Computer Science and Artificial Intelligence Laboratory

Technical Report

MIT-CSAIL-TR-2006-072 November 9, 2006

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

Charles Bouillaguet, Viktor Kuncak, Thomas Wies, Karen Zee, and Martin Rinard

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

On 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, 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, including 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 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.

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 [23]. 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. The context of this work is the Jahob system for verifying data structure consistency properties [20]. 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.

- 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 properties to decidable fragments of first-order logic as suggested in [23], 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 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 our specifications. 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 made verification nontrivial 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 [32].

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 the 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. We expect that the techniques we identify in this paper will help make future theorem provers even more useful for program verification tasks.

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.

```
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

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 [36]. 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.

```
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

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.

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 [11]. For example, if the developer incorrectly specifies specifica-

```
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

tion 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). Jahob then splits the verification condition into a number of smaller formulas, each of which can be potentially proved using a different theorem prover or a decision procedure. The specification language and the generated verification conditions in Jahob are expressed in higher-order logic [36]. 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 [43] and E [41].

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.

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 [22,44,45]. 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 (using rules in Figure 7). 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 8).

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 $x \in \{y \,|\, \varphi\}$ with $\varphi[y \mapsto x]$. (Figure 9 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 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 fieldWrite and arrayWrite in our input language) translate using case analysis (Figure 10). 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 12) 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 3. Our translation eliminates tuples by transforming them into individual components. Figure 11 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 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 can often be translated into monadic second-order logic [44, 45]) and symbolic cardinality constraints (as in BAPA [21]). 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. [4]

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 [5] (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 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.[5] 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 [28, Chapter 6, Section 8] is to introduce a unary predicate P_s for each sort s and replace $\exists x{::}s.F(x)$ with $\exists x.P_s(x) \wedge F(x)$ 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 $s_1 \times \ldots s_n \rightarrow s$, introduce a Horn clause $\forall x_1, \ldots, x_n.\ P_{s_1}(x_1) \wedge \ldots \wedge P_{s_n}(x_n) \Rightarrow P_s(f(x_1, \ldots, x_n))$.

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 $\forall x.F(x) \wedge \exists x.\neg F(x)$ 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 [3] (without paramodulation) under these conditions, we can observe the following.

[4] We encountered an example of a formula $\varphi_1 \wedge \varphi_2$ 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 $\varphi_1 \wedge \varphi_2$.

[5] The resulting multisorted logic has no sort corresponding to booleans (as in [28, Chapter 6]). Instead, propositional operations are part of the logic itself.

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

Theorem 1. *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 1 is in 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. 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 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 [41] (using the TPTP format for first-order formulas [42]) and SPASS [43] (using its native format). We also implemented filtering described in 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 (Appendix H gives benchmark details).

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. [6]

[6] 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+ 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**

Fig. 6. Benchmarks Characteristics and Verification Times

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.

6 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 [6] 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 [9] is a verification system for Java that uses JML [24] 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 [4, 7, 13], 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 paper, 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 [2, 37], 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 [26] and ESC/Java2 [8] 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 [14, 15, 19, 47].

The Krakatoa tool [29] 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 [25, 39, 40] typically verify weaker properties than in our examples. In [27] the authors use the TVLA system to verify insertion sort and bubble sort. In [38, 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 [38] 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 [16, 31, 33] 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 [16, 33] 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.

The filtering of assumptions also appears in [32]. 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 [23]. 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 [10], array bounds [46], and mutable structures [48], usually enforcing weaker properties than in our case. Recently, researchers have developed a promising approach [34] based on separation logic [17] 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 [12] the authors use a constraint solver based on translation to propositional logic to identify all errors within a given scope. They apply the technique to analysis of real-world implementations of linked lists. Another approach for finding bugs is exhaustive testing by generating tests that satisfy given preconditions [18, 30]. These techniques are very effective at finding bugs, but do not guarantee the absence of errors.

7 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.

Acknowledgements. We thank Konstantine Arkoudas, Lawrence Paulson, Stephan Schulz, and Christoph Weidenbach for useful discussions. We thank VMCAI'07 reviewers for useful feedback.

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. A. Armando, S. Ranise, and M. Rusinowitch. A rewriting approach to satisfiability procedures. *Information and Computation*, 183(2):140–164, 2003.
3. L. Bachmair and H. Ganzinger. Resolution theorem proving. In *Handbook of Automated Reasoning (Volume 1)*, chapter 2. Elsevier and The MIT Press, 2001.
4. T. Ball, S. Lahiri, and M. Musuvathi. Zap: Automated theorem proving for software analysis. Technical Report MSR-TR-2005-137, Microsoft Research, 2005.
5. H. P. Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science, Vol. II*. Oxford University Press, 2001.
6. 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.
7. C. Barrett and S. Berezin. CVC Lite: A new implementation of the cooperating validity checker. In *Proc. 16^{th} Int. Conf. on Computer Aided Verification (CAV '04)*, volume 3114 of *Lecture Notes in Computer Science*, pages 515–518, 2004.
8. D. R. Cok. Reasoning with specifications containing method calls and model fields. *Journal of Object Technology*, 4(8):77–103, September–October 2005.
9. D. R. Cok and J. R. Kiniry. Esc/java2: Uniting ESC/Java and JML. In *CASSIS: Construction and Analysis of Safe, Secure and Interoperable Smart devices*, 2004.
10. R. Davies. *Practical Refinement-Type Checking*. PhD thesis, CMU, 2005.
11. W.-P. de Roever and K. Engelhardt. *Data Refinement: Model-oriented proof methods and their comparison*. Cambridge University Press, 1998.
12. G. Dennis, F. Chang, and D. Jackson. Modular verification of code with SAT. In *ISSTA*, 2006.
13. D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Laboratories Palo Alto, 2003.
14. J. V. Guttag, E. Horowitz, and D. R. Musser. Abstract data types and software validation. *Communications of the ACM*, 21(12):1048–1064, 1978.
15. M. Huisman. *Java program verification in higher order logic with PVS and Isabelle*. PhD thesis, University of Nijmegen, 2001.
16. J. Hurd. An LCF-style interface between HOL and first-order logic. In *CADE-18*, 2002.
17. S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *Proc. 28th ACM POPL*, 2001.
18. S. Khurshid and D. Marinov. TestEra: Specification-based testing of java programs using SAT. *Autom. Softw. Eng.*, 11(4):403–434, 2004.
19. V. Kuncak. Binary search trees. The Archive of Formal Proofs, `http://afp.sourceforge.net/`, April 2004.
20. V. Kuncak. *Modular Data Structure Verification*. PhD thesis, EECS Department, Massachusetts Institute of Technology, February 2007.
21. V. Kuncak, H. H. Nguyen, and M. Rinard. An algorithm for deciding BAPA: Boolean Algebra with Presburger Arithmetic. In *20th International Conference on Automated Deduction, CADE-20*, Tallinn, Estonia, July 2005.
22. V. Kuncak, H. H. Nguyen, and M. Rinard. Deciding Boolean Algebra with Presburger Arithmetic. *Journal of Automated Reasoning*, 2006. http://dx.doi.org/10.1007/s10817-006-9042-1.
23. V. Kuncak and M. Rinard. Decision procedures for set-valued fields. In *1st International Workshop on Abstract Interpretation of Object-Oriented Languages (AIOOL 2005)*, 2005.
24. G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML. Technical Report 96-06p, Iowa State University, 2001.
25. O. Lee, H. Yang, and K. Yi. Automatic verification of pointer programs using grammar-based shape analysis. In *ESOP*, 2005.

26. K. R. M. Leino and P. Müller. A verification methodology for model fields. In *ESOP'06*, 2006.
27. T. Lev-Ami, T. Reps, M. Sagiv, and R. Wilhelm. Putting static analysis to work for verification: A case study. In *International Symposium on Software Testing and Analysis*, 2000.
28. M. Manzano. *Extensions of First-Order Logic*. Cambridge University Press, 1996.
29. 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.
30. D. Marinov. *Automatic Testing of Software with Structurally Complex Inputs*. PhD thesis, MIT, 2005.
31. J. Meng and L. C. Paulson. Experiments on supporting interactive proof using resolution. In *IJCAR*, 2004.
32. J. Meng and L. C. Paulson. Lightweight relevance filtering for machine-generated resolution problems. In R. S. Geoff Sutcliffe and C. W. P. Stephan Schulz (editors), editors, *ESCoR: Empirically Successful Computerized Reasoning*, volume 192, 2006.
33. J. Meng and L. C. Paulson. Translating higher-order problems to first-order clauses. In R. S. Geoff Sutcliffe and C. W. P. Stephan Schulz (editors), editors, *ESCoR: Empirically Successful Computerized Reasoning*, volume 192, pages 70–80, 2006.
34. 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.
35. R. Nieuwenhuis and A. Rubio. Paramodulation-based theorem proving. In *Handbook of Automated Reasoning (Volume 1)*, chapter 7. Elsevier and The MIT Press, 2001.
36. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer-Verlag, 2002.
37. V. Prevosto and U. Waldmann. SPASS+T. In *ESCoR: Empirically Successful Computerized Reasoning*, volume 192, 2006.
38. J. Reineke. Shape analysis of sets. Master's thesis, Universität des Saarlandes, Germany, June 2005.
39. R. Rugina. Quantitative shape analysis. In *Static Analysis Symposium (SAS'04)*, 2004.
40. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM TOPLAS*, 24(3):217–298, 2002.
41. S. Schulz. E – A Brainiac Theorem Prover. *Journal of AI Communications*, 15(2/3):111–126, 2002.
42. G. Sutcliffe and C. B. Suttner. The tptp problem library: Cnf release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998.
43. 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.
44. 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.
45. T. Wies, V. Kuncak, K. Zee, A. Podelski, and M. Rinard. On verifying complex properties using symbolic shape analysis. Technical Report MPI-I-2006-2-1, Max-Planck Institute for Computer Science, 2006.
46. H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. *SIGPLAN Notices*, 33(5):249–257, 1998.
47. K. Zee, P. Lam, V. Kuncak, and M. Rinard. Combining theorem proving with static analysis for data structure consistency. In *International Workshop on Software Verification and Validation (SVV 2004)*, Seattle, November 2004.
48. D. Zhu and H. Xi. Safe programming with pointers through stateful views. In *Proc. 7th Int. Symp. Practical Aspects of Declarative Languages*. Springer-Verlag LNCS vol. 3350, 2005.

A 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 [32], 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.

B HOL to FOL Translation Rules

Translation rules that formalize the description in Section 3 are in Figures 7, 8, 9, 10, and 11. Figure 12 shows our partial axiomatization of linear arithmetic.

VAR-TRUE

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

VAR-FALSE

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

VAR-DEF

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

$v \notin FV(\varphi)$
VAR-TRUE cannot be applied
VAR-FALSE cannot be applied

Fig. 7. Rules for definition substitution

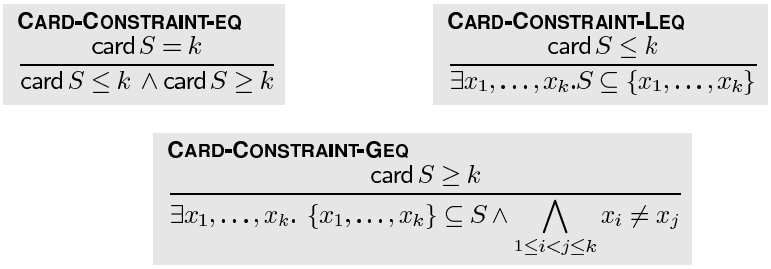

Fig. 8. Rules for constant cardinality constraints

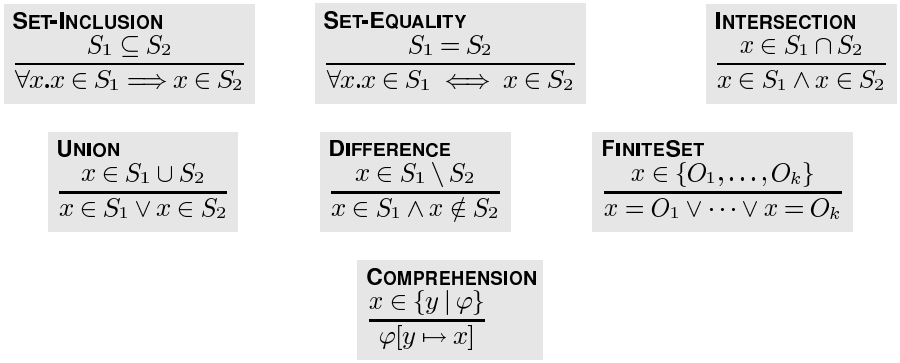

Fig. 9. Rules for complex set expressions

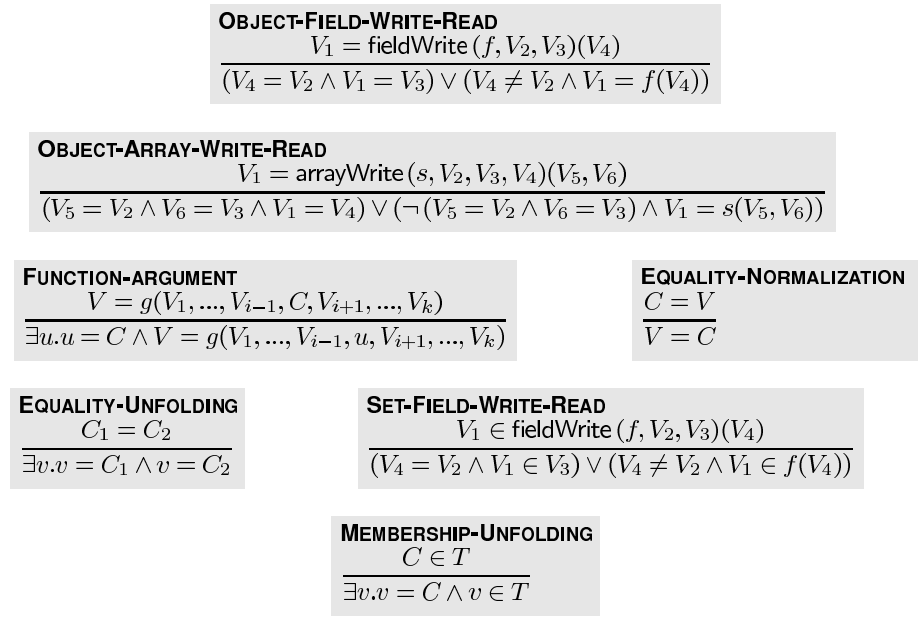

Fig. 10. Rewriting rules to rewrite complex field expressions. C denotes a complex term; s denotes a simple term.

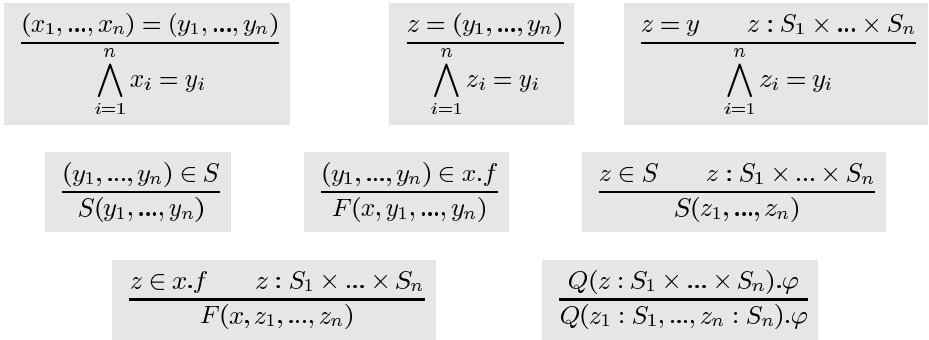

Fig. 11. Rules for removal of tuples

$$
\begin{aligned}
\forall n.\ & n \leq n \\
\forall n\, m.\ & (n \leq m \wedge m \leq n) \Rightarrow n = m \\
\forall n\, m.\ & (n \leq m \wedge m \leq o) \Rightarrow n \leq o \\
\forall n\, m.\ & (n \leq m) \iff (n = m \vee \neg(m \leq n)) \\
\forall n\, m\, p\, q.\ & (n \leq m \wedge p \leq q) \Rightarrow n + p \leq m + q \\
\forall n\, m\, p\, q.\ & (n \leq m \wedge p \leq q) \Rightarrow n - q \leq m - p \\
\forall n\, m\, p.\ & n \leq m \Rightarrow n + p \leq m + p \\
\forall n\, m\, p.\ & n \leq m \Rightarrow n - p \leq m - p \\
\forall n\, m.\ & n + m = m + n \\
\forall n\, m\, p.\ & (n + m) + p = n + (m + p) \\
\forall n.\ & n + 0 = n \\
\forall n.\ & n - 0 = n \\
\forall n.\ & n - n = 0
\end{aligned}
$$

Fig. 12. Arithmetic axioms optionally conjoined with the formulas

C 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.

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

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

Fig. 13. Syntax of Unsorted First-Order Logic with Equality

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

Fig. 14. Syntax of Multisorted First-Order Logic with Equality

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:

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

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

C.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 $\mathsf{ar}(f) : s_1 * \ldots * s_n \to s_{n+1}$ if $\mathsf{ar}(f) = (s_1, \ldots, s_n, s_{n+1})$.

Figure 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 $s_1 * \ldots * s_n \to s_{n+1}$, 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.

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

C.3 Omitting Sorts

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 15 make this definition more precise.

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

Fig. 15. Unsorted formula associated with a multisorted formula

D 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.

D.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 2 (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 3 (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 4 (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 $\mathcal{P}$.*

Definition 5 (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 6 (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 $\sigma_{\mathcal{P}} = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$.*

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

Fig. 16. Unification algorithm

We define the unification algorithm as the set of rewriting rules in Figure 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 1. *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 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.

Lemma 1 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.

D.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.

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

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

Fig. 17. Resolution rules

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 17.

Lemma 2. *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 2 follows from Lemma 1: 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 2 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 2. *Let φ be a multisorted formula without equality. If φ^* is valid, so is φ.*

E Completeness of Omitting Sorts

This section continues Section D and argues that eliminating sort information does not reduce the number of provable formulas. The following lemma is analogous to Lemma 2 and states that resolution steps on multisorted clauses can be performed on the corresponding unsorted clauses.

Lemma 3. *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 2 we obtain Theorem 3.

Theorem 3. *Let φ be a many-sorted formula without equality. If φ is valid then so is φ^*.*

F Soundness of Omitting Sorts in Logic with Equality

Sections D and E 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:

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

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

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

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 1 stated in Section 4, which implies soundness of omitting sorts even in the presence of an overloaded equality operator. The key step in the proof of Theorem 1 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 4. *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 1. 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 4 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:

$$\begin{aligned}
[\![P(t_1, \ldots, t_n)]\!]^{\mathcal{I}}_{\rho} &= ([\![t_1]\!]^{\mathcal{I}}_{\rho}, \ldots, [\![t_n]\!]^{\mathcal{I}}_{\rho}) \in [\![P]\!]^{\mathcal{I}} \\
\text{(by definition of } \mathcal{I}^* \text{ and } f_i \text{ injectivity)} \quad &= (f_{i_1}([\![t_1]\!]^{\mathcal{I}}_{\rho}), \ldots, f_{i_n}([\![t_n]\!]^{\mathcal{I}}_{\rho})) \in [\![P]\!]^{\mathcal{I}^*} \\
\text{(by Lemma 4)} \quad &= ([\![t_1{}^*]\!]^{\mathcal{I}^*}_{\rho^*}, \ldots, [\![t_n{}^*]\!]^{\mathcal{I}^*}_{\rho^*}) \in [\![P]\!]^{\mathcal{I}^*} \\
&= [\![P(t_1, \ldots, t_n)^*]\!]^{\mathcal{I}^*}_{\rho^*}
\end{aligned}$$

- The cases $\varphi = \varphi_1 \wedge \varphi_2$ 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 $[\![\exists z{::}s_v.\varphi_0]\!]^{\mathcal{I}}_{\rho}$.

G Sort Information and Proof Length

Theorem 1 shows that omitting sort information is sound for disjoint sorts of the same cardinality. Moreover, experimental results in Section 4 show that omitting sorts is often beneficial compared to the standard relativization encoding of sorts using unary predicates [28, Chapter 6, Section 8], even for SPASS [43] 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 [35], 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 $1 \leq i \leq n$. 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.

H Further Details on Verified Benchmarks

This section gives some additional details about the benchmarks presented in Figure 6.

`FuncTree` is a functional binary search tree implementing a map, sketched in Section 2. To give an idea of the complexity of this implementation, Figures 18 shows the `remove` method, which is the most difficult to verify in the functional tree implementation. Note that Jahob can infer that it is enough to remove only one node from the tree, because the uniqueness of keys follows from the strict ordering constraints. The auxiliary operations `max` and `remove_max` are in Figures 19 and 20.

```
public static FuncTree remove(int k, FuncTree t)
/*:   ensures "result..content = t..content - {(x,y). x=k}" */
{
    if (t == null) return null;
    else if (k == t.key) {
       if ((t.right == null) && (t.left == null)) return null;
       else if (t.left == null) return t.right;
       else if (t.right == null) return t.left;
       else {
         Pair m = max(t.left);
         FuncTree foo = remove_max(t.left, m.key, m.data);

         FuncTree r = new FuncTree();
         r.key = m.key;
         r.data = m.data;
         r.left = foo;
         r.right = t.right;
         //: "r..content" := "t..content - {(x,y). x=k}"
         return r;
       }
    }
    else {
        FuncTree new_left;
        FuncTree new_right;
        if (k < t.key) {
            new_left = remove(k, t.left);
            new_right = t.right;
        } else {
            new_left = t.left;
            new_right = remove(k, t.right);
        }
        FuncTree r = new FuncTree();
        r.key = t.key;
        r.data = t.data;
        r.left = new_left;
        r.right = new_right;
        //: "r..content" := "t..content - {(x,y). x=k}"
        return r;
    }
}
```

Fig. 18. Removal of an element

```
class Pair {
    public int key;
    public Object data;
}
public static Pair max(FuncTree t)
/*: requires "t..content ~= {}"
    ensures "result ~= null
           & result..Pair.data ~= null
           & ((result..Pair.key, result..Pair.data) : t..content)
           & (ALL k. (k ~= result..Pair.key -->
               (ALL v. ((k,v) : t..content --> k < result..Pair.key))))"
*/
{
    if (t.right == null) {
        Pair r = new Pair();
        r.key = t.key;
        r.data = t.data;
        return r;
    } else {
        return max(t.right);
    }
}
```

Fig. 19. Computing the maximal element

```
private static FuncTree remove_max(FuncTree t, int k, Object v)
/*: requires "(k,v) : t..content &
         (ALL x.(x ~= k --> (ALL y. ((x,y) : t..content --> x < k)))) &
         theinvs"
    ensures "result ~= t
            & result..content = t..content - {(k,v)}
            & (ALL x. (ALL y. ((x,y) : result..content --> x < k)))
            & theinvs"
        */
{
    if (t.right == null) {
        return t.left;
    } else {
        FuncTree new_right = remove_max1(t.right, k, v);

        FuncTree r = new FuncTree();
        r.key = t.key;
        r.data = t.data;
        r.left = t.left;
        r.right = new_right;
        //: "r..content" := "t..content - {(k,v)}"
        return r;
    }
}
```

Fig. 20. Removal of the maximal element

`AssocList` is an implementation of an instantiable relation using a functional linked list. The methods in `AssocList` are pure in the sense that they only mutate newly allocated heap objects, without changing any objects allocated before procedure execution. The `remove_all` method in `AssocList` removes all bindings for a given key, and the `remove` method removes all bindings for the given key-value pair. This difference between the behaviors of `remove` and `remove_all` is naturally characterized by the different contracts of these two procedures. The `image` method returns the set of values bound to a a given key. The `inverseImage` method is analogous. `domain` returns the set of all keys and `range` the set of all values. The contracts of these procedures are easily expressed using set algebra and set comprehension.

`ImperativeList` is an implementation of an instantiable set using an imperative linked list. Unlike `AssocList`, it performs mutations on an existing data structure. It contains operations for element insertion (`add`), element removal (`remove`), and testing the membership of elements in a list (`member`). We found that the verification of most operations was similarly easy as in the verification of insertion into a functional data structure. The verification of `remove` was more difficult. The `remove` method contains a loop, so we had to supply a loop invariant. Moreover, removal of an element x in the middle of the list requires updates to the specification fields of all elements preceding x in the list. These additional abstract mutations required us to state as loop invariant a variation of the class representation invariants.

The `Hashtable` class is an efficient imperative implementation of an instantiable relation. The class contains an encapsulated mutable array which stores entries of the (previously verified) `AssocList` class. The verification of the hash table uses the specification of `AssocList` in terms of a single relation instead of having to reason about the linked list data structure. This implementation shows how hierarchical abstraction is useful even when verifying individual data structures. The hash table implements standard `add`, `remove`, and `update` operations on key-value pairs with natural preconditions and postconditions. To verify the correctness of `remove` and `update`, the class contains a representation invariant (named `Coherence` in Figure 21) specifying that a node storing a given key-value pair is in the bucket whose array index is a function of the key. Our specification uses an uninterpreted specification variable `h` to represent the result of computing the bucket for a given key object and a given table size, as opposed to verifying a user-specified hashing function. Our implementation of the `add` procedure dynamically resizes the array containing the buckets when the filling ratio is too high. As in standard hash table implementations, this operation rehashes all elements into a larger array, which we implemented and verified using a tail-recursive method.

The `Library` class implements a simple library system built on top of container data structures. It maintains two sets of objects: the set *persons* of registered users of a library system, and the set *books* of all publication items in the library. The system also maintains a *borrows* relation, which relates each person to the books that the person is currently using from the library. The sets and relations are implemented using the previously described container classes. The system enforces consistency properties between these three containers. For example, for each (person, book) pair in the borrows relation, the person belongs to the person set and the book belongs to the book set. Also, each book is borrowed by at most one person at any given point of time. The class supports library operations such as checking out and returning a book, checking who is the reader of a borrowed book, and removing a book from the system.

```
class Hashtable {
  private AssocList[] table = null;
  private int size;

/*:
  public ghost specvar content :: "(obj * obj) set"
  public ghost specvar init :: "bool" = "False :: bool"

  private static ghost specvar h :: "obj => int => int"

  invariant HiddenArray: "init --> table : hidden"

  invariant contentDefinition: "init -->
     content = {(k,v). EX i. 0 <= i & i < table..Array.length
                           & (k,v) : table.[i]..AssocList.content}"

  invariant TableNotNull: "init --> table ~= null"
  invariant TableAlloc: "table : Object.alloc"

  invariant Coherence: "init --> (ALL i k v.
   0 <= i & i < table..Array.length -->
       ((k,v) : table.[i]..AssocList.content -->
          h k (table..Array.length) = i))"

  invariant TableInjectivity: "ALL u v.
   u : Object.alloc & u : Hashtable & u ~= null &
   v : Object.alloc & v : Hashtable & v ~= u  --> u..table ~= v..table"

  invariant TableSize: "init --> table..Array.length > 0"  */
```

Fig. 21. Class invariants of the hash table implementation