Using First-Order Theorem Provers in Data Structure Verification

Charles Bouillaguet
Ecole Normale Supérieure, Cachan, France

Viktor Kuncak
Martin Rinard
MIT CSAIL
Inconsistent data structures

- Can cause program crashes

Unconsistent outcome of operations
removing two instead of one element

Looping
Implementing data structures is hard

- Often small, but complex code
- Lots of pointers
- Unbounded, dynamic allocation
- Complex shape invariants
  - dag, parents pointers
  - Properties involving arithmetic (ordering…)
- Need strong invariants to guarantee correctness
  - e.g. lookup in ordered tree needs sortedness
How to obtain reliable data structure implementations?

- **Approach**
  - Prove that the program is **correct**
  - for **all program executions** (sound)

- **Verified properties:**
  - Program does not crash in data structure
  - Data structure invariants are preserved
  - Data structure content is correctly updated

- **Goal:** high level of **automation**
- **Infrastructure:** **Jahob** system for verifying data structure implementation
Summary of verified data structures

- Implementations of sets
  - add an element
  - get an arbitrary element
  - remove a given element
  - test membership
  - test emptiness

- Implementations of relations
  - add a binding
  - remove all bindings for a given key
  - test key membership
  - retrieve data bound to a key
  - test emptiness

verified data structures:
- linked list
- ordered tree
- hash table
Example verified client

- Implementations of sets
- Implementations of relations

Implementation of a library system
- get the current reader of a book
- get the books of a reader
- check out a book from the library
- return a book
- decommission a book

- Internal consistency
Outline

- Introduction
- Example: ordered trees
- Overview of the verification process
- Translation to First-Order Logic
- Sorts elimination
- Assumption filtering
- Experimental results
- Related work
- Conclusions
An Example: Ordered Trees

- Implementation of a finite map
- Each Node has a key, a value, a left and right subtree
- Recursive, functional (pure) methods
  - mutate only newly allocated objects
  - keep multiple versions efficiently
  - easier to verify
- Operations: insert, lookup, remove
- Representation invariants:
  - tree shaped (acyclicity, unique parent)
  - ordering constraints
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,v) ~: t..content)"
    ensures "result..content = t..content Un {(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 "((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}"
public final class FuncTree {
    private int key;
    private Object data;
    private FuncTree left, right;

    /*: public ghost specvar content :: "(int * obj) set" = {};
    invariant ("content definition") "this ~= null --> content = {(key, data)} Un left..content Un right..content"
    invariant ("null implies empty") "this = null --> content = {}"
    invariant ("left children are smaller")
      "ALL k v. (k,v) : left..content --> k < key"
    invariant ("right children are bigger")
      "ALL k v. (k,v) : right..content --> k > key"
    */

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) { … } else {
        FuncTree new_left, new_right;
        if (k < t.key) {
            new_left = remove(k, t.left);
            new_right = t.right;
        } else {
            …
        }
    } else {
    …
}
FuncTree r = new FuncTree();
r.key = t.key; r.data = t.data;
r.left = new_right; r.right = new_right;
//: "r..content" := "t..content - {(x,y). x=k}"
return r;
}
How to verify these properties?
How to verify these properties?

- Transform program into a logic formula
  - Using weakest precondition
  - The program is correct iff the formula is valid

- Prove the formula
  - very difficult formulas: interactively (Coq, Isabelle)
  - decidable classes: automated (MONA, CVCL)
  - this talk: difficult formulas in automated way :)  
    • use first-order provers: SPASS, E, Vampire

low efficiency  
1 line per grad student minute  
parallelization looks non-trivial
Formulæ generation outline

- java files
- java parser
- specification parser
- three-address code
- loops/calls desugaring
- Loop-free Guarded Command language
- Verification condition generator
- HOL Formula
new_left = remove(k, t.left);
\[ r.data = t.data; \]
\[ \text{tmp}_27 = t.left; \]
\[ \text{tmp}_28 = \text{FuncTree}.remove(k, \text{tmp}_27) \]
\[ \text{new_left} = \text{tmp}_28; \]
\[ \text{tmp}_35 = t.data; \]
\[ r.data = \text{tmp}_35; \]

```
flatten expressions using fresh variables
```

```
java
parser
specification
parser
three-address code
loops/calls
desugaring
Loop-free Guarded
Command language
Verification condition
generator
HOL Formula
```
Formula generation outline

1. java files
2. java parser
3. specification parser
4. three-address code
5. loops/calls desugaring
6. Loop-free Guarded Command language
7. Verification condition generator
8. HOL Formula
9. Loop invariant inference
Formula generation outline

<table>
<thead>
<tr>
<th>Stmt</th>
<th>wlp(Stmt, φ)</th>
</tr>
</thead>
<tbody>
<tr>
<td>assert e</td>
<td>e ∧ φ</td>
</tr>
<tr>
<td>assume e</td>
<td>e ⇒ φ</td>
</tr>
<tr>
<td>x := e</td>
<td>φ(x := e)</td>
</tr>
<tr>
<td>Stmt₁ ; Stmt₂</td>
<td>wlp(Stmt₁, wlp(Stmt₂, φ))</td>
</tr>
<tr>
<td>Stmt₁ □ Stmt₂</td>
<td>wlp(Stmt₁, φ) ∧ wlp(Stmt₂, φ)</td>
</tr>
<tr>
<td>havoc x</td>
<td>∀x. φ</td>
</tr>
</tbody>
</table>

- Weakest Liberal Precondition
- Liberal = Termination not enforced
- adapted from Dijkstra ‘76
Formulas in Jahob

- Specification language: rich subset of Isabelle’s language.
  - Convenient to express complex properties
- Higher-Order features
  - Sets, set comprehension, cardinality, first-class functions, lambda binders, tuples, arbitrary quantification …
- We can use Isabelle to prove these formulas
  - by hand…
  - little automation, and slow
- How can we do it in a more automated way?
Automated reasoning in Jahob
First-Order Theorem Provers

- Resolution: complete (semi-algorithm for validity)
  - may loop/run out of memory on non-valid formulas
- Resolution-based automated theorem provers:
  - SPASS, E, Vampire, Theo, Prover9, Darwin
  - continuously improving (yearly competition)
  - effective on formulas with short proofs

- Can we use them to improve automation?

- Input: unsorted first-order logic with equality
Outline

- Introduction
- Example: ordered trees
- Verification process
- **Translation to First-Order Logic**
- Sorts elimination
- Assumption filtering
- Experimental results
- Related work
- Conclusions
Approach to translation HOL → FOL

- idea: translate what you can
  - lambda reduction and substitution
  - cardinality constraints
  - set expressions
  - detupling
  - fields, flattening

- Avoid translations with many axioms
  - e.g. avoid axiomatizing set theory

- Sound approximation for the rest
  - replace by True in assumptions
  - replace by False in goal
(but take polarity into account)
Lambda reduction and substitution

- No $\lambda$-binder, no partial functions in FOL, but uninterpreted function symbols
- Arguments applied to $\lambda$: $\beta$-reduction
- To trigger this situation: definition unfolding
  
  \[
  \text{content} = \lambda \text{this.} \{ \text{n..data} \mid \text{n : this..first} \} \rightarrow \\
  \text{result..content} = \{ \}
  \]

  becomes

  \[
  \{ \text{n..data} \mid \text{n : this..first} \} = \{ \}
  \]
Cardinality Constraints

- Rewrite using set inclusion and fresh constants

- Only possible to handle constant bounds
  - Would need more expressive BAPA otherwise
Reduction of Sets Expressions

- Standard set-theoretic reduction to the membership operator

\[
\{n..\text{data} \mid n : \text{this..first} \} = \{\}
\]

becomes

\[
\text{ALL } x. (\text{EX } n. x = n..\text{data} \land n : \text{this..first}) \leftrightarrow \text{False}
\]

- Membership easily expressed in FOL
Sets (cont’d)

- Sets: Unary predicates
  \[ x \in S \rightarrow S(x) \]

- Set-valued abstract fields: Binary predicates
  \[ x \in y.f \rightarrow F(x,y) \]

- We cannot afford quantification over sets
  - Not surprising in FOL!
  - Not a problem in practice
    \[ \text{result..content} = \text{t..content} - \{(x,y). \ x=k\} + \{(k,v)\} \]
Detupling

- Tuple expressions can be reduced
- A n-tuple variable is transformed into n variables
  - $\exists (x : O \times I). \varphi \rightarrow \exists (x_0 : O)(x_i : I). \lbrack \lbrack \varphi \rbrack \rbrack$
  - $x = y \rightarrow x_0 = y_0 \land x_i = y_i$
  - $f(x) \rightarrow f(x_0, x_i)$
- Sets of n-tuples become n-ary predicates
  - $x \in S \rightarrow S(x_0, x_i)$
Handling of fields

- In the specification language
  - Fields are functions:
    \[ y = x \cdot f \rightarrow y = f \cdot x \]
  - Fields modification generates a new function
    \[ x \cdot f = a \rightarrow f := (\lambda z. \text{if } z=x \text{ then } a \text{ else } f \cdot z) \]

- In FOL, def. unfolding + \( \beta \)-reduction
  \[ y = (\lambda z. \text{if } z=x \text{ then } a \text{ else } f \cdot z) \cdot u \]
  \[ \text{Becomes:} \]
  \[ (u = x \land y = a) \lor (u \neq a \land y = f \cdot u) \]

potentially exponential explosion !!!
Avoiding explosion: Flattening

- To avoid explosion, introduce fresh variables for non-variable duplicated terms

\[ y = (\lambda \, z. \text{if } z=x \text{ then } a \text{ else } f \, z) \, u \]

- Becomes:
  \[ \exists \, u', \, a'. \ (u' = u) \land (a' = a) \land [\ (u' = x \land y = a') \lor (u' \neq a' \land y = f \, u')] \]

- Polynomial expansion only
Avoiding alternation in flattening

- Careful introduction of fresh variables
  - Introduce using either $\exists$ or $\forall$, since:
    $$\begin{align*}
    (\exists \, x. \ x=a \land \varphi) & \iff (\forall \, x. \ x=a \Rightarrow \varphi)
    \end{align*}$$

- Use the same as the previous one
- If negation encountered, switch (or use NNF form)
- Start in existential mode in the assumptions
  - Introduces a constant instead of a variable, because of Skolemization in resolution provers
- Start in universal mode in the goal
Arithmetic

- Numbers are uninterpreted constants in FOL
  - Provers do not know that 1+1=2!

- Solutions
  - Provide an encoding: Peano (unary) or binary, and give rules for “+”, “≤”
    - Would be complete, but tremendously inefficient
  - Provide partial, incomplete axiomatization
    - Cannot deduce 1+1=2!
    - Usual order relation, comparison between constants in formula
    - Optionally, compatibility of “+” with “≤”
    - Satisfactory results in practice
    - Prove ordering constraint of the ordered tree
Observation

- Most formulas are fast/easy to prove
- Problem often concentrated in a small number that take very long to prove

- Next: two techniques to make them easier
Outline

- Introduction
- Example: ordered trees
- Verification process
- Translation to First-Order Logic
- **Sorts elimination**
- **Assumption filtering**
- Experimental results
- Related work
- Conclusions
Types and Sorts

▪ Java class hierarchy encoded as sets
  ▪ Flexible, automatically translated
▪ In Isabelle formulas, obj, int and bool types
  ▪ This type information can be encoded using unary predicates:
    
    - \( \forall (x : \text{Object}) \varphi \rightarrow \forall x. (\text{Object}(x) \Rightarrow \varphi) \)
    - \( \exists (x : \text{Object}) \varphi \rightarrow \exists x. (\text{Object}(x) \land \varphi) \)

▪ we need to declare sort of constants and function symbols
▪ Sorts can cut branching factor in prover
Omitting Sort Information

- Sort information is making formulas bigger and proofs longer.
  - On Tree.remove, average proof length grows from 10 to 20 when putting sort guards (in # of resolution steps)
  - Makes some formula much harder
Effect on hard formulas

- Formulas that take more than 1s to prove, from the Tree implementation

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Time (s)</th>
<th>Proof length</th>
<th>Generated clauses</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>with</td>
<td>w/o</td>
<td>with</td>
</tr>
<tr>
<td>Tree.remove</td>
<td>4.5</td>
<td>0.53</td>
<td>250</td>
</tr>
<tr>
<td></td>
<td>44.0</td>
<td>0.46</td>
<td>1 082</td>
</tr>
<tr>
<td></td>
<td>5.2</td>
<td>0.75</td>
<td>209</td>
</tr>
<tr>
<td></td>
<td>30.1</td>
<td>0.38</td>
<td>869</td>
</tr>
<tr>
<td></td>
<td>5.8</td>
<td>0.75</td>
<td>249</td>
</tr>
<tr>
<td></td>
<td>7.3</td>
<td>0.28</td>
<td>863</td>
</tr>
<tr>
<td>Tree.remove_max</td>
<td>83.1</td>
<td>4.8</td>
<td>797</td>
</tr>
<tr>
<td></td>
<td>37.9</td>
<td>0.85</td>
<td>2 622</td>
</tr>
</tbody>
</table>
Omitting Sorts (cont’d)

- Great speed-up (up to 100 times)!
- However:
  \[ \forall (x \, y : S). \, x = y \]
  \[ \exists (x \, y : T). \, x \neq y \]
  - Satisfiable with sorts \((S=\{a\}, \, T=\{b, c\})\)…
  - Unsatisfiable without!
- Omitting sort guards breaks soundness!!!
Omitting Sorts Theorem

We proved the following

Theorem. Suppose that
  i. Sorts are pair-wise disjoint (no sub-sorting)
  ii. Sorts have the same cardinality

Then omitting sort guards is

sound and complete

This justify this useful optimization
Assumption filtering

- Provers get confused by too many assumptions
- Lots of useless assumptions
  - Hardest shown benchmark needs 12 out of 56
  - Gets worse on harder problem (Hash table)
    - Hashtable.Add: 211 sec with full assumptions
    - Array bound check requires order axioms
    - Order axioms confuse provers, even when proof do not require them

- Assumption filtering
  - Try to eliminate of irrelevant assumptions automatically
  - Give a score to assumption, then filter
Assumption scoring

- Idea: symbol tracking
  - relevant assumptions contain relevant symbols
  - relevant symbols are contained in the goal and in relevant assumptions
  - assumptions get score based on proportion of relevant symbols they contain
  - score bigger than threshold:
    - assumption becomes relevant
    - relevant symbols are updated
  - Iterate several (=5) times

Hashtable.Add: 1.3 sec with filtered assumptions over 100 x speedup
## Experimental results

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>lines of code</th>
<th>lines of specification</th>
<th># of methods</th>
<th>verif. time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sets as functional linked list</td>
<td>60</td>
<td>24</td>
<td>9</td>
<td>7.5s</td>
</tr>
<tr>
<td>Sets as imperative linked list</td>
<td>60</td>
<td>47</td>
<td>6</td>
<td>17s</td>
</tr>
<tr>
<td>Relation as functional Linked list</td>
<td>76</td>
<td>26</td>
<td>9</td>
<td>60s</td>
</tr>
<tr>
<td>Relation as functional Ordered trees</td>
<td>186</td>
<td>38</td>
<td>10</td>
<td>70s</td>
</tr>
<tr>
<td>Relation as hash table (using f.list)</td>
<td>41</td>
<td>39</td>
<td>6</td>
<td>51s</td>
</tr>
</tbody>
</table>
Verification effort

- Decreased as we improved the system
  - functional list was easy
  - a few days for trees
  - two hours for hash table
- Currently the most usable method for proving formulas in Jahob
Related work

- Interactive Provers – Isabelle, Coq, HOL, PVS, ACL2
- First-Order ATP
  - Vampire – Voronkov [04]
  - SPASS – Weidenbach [01]
  - E – Shultz [IJCAR04]
- Program Checking
  - ESC/Java – Flanagan, Leino, Lillibridge, Nelson, Saxe, Stata ‘02
  - Krakatoa – Marche, Paulin-Mohring, Urbain [03]
  - Spec# – Barnett, DeLine, Jacobs, Fähndrich, Leino, Schulte, Venter [05]
  - Hob system: verify set implementations (we verify relations)
- Shape analysis
  - PALE - Møller and Schwartzbach [PLDI01]
  - TVLA - Sagiv, Reps, and Wilheim [TOPLAS02]
  - Roles - Kuncak, Lam, and Rinard [POPL02]
Conclusion

- Jahob verification system
- Automation by translation HOL → FOL
  - omitting sorts theorem gives speedup
  - filtering automates selection of assumptions
- Promising experimental results
  - strong properties: correct implementation
    - Do not crash
    - operations correctly update the content, clarifies behavior in case of duplicate keys, …
    - representation invariants preserved (ordering, treeness, each element is in appropriate bucket)
  - 180 lines in 70 seconds, hash table in seconds
  - verification effort much smaller than using interactive provers
Formal Methods are the Future of computer Science. Always have been… Always will be.

Questions ?
Converting to GCL

- **Conditionnal statement:** easy
  - 
    ```
    \[\text{if cond then } t\text{branch else } f\text{branch}\] =
    (\text{Assume cond; } \[\text{tbranch}\])
    \(\Box\) (\text{Assume } \neg\text{cond; } \[\text{fbranch}\])
    ```

- **Procedure calls:**
  - Could inline (potentially exponential blowup)
  - Desugaring (modularity):
    - ```
      \[\text{r = CALL } m(x, y, z)\] =
      \text{Assert (m’s precondition)};
      \text{Havoc } r;
      \text{Havoc \{vars modified by m\};}
      \text{Assume (m’s postcondition)}
    ```
Loops: invariant required

- while /*: invariant */ (condition) {lbody} |] =
  assert invariant;
  havoc vars(lbody);
  assume invariant;
  ((assume condition;
    [| lbody |];
    assert invariant;
    assume false)
  □ (assume !condition))

- invariant hold initially
- no assumptions on variables except that invariant hold
- condition hold
- invariant is preserved
- no need to verify anything more
- or condition do not hold and execution continues
Verification condition for remove

And 200 more kilobytes...

Infeasible to prove directly
Splitting heuristic

- Verification condition is big conjunction
  - conjunctions in postcondition
  - proving each invariant
  - proving each branch in program
- Solution: split VC into individual conjuncts
- Prove each conjunct separately
- Each conjunct has form
  \[ H_1 \land \ldots \land H_n \Rightarrow G_i \]
  Tree.Remove has 230 such conjuncts
- How do we prove them?
Detupling (cont’d)

- Complete rules:

\[
\frac{(x_1, \ldots, x_n) = (y_1, \ldots, y_n)}{\bigwedge_{i=1}^{n} x_i = y_i}
\]

\[
\frac{z = (y_1, \ldots, y_n)}{\bigwedge_{i=1}^{n} z_i = y_i}
\]

\[
\frac{z = y}{\bigwedge_{i=1}^{n} z_i = y_i}
\]

\[
\frac{(y_1, \ldots, y_n) \in S}{S(y_1, \ldots, y_n)}
\]

\[
\frac{(y_1, \ldots, y_n) \in x.f}{F(x, y_1, \ldots, y_n)}
\]

\[
\frac{z \in S}{z : S_1 \times \ldots \times S_n}
\]

\[
\frac{z \in S \times \ldots \times S_n}{S(z_1, \ldots, z_n)}
\]

\[
\frac{z \in x.f}{z : S_1 \times \ldots \times S_n}
\]

\[
\frac{Q(z : S_1 \times \ldots \times S_n) \cdot \varphi}{Q(z_1 : S_1, \ldots, z_n : S_n) \cdot \varphi}
\]
Handling of Fields (cont’d)

- We dealt with field updates
  - New function expressed in terms of old one
- Base case: field variables
  - Natural encoding in FOL using functions:

\[ x = y.f \rightarrow x = f(y) \]
Future work

- Verify more examples
  - balanced trees
  - fancy priority queues (binomial, Fibonacci, …)
  - hash table with dynamic resizing
- hash function
- verify clients of data structures
- Improve assumption filtering
  - take rarity of symbols into account
  - check for occurring polarity
  - …