94, 103, 108, 140, 142

[191] Steven S. Muchnick and Neil D. Jones, editors. *Program Flow Analysis: Theory and Applications*. Prentice-Hall, Inc., 1981. 91

[192] Madanlal Musuvathi, David Y.W. Park, Andy Chou, Dawson R. Engler, and David L. Dill. CMC: A pragmatic approach to model checking real code. In *OSDI'02*, 2002. 39

[193] David A. Naumann and Michael Barnett. Towards imperative modules: Reasoning about invariants and sharing of mutable state. In *LICS*, pages 313–323, 2004. 40

[194] Toh Ne Win, Michael D. Ernst, Stephen J. Garland, Dilsun Kırlı, and Nancy Lynch. Using simulated execution in verifying distributed algorithms. *Software Tools for Technology Transfer*, 6(1):67–76, July 2004. 145

[195] Greg Nelson. Techniques for program verification. Technical report, XEROX Palo Alto Research Center, 1981. 39, 61

[196] Greg Nelson. Verifying reachability invariants of linked structures. In *POPL*, 1983. 107

[197] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM TOPLAS*, 1(2):245–257, 1979. 61, 64, 140

[198] Huu Hai Nguyen, Cristina David, Shengchao Qin, and Wei-Ngan Chin. Automated verification of shape, size and bag properties via separation logic. In *VMCAI*, 2007. 89, 143

[199] Robert Nieuwenhuis and Albert Rubio. Paramodulation-based theorem proving. In *Handbook of Automated Reasoning (Volume 1)*, chapter 7. Elsevier and The MIT Press, 2001. 76, 87

[200] Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL Tutorial Draft*, March 8 2002. 29, 48, 58, 63

[201] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer-Verlag, 2002. 11, 15, 19, 43, 46, 140

[202] Hans Jürgen Ohlbach and Jana Koehler. How to extend a formal system with a boolean algebra component. In W. Bibel P.H. Schmidt, editor, *Automated Deduction. A Basis for Applications*, volume III, pages 57–75. Kluwer Academic Publishers, 1998. 125, 130, 139

[203] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998. 67

[204] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th CADE*, volume 607 of *LNAI*, pages 748–752, jun 1992. 63, 64, 140

[205] Leszek Pacholski, Wieslaw Szwast, and Lidia Tendera. Complexity results for first-order two-variable logic with counting. *SIAM J. on Computing*, 29(4):1083–1117, 2000. 47

[206] Christos H. Papadimitriou. On the complexity of integer programming. *J. ACM*, 28(4):765–768, 1981. 130

[207] Lawrence C. Paulson. Isabelle: the next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–386. Academic Press, 1990. 63

[208] Benjamin Pierce. *Types and Programming Languages*. The MIT Press, Cambridge, Mass., 2001. 45

[209] Amir Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57, 1977. 147

[210] Andreas Podelski and Andrey Rybalchenko. Transition predicate abstraction and fair termination. In *ACM POPL*, 2005. 115

[211] Andreas Podelski and Thomas Wies. Boolean heaps. In *Proc. Int. Static Analysis Symposium*, 2005. 33, 91, 92, 96, 108

[212] M. Presburger. Über die vollständigkeit eines gewissen systems der aritmethik ganzer zahlen, in welchem die addition als einzige operation hervortritt. In *Comptes Rendus du premier Congrès des Mathématiciens des Pays slaves, Warsawa*, pages 92–101, 1929. 120, 139

[213] Virgile Prevosto and Uwe Waldmann. SPASS+T. In *ESCoR: Empirically Successful Computerized Reasoning*, volume 192, 2006. 88, 145

[214] William Pugh. Skip lists: A probabilistic alternative to balanced trees. In *Communications of the ACM 33(6):668–676*, 1990. 92, 94

[215] William Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. In *Supercomputing '91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 4–13. ACM Press, 1991. 9, 120, 139

[216] Silvio Ranise and Cesare Tinelli. The SMT-LIB format: An initial proposal. In *Proceedings of the Workshop on Pragmatics of Decision Procedures in Automated Reasoning (PDPAR)*, 2003. 58, 65

[217] C. R. Reddy and D. W. Loveland. Presburger arithmetic with bounded quantifier alternation. In *ACM STOC*, pages 320–325. ACM Press, 1978. 121, 122, 139

[218] Jan Reineke. Shape analysis of sets. Master's thesis, Universität des Saarlandes, Germany, June 2005. 89

[219] Thomas Reps, Mooly Sagiv, and Alexey Loginov. Finite differencing of logical formulas for static analysis. In *Proc. 12th ESOP*, 2003. 107, 146

[220] Thomas Reps, Mooly Sagiv, and Greta Yorsh. Symbolic implementation of the best transformer. In *Proc. 5th International Conference on Verification, Model Checking and Abstract Interpretation*, 2004. 146

[221] Peter Revesz. Quantifier-elimination for the first-order theory of boolean algebras with linear cardinality constraints. In *Proc. Advances in Databases and Information Systems (ADBIS'04)*, 2004. 109, 116, 130, 139

[222] Piotr Rudnicki and Andrzej Trybulec. On equivalents of well-foundedness. *J. Autom. Reasoning*, 23(3-4):197–234, 1999. 63

[223] Harald Ruess and Natarajan Shankar. Deconstructing Shostak. In *Proc. 16th IEEE LICS*, 2001. 140

[224] Radu Rugina. Quantitative shape analysis. In *Static Analysis Symposium (SAS'04)*, 2004. 10, 89, 140

[225] Radu Rugina and Martin Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *PLDI'00*, pages 182–195, 2000. 9

[226] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM TOPLAS*, 24(3):217–298, 2002. 9, 10, 89, 91, 107, 140, 146

[227] Alexander Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, 1998. 136

[228] Stephan Schulz. E – A Brainiac Theorem Prover. *Journal of AI Communications*, 15(2/3):111–126, 2002. 13, 58, 71, 78

[229] András Sebö. Hilbert bases, Caratheodory's theorem and combinatorial optimization. In R. Kannan and W. Pulleyblank, editors, *Integer Programming and Combinatorial Optimization I*. University of Waterloo Press, 1990. 138, 139

[230] Natarajan Shankar. Using decision procedures with a higher-order logic. In *Proc. 2001 International Conference on Theorem Proving in Higher Order Logics*, 2001. 64

[231] Joerg Siekmann, Christoph Benzmueller, Vladimir Brezhnev, Lassaad Cheikhrouhou, Armin Fiedler, Andreas Franke, Helmut Horacek, Michael Kohlhase, Andreas Meier, Erica Melis, Markus Moschner, Immanuel Normann, Martin Pollet, Volker Sorge, Carsten Ullrich, Claus-Peter Wirth, and Juergen Zimmer. Proof development with OMEGA. *LNAI*, 2392:144, 2002. 63

[232] Thoralf Skolem. Untersuchungen über die Axiome des Klassenkalküls and über "Produktations- und Summationsprobleme", welche gewisse Klassen von Aussagen betreffen. Skrifter utgit av Vidnskapsselskapet i Kristiania, I. klasse, no. 3, Oslo, 1919. 112

[233] Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE TSE*, January 1986. 10

[234] G. Sutcliffe and C. B. Suttner. The TPTP problem library: CNF release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998. 78

[235] J. W. Thatcher and J. B. Wright. Generalized finite automata theory with an application to a decision problem of second-order logic. *Mathematical Systems Theory*, 2(1):57–81, August 1968. 91, 129

[236] Wolfgang Thomas. Languages, automata, and logic. In *Handbook of Formal Languages Vol.3: Beyond Words*. Springer-Verlag, 1997. 129

[237] Cesare Tinelli. Cooperation of background reasoners in theory reasoning by residue sharing. *Journal of Automated Reasoning*, 30(1):1–31, January 2003. 62, 64

[238] Cesare Tinelli and Mehdi T. Harandi. A new correctness proof of the Nelson–Oppen combination procedure. In F. Baader and K. U. Schulz, editors, *Frontiers of Combining Systems*, Applied Logic, pages 103–120, March 1996. 64

[239] Cesare Tinelli and Calogero Zarba. Combining nonstably infinite theories. *Journal of Automated Reasoning*, 34(3), 2005. 64, 140

[240] Frank Tip, Adam Kiezun, and Dirk Bäumer. Refactoring for generalization using type constraints. In *OOPSLA '03*, pages 13–26, 2003. 146

[241] Ashish Tiwari. *Decision procedures in automated deduction*. PhD thesis, Department of Computer Science, State University of New York at Stony Brook, 2000. 64, 140

[242] Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, 2007. 39, 63

[243] John Venn. On the diagrammatic and mechanical representation of propositions and reasonings. *Dublin Philosophical Magazine and Journal of Science*, 9(59):1–18, 1880. 117

[244] Andrei Voronkov. The anatomy of Vampire (implementing bottom-up procedures with code trees). *Journal of Automated Reasoning*, 15(2):237–265, 1995. 9, 58, 140

[245] Igor Walukiewicz. Monadic second-order logic on tree-like structures. *Theoretical Computer Science*, 275(1–2):311–346, March 2002. 64

[246] Tjark Weber. Bounded model generation for Isabelle/HOL. volume 125 of *Electronic Notes in Theoretical Computer Science*, pages 103–116. Elsevier, July 2005. 144

[247] Ben Wegbreit and Jay M. Spitzen. Proving properties of complex data structures. *Journal of the ACM (JACM)*, 23(2):389–396, 1976. 19

[248] C. Weidenbach. Combining superposition, sorts and splitting. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume II, chapter 27, pages 1965–2013. Elsevier Science, 2001. 9, 13, 26, 58, 71, 78, 87, 140

[249] Markus Wenzel. *Isabelle/Isar — a versatile environment for human-readable formal proof documents*. PhD thesis, Technische Universitaet Muenchen, 2002. 144

[250] H. Whitney. On the abstract properties of linear independence. *American Journal of Mathematics*, 57:509–533, 1935. 134, 138

[251] Thomas Wies. Symbolic shape analysis. Master's thesis, Universität des Saarlandes, Saarbrücken, Germany, September 2004. 63, 91, 92, 94, 96, 108

[252] Thomas Wies, Viktor Kuncak, Patrick Lam, Andreas Podelski, and Martin Rinard. On field constraint analysis. Technical Report MIT-CSAIL-TR-2005-072, MIT-LCS-TR-1010, MIT CSAIL, November 2005. 91, 94

[253] Thomas Wies, Viktor Kuncak, Patrick Lam, Andreas Podelski, and Martin Rinard. Field constraint analysis. In *Proc. Int. Conf. Verification, Model Checking, and Abstract Interpratation*, 2006. 10, 33, 91, 94, 143

[254] Thomas Wies, Viktor Kuncak, Karen Zee, Andreas Podelski, and Martin Rinard. On verifying complex properties using symbolic shape analysis. Technical Report MPI-I-2006-2-1, Max-Planck Institute for Computer Science, 2006. `http://arxiv.org/abs/cs.PL/0609104`. 10, 20, 33, 63, 91, 94, 96, 145, 146

[255] Jim Woodcock and Jim Davies. *Using Z*. Prentice-Hall, Inc., 1996. 39

[256] XEmacs: The next generation of Emacs. http://www.xemacs.org/. Last visited December 9, 2006. 47

[257] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. *SIGPLAN Notices*, 33(5):249–257, 1998. 9, 89

[258] Greta Yorsh, Thomas Ball, and Mooly Sagiv. Testing, abstraction, theorem proving: better together! In *ISSTA*, pages 145–156, 2006. 145

[259] Greta Yorsh, Thomas Reps, and Mooly Sagiv. Symbolically computing most-precise abstract operations for shape analysis. In *10th TACAS*, 2004. 108, 140

[260] Greta Yorsh, Thomas Reps, Mooly Sagiv, and Reinhard Wilhelm. Logical characterizations of heap abstractions. *TOCL*, 8(1), 2007. 146

[261] Greta Yorsh, Alexey Skidanov, Thomas Reps, and Mooly Sagiv. Automatic assume/guarantee reasoning for heap-manupilating programs. In *1st AIOOL Workshop*, 2005. 108

[262] γreta Yorsh, Thomas Reps, Mooly Sagiv, and Reinhard Wilhelm. Logical characterizations of heap abstractions. *ACM Transactions on Computational Logic (TOCL)*, 8(1), January 2007. 108

[263] Calogero G. Zarba. *The Combination Problem in Automated Reasoning*. PhD thesis, Stanford University, 2004. 64, 140

[264] Calogero G. Zarba. Combining sets with elements. In Nachum Dershowitz, editor, *Verification: Theory and Practice*, volume 2772 of *Lecture Notes in Computer Science*, pages 762–782. Springer, 2004. 140

[265] Calogero G. Zarba. A quantifier elimination algorithm for a fragment of set theory involving the cardinality operator. In *18th International Workshop on Unification*, 2004. 109, 139, 143

[266] Calogero G. Zarba. Combining sets with cardinals. *J. of Automated Reasoning*, 34(1), 2005. 63, 110, 128, 130, 140

[267] Karen Zee, Patrick Lam, Viktor Kuncak, and Martin Rinard. Combining theorem proving with static analysis for data structure consistency. In *International Workshop on Software Verification and Validation (SVV 2004)*, Seattle, November 2004. 38, 49, 51, 88

[268] Lintao Zhang, Conor F. Madigan, Matthew W. Moskewicz, and Sharad Malik. Efficient conflict driven learning in boolean satisfiability solver. In *ICCAD*, pages 279–285, 2001. 9

[269] Dengping Zhu and Hongwei Xi. Safe programming with pointers through stateful views. In *Proc. 7th Int. Symp. Practical Aspects of Declarative Languages*. Springer-Verlag LNCS vol. 3350, 2005. 89