Using First-Order Theorem Provers in the Jahob Data Structure Verification System

Charles Bouillaguet[1], Viktor Kuncak[2],
Thomas Wies[3], Karen Zee[2], and Martin Rinard[2]

[1] Ecole Normale Supérieure de Cachan, Cachan, France
`charles.bouillaguet@ens.fr`
[2] MIT Computer Science and Artificial Intelligence Lab, Cambridge, USA
`{vkuncak,kkz,rinard}@csail.mit.edu`
[3] Max-Planck-Institut für Informatik, Saarbrücken, Germany
`wies@mpi-inf.mpg.de`

Abstract. This paper presents our integration of efficient resolution-based theorem provers into the Jahob data structure verification system. Our experimental results show that this approach enables Jahob to automatically verify the correctness of a range of complex dynamically instantiable data structures, such as hash tables and search trees, without the need for interactive theorem proving or techniques tailored to individual data structures.

Our primary technical results include: (1) a translation from higher-order logic to first-order logic that enables the application of resolution-based theorem provers and (2) a proof that eliminating type (sort) information in formulas is both sound and complete, even in the presence of a generic equality operator. Our experimental results show that the elimination of type information often dramatically decreases the time required to prove the resulting formulas.

These techniques enabled us to verify complex correctness properties of Java programs such as a mutable set implemented as an imperative linked list, a finite map implemented as a functional ordered tree, a hash table with a mutable array, and a simple library system example that uses these container data structures. Our system verifies (in a matter of minutes) that data structure operations correctly update the finite map, that they preserve data structure invariants (such as ordering of elements, membership in appropriate hash table buckets, or relationships between sets and relations), and that there are no run-time errors such as null dereferences or array out of bounds accesses.

1 Introduction

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 paper explores the verification of programs with such data structures using resolution-based theorem provers for first-order logic with equality. We only summarize the main ideas here; see [4] for details.

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. This approach is suitable for verifying clients of data structures, because such verification need not deal with transitive closure present in the implementation of recursive data structures. The context of this work is the Jahob system for verifying data structure consistency properties [7]. Our initial goal was to incorporate first-order theorem provers into Jahob to verify data structure clients. While we have indeed successfully verified data structure clients, we also discovered that this approach has a wider range of applicability than we had initially anticipated, in several respects. 1) 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. 2) Theorem provers were effective at dealing with quantified invariants that often arise when reasoning about unbounded numbers of objects. 3) Using a simple partial axiomatization of linear arithmetic, we were able to verify not only linking properties traditionally addressed by shape analyses, but also ordering properties in a binary search tree, hash table invariants, and bounds for all array accesses.

The context of our results. We find our current results encouraging and attribute them to several factors. Our use of ghost variables eliminated the need for transitive closure in specifications for our examples. Our use of recursion in combination with Jahob's approach to handling procedure calls resulted in more tractable verification conditions. 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 makes verification non-trivial even under these assumptions, 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 paper; the use of the third technique confirms previous observations about the usefulness of assumption filtering in automatically generated first-order formulas [13].

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
   a) 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 the postcondition
   $\mathsf{content} = (\mathsf{old\,content} \setminus \{(x,y) \mid x = \mathsf{key}\}) \cup \{(\mathsf{key}, \mathsf{value})\};$
   b) 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;
   c) preserves the representation invariants; and
   d) 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 to 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 paper to help make future theorem provers even more useful for program verification tasks. From the program verification perspective, our experience suggests that we will soon have a verified library of linked data structures that we can use to build and verify larger applications.

```
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}"
```

Fig. 1. Method contracts for a tree implementation of a map

2 Binary Tree Example

We illustrate our technique using an example of a binary search tree implementing a finite map. Our implementation is written in Java and is persistent, which means that the data structure operations do not mutate existing objects, only newly allocated objects. This makes the verification easier and provides a data structure which is useful in, for example, backtracking algorithms.

Figure 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. In Jahob, the developer specifies annotations such as procedure contracts in special comments `/*: ... */` that begin with a colon. The formulas in annotations belong to an expressive subset of the language used by the Isabelle proof assistant [16]. This language supports set comprehensions and tuples, which makes the specification of procedure contracts in this example very natural. Single dot `.` informally means "such that", both for quantifiers and set comprehensions. The notation `f x` denotes function `f` applied to argument `x`. Jahob models instance fields as functions from objects to values (objects, integers, or booleans). The operator `..` is a variant of function application given by `x..f = f x`. Operator `:` denotes set membership, `~=` denotes disequality, `Un` (or, overloaded, `+`) denotes union and `\<setminus>` (or, overloaded, $-$) denotes set difference.

```
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);
}
```

Fig. 2. Lookup operation for retrieving the element associated with a given key

Figure 2 presents the tree 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`, `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 cannot be captured by the public precondition; they are instead given by *representation invariants*. Figure 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 the invariants in Figure 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.

```
class FuncTree {
  private int key;
  private Object data;
  private FuncTree left, right;
 /*:
 public ghost specvar content :: "(int * obj) set" = "{}";

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

 invariant contentDefinition: "this ~= null  -->
             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" */
```

Fig. 3. Fields and representation invariants for the tree implementation

Figure 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, which means that its value changes only in response to specification assignment statements, such as the one in the penultimate line of Figure 4. The use of ghost variables is sound and can be explained using simulation relations [5]. 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 2.

Jahob verifies (in 10 seconds) that the update operation establishes the postcondition, correctly maintains all invariants, and performs no null dereferences. 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 paper 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 [16]. In the rest of this paper we show how we translate such verification conditions to first-order logic and prove them using theorem provers such as SPASS [22] and E [20].

```
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;
}
```

Fig. 4. Map update implementation for functional tree

3 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 output formula is valid, so is the input formula. The details of the translation are in [4].

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 paper to be combined with other translations [8, 23]. 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 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. 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 constructs in first-order logic with equality.

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 $\varphi[y \mapsto x]$. 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 in this paper. 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 $\mathsf{fieldWrite}$ and $\mathsf{arrayWrite}$ in our input language) translate using case analysis. If applied to arbitrary expressions, such case analysis would duplicate complex subterms, potentially leading to an exponential expansion. To avoid this problem, the translation first flattens expressions by introducing fresh variables and then duplicates only variables and not complex expressions, keeping the size of 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 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 examples.

Tuples. Tuples in the input language are useful, for example, as elements of sets representing relations, such as the `content` ghost field in Figure 3. Our translation eliminates tuples by transforming them into individual components. 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 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 other Jahob components can translate into monadic second-order logic [23]) and symbolic cardinality constraints (as in BAPA [8]). 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.

4 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 [2] (we allow some simple forms of parametric 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 $\mathsf{obj\,set}$. It encodes primitive Java integers as mathematical integers of type int (which is disjoint from obj). The result of the translation in Section 3 is a formula in multisorted first-order logic with equality and two disjoint sorts, obj and int.[4] 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 [11, Chapter 6, Section 8] is to introduce a unary predicate P_s for each sort s and replace $\exists x{::}s.F(x)$ with $\exists x.P_s(x) \land F(x)$ 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 $s_1 \times \dots s_n \to s$, introduce a Horn clause $\forall x_1, \dots, x_n.\ P_{s_1}(x_1) \land \dots \land P_{s_n}(x_n) \Rightarrow P_s(f(x_1, \dots, x_n))$.

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 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 and have the same cardinality. Moreover, there is no overloading of predicate or function symbols other than equality. Under these assumptions, we have the following result. Let φ^* denote the result of omitting all sort information from a multisorted formula φ and representing the equality (regardless of the sort of arguments) using the built-in equality symbol.

Theorem 1. *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,*

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

[4] The resulting multisorted logic has no sort corresponding to booleans (as in [11, Chapter 6]). Instead, propositional operations are part of the logic itself.

The proof of Theorem 1 is in [4, Appendix F]. 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 $f_i : S_i \to S$. Theorem 1 implies that if a formula $(\neg\psi)^*$ is unsatisfiable, then so is $\neg\psi$. Therefore, if ψ^* is valid, so is ψ.

A resolution theorem prover with paramodulation rules can derive ill-sorted clauses as consequences of φ^*. However, Theorem 1 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 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.

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.
FuncTree.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
FuncTree.RemoveMax	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

Fig. 5. Verification time, and proof data using the provers SPASS and E, on the hardest formulas from our examples.

5 Experimental Results

We implemented our translation to first-order logic and the interfaces to the first-order provers E [20] (using the TPTP format for first-order formulas [21]) and

SPASS [22] (using its native format). We also implemented filtering described in [4, Appendix A] 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 2, 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 6 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 comments. [5]

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.

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.

6 Related Work

We are not aware of any other system capable of verifying, without interactive theorem proving, such strong properties of operations on data structures that use arrays, recursive memory cells, and integer keys.

[5] 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.

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	Verification time (sec)	decision procedures (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	26
		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**	**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**

Fig. 6. Benchmarks Characteristics and Verification Times

Verification systems. Boogie [3] 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. Specification variables are present in Boogie [9] 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.

Abstract interpretation. Shape analyses [18,19] typically verify weaker properties than in our examples. In [10] the authors use the TVLA system to verify insertion sort and bubble sort. In [17, Page 35], the author uses TVLA to verify implementations of insertion and removal operations on sets implemented as mutable lists and binary search trees. The approach [17] 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 and results in much smaller running times. The analysis we describe in this paper 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 [6,12,14] 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 [6,14] 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 [1]. In contrast, we were able to prove that omitting sort information preserves soundness and completeness when sorts are disjoint and have the same cardinality.

Type systems and separation logic. Recently, researchers have developed a promising approach [15] 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.

References

1. K. Arkoudas, K. Zee, V. Kuncak, and M. 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.

2. H. P. Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science, Vol. II.* Oxford University Press, 2001.
3. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *CASSIS: Int. Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart devices*, 2004.
4. C. Bouillaguet, V. Kuncak, T. Wies, K. Zee, and M. 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.
5. W.-P. de Roever and K. Engelhardt. *Data Refinement: Model-oriented proof methods and their comparison.* Cambridge University Press, 1998.
6. J. Hurd. An LCF-style interface between HOL and first-order logic. In *CADE-18*, 2002.
7. V. Kuncak. *Modular Data Structure Verification.* PhD thesis, EECS Department, Massachusetts Institute of Technology, February 2007.
8. V. Kuncak, H. H. Nguyen, and M. Rinard. Deciding Boolean Algebra with Presburger Arithmetic. *J. of Automated Reasoning*, 2006. http://dx.doi.org/10.1007/s10817-006-9042-1.
9. K. R. M. Leino and P. Müller. A verification methodology for model fields. In *ESOP'06*, 2006.
10. T. Lev-Ami, T. Reps, M. Sagiv, and R. Wilhelm. Putting static analysis to work for verification: A case study. In *Int. Symp. Software Testing and Analysis*, 2000.
11. M. Manzano. *Extensions of First-Order Logic.* Cambridge University Press, 1996.
12. J. Meng and L. C. Paulson. Experiments on supporting interactive proof using resolution. In *IJCAR*, 2004.
13. J. Meng and L. C. Paulson. Lightweight relevance filtering for machine-generated resolution problems. In *ESCoR: Empirically Successful Computerized Reasoning*, 2006.
14. J. Meng and L. C. Paulson. Translating higher-order problems to first-order clauses. In *ESCoR: Empir. Successful Comp. Reasoning*, pages 70–80, 2006.
15. H. H. Nguyen, C. David, S. Qin, and W.-N. Chin. Automated verification of shape, size and bag properties via separation logic. In *VMCAI*, 2007.
16. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer-Verlag, 2002.
17. J. Reineke. Shape analysis of sets. Master's thesis, Universität des Saarlandes, Germany, June 2005.
18. R. Rugina. Quantitative shape analysis. In *Static Analysis Symposium (SAS'04)*, 2004.
19. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM TOPLAS*, 24(3):217–298, 2002.
20. S. Schulz. E – A Brainiac Theorem Prover. *Journal of AI Communications*, 15(2/3):111–126, 2002.
21. G. Sutcliffe and C. B. Suttner. The tptp problem library: Cnf release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998.
22. 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.
23. T. Wies, V. Kuncak, P. Lam, A. Podelski, and M. Rinard. Field constraint analysis. In *Proc. Int. Conf. Verification, Model Checking, and Abstract Interpratation*, 2006.