An Integrated Proof Language for Imperative Programs

Karen Zee
Massachusetts Institute of Technology
CSAIL, Cambridge, MA, USA
kkz@csail.mit.edu

Viktor Kuncak
École Polytechnique Fédérale de Lausanne
IC, Lausanne, VD, Switzerland
viktor.kuncak@epfl.ch

Martin C. Rinard
Massachusetts Institute of Technology
CSAIL, Cambridge, MA, USA *
rinard@csail.mit.edu

Abstract

We present an integrated proof language for guiding the actions of multiple reasoning systems as they work together to prove complex correctness properties of imperative programs. The language operates in the context of a program verification system that uses multiple reasoning systems to discharge generated proof obligations. It is designed to 1) enable developers to resolve key choice points in complex program correctness proofs, thereby enabling automated reasoning systems to successfully prove the desired correctness properties; 2) allow developers to identify key lemmas for the reasoning systems to prove, thereby guiding the reasoning systems to find an effective proof decomposition; 3) enable multiple reasoning systems to work together productively to prove a single correctness property by providing a mechanism that developers can use to divide the property into lemmas, each of which is suitable for a different reasoning system; and 4) enable developers to identify specific lemmas that the reasoning systems should use when attempting to prove other lemmas or correctness properties, thereby appropriately confining the search space so that the reasoning systems can find a proof in an acceptable amount of time.

The language includes a rich set of declarative proof constructs that enables developers to direct the reasoning systems as little or as much as they desire. Because the declarative proof statements are embedded into the program as specialized comments, they also serve as verified documentation and are a natural extension of the assertion mechanism found in most program verification systems.

We have implemented our integrated proof language in the context of a program verification system for Java and used the resulting system to verify a collection of linked data structure implementations. Our experience indicates that our proof language makes it possible to successfully prove complex program correctness properties that are otherwise beyond the reach of automated reasoning systems.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; F.3.1 [*Logics and Meaning of Programs*]: Specifying and Verifying and Reasoning about Programs

* This research was supported in part by DARPA Cooperative Agreements SA 8750-06-2-0189 and FA 8750-04-2-0254; United States National Science Foundation Grants 0341620, 032583, 0509415, 0811397, and 0835652; and Swiss National Science Foundation Grant "Precise and Scalable Analyses for Reliable Software".

PLDI'09, June 15–20, 2009, Dublin, Ireland.

General Terms Algorithms, Languages, Reliability, Verification
Keywords verification, theorem prover, proof system

1. Introduction

We have developed a system, Jahob, for specifying and verifying Java programs [46]. Because Jahob uses higher-order logic as its specification language, it can express very sophisticated correctness properties. Instead of relying on a single monolithic prover, Jahob uses *integrated reasoning* — it contains interfaces to a variety of internal and external reasoning systems and uses these systems in a coordinated way to prove complex verification conditions. This approach allows Jahob to quickly exploit new reasoning systems as they become available and to incorporate specialized decision procedures that can prove important properties within arbitrarily narrow domains.

Jahob currently interfaces to first-order provers (SPASS [43] and E [41]), SMT provers (CVC3 [22] and Z3 [15]), MONA [39], and the BAPA decision procedure [27, 29]. If all of these provers working together fail to prove a verification condition, the developer can manually prove the desired property using Jahob's interfaces to interactive theorem provers (Isabelle [35] and Coq [10]).[1]

In theory, this approach makes it possible to solve program verification problems requiring arbitrarily complex reasoning. But in practice, to exploit this capability, developers must become proficient in both the verification system and an interactive theorem prover — two separate systems with radically different basic concepts, capabilities, and limitations. This approach also requires developers to maintain, in addition to the annotated program, a set of associated proof scripts. And although the interactive prover enables developers to manually prove verification conditions that fail to prove automatically, it also divorces the proof from its original context within the annotated program and denies the developer access to the substantial automated reasoning power available via the Jahob prover interfaces.

1.1 Integrated Proof Language

The Jahob proof language addresses these issues by making it possible for developers to control proofs of program correctness properties while remaining completely within a single unified programming and verification environment. The proof commands are directly included in the annotated program and are verified by the underlying reasoning system as part of the standard program verification workflow. Because the proof language is seamlessly integrated into the verification system, all of the automated reasoning capabilities of the Jahob system are directly available to the developer. We have found that this availability enables developers to avoid the use of external interactive theorem provers altogether. Instead, developers simply use the Jahob proof language to resolve key choice points in the proof search space. Once these choice points have

[1] The complete source code for Jahob, along with examples of its use, is available at javaverification.org.

been resolved, the automated provers can then perform all of the remaining steps required to discharge the verification conditions. This approach effectively leverages the complementary strengths of the developer and the automated reasoning system by allowing the developer to communicate key proof structuring insights to the reasoning system. These insights then enable the reasoning system to successfully traverse the (in practice large and complex) proof search space to obtain formal proofs of the desired verification conditions. The following techniques are of particular interest:

- **Lemma Identification:** The developer can identify key lemmas for the Jahob reasoning system to prove. These lemmas can then help the reasoning system find an appropriate proof decomposition. Such a proof decomposition can be especially important when multiple provers must cooperate to prove a single correctness property. In this case separating the property into lemmas, each of which contains facts suitable for a specific prover, then combining the lemmas, may be the only way to obtain a proof.
- **Witness Identification:** The developer can identify the witness that enables the proof of an existentially quantified verification condition. Because there are, in general, an unbounded number of potential witnesses (very few of which may lead to a successful proof), the difficulty of finding an appropriate witness is often a key obstacle that prevents a fully automated system from obtaining a proof. But our results show that enabling the developer to remove this key obstacle usually leaves the automated system easily able to successfully navigate the proof search space to prove the desired correctness property.
- **Quantifier Instantiation:** The developer can identify how to instantiate specific universally quantified formulas. The potentially unbounded number of possible quantifier instantiations can make developer insight particularly useful in enabling successful proofs.
- **Case Split Identification:** The developer can identify the specific cases to analyze for case analysis proofs.
- **Induction:** The developer can identify an induction variable and induction property that lead to a successful proof by induction.
- **Assumption Base Control:** Modern theorem provers are usually given a set of facts (we call this set the *assumption base*), then asked to prove a consequent fact that follows from this set. An assumption base that contains irrelevant facts can produce an overly large proof search space that impedes the ability of the provers to find a proof of the consequent. Our integrated proof language enables developers to control the assumption base (and thereby productively focus the proof search space on the property of interest) by identifying a set of relevant facts for the provers to use when proving a specific verification condition. We have found this functionality essential in enabling modern provers to successfully prove the complex verification conditions that arise in proofs of sophisticated program correctness properties.

1.2 Correctness of Linked Data Structures

Because of the challenges that aliasing, indirection, and an unbounded number of objects pose for automated reasoning systems, recursive linked data structures comprise an especially challenging verification problem [46]. This paper presents our experience using our integrated proof language to verify the correctness of a collection of linked data structures. Because the correctness proofs establish complex correctness properties of the data structure implementations, the verification conditions involve constructs (such as transitive closure and quantifiers) that are known to be intractable for automated reasoning systems [23, 28]. Despite this intractability, we are able to use the integrated proof language to successfully identify key proof structuring choices and thereby enable the automated reasoning system to perform the required correctness proofs. This approach eliminates the need to use external interactive theorem provers, leaving the integrated provers able to successfully carry almost all of the formal reasoning burden.

1.3 Contributions

This paper makes the following contributions:

- **Language:** It presents our integrated proof language for proving correctness properties of Java programs. The language includes a rich set of declarative proof constructs that enable developers to direct the reasoning system as little or as much as desired. Because the declarative proof statements are embedded into the program as specialized comments, they also serve as verified documentation and are a natural extension of the assertion mechanism found in most program verification systems.
- **Soundness:** It presents a proof that our integrated proof language is sound.
- **Design:** It presents the rationale behind the design of the integrated proof language. The basic idea is to provide constructs that allow the developer to identify key choice points in the proof search space, then guide the proof by specifying how to resolve these choice points. This approach appropriately leverages the developer's insight to enable the automated reasoning system to successfully prove complex program correctness properties.
- **System:** We have implemented our proof language as an extension to Java. To the best of our knowledge, this is the first system to integrate a sophisticated proof language into an existing programming language with mutable state and object references and to use this proof language to show the correctness of a substantial collection of linked data structures.
- **Results:** We have used our language to prove the correctness of a collection of mutable linked data structure implementations. Our results show that:
  - **Elimination of Interactive Theorem Proving:** Despite the complex and in some cases inherently intractable properties that arise in verifying mutable linked data structure implementations, our integrated proof language eliminates the need to use external interactive theorem provers to establish the correctness of our data structure implementations. Developers instead stay completely within a single unified development and verification environment, with no need to learn a second external system with very different basic concepts, capabilities, and limitations.
  - **Usage Patterns:** In most cases, the identification of several key lemmas is all that is required to enable the automated reasoning system to prove the desired verification conditions. But even though more sophisticated techniques are, in comparison, used less frequently, they are required for proofs of the complex correctness properties that occasionally arise in our set of data structures.
- **Experience:** It discusses our experience using our approach to verify a set of mutable linked data structure implementations. This discussion illuminates the interplay between the various different components of our integrated reasoning system and illustrates how the integrated proof language makes it possible to build on existing automated reasoning systems to verify very sophisticated program correctness properties with tractable developer effort.

1.4 Implications

To date, the vast majority of automated reasoning systems have focused on checking relatively simple consistency properties. As the field matures, the focus will shift to more sophisticated properties involving inherently less tractable formalisms. We believe the basic concepts behind our integrated proof language (extensive use of automated reasoning combined with appropriate developer guidance at key points in the proof search space) will play a prominent role in enabling the continued expansion in the sophistication and utility of the properties that our community is able to verify.

2. Example

We next present an example that illustrates the use of our integrated proof language. Figure 1 presents state declarations and two methods from the ArrayList class.[2] The concrete state of an ArrayList object consists of an array elements, which stores the objects in the ArrayList, and size, which stores the number of objects in elements. The abstract state consists of content, which represents the abstract state of the ArrayList as a set of int, obj pairs; csize, which contains the number of pairs in content; and init, which is false before the list has been initialized and true after. The two vardefs declarations comprise an abstraction function which define the relationship between the abstract and concrete states.[3]

The remove(o) method removes the object o from the array list if it is present. The modifies clause indicates that the method may change the content and csize components of the abstract state. The ensures clause contains the method postcondition. This postcondition identifies two possible cases. In the first case the array list contains o. In this case, the method removes the first occurrence of a pair containing o from content, shifts the remaining pairs down to occupy the gap left by the removal and maintain a dense relation, and returns true. In the second case the array list does not contain o. In this case the method does not change the abstract state and returns false.

The body of the remove method searches for the first occurrence of o in the array. The while loop has an invariant that formalizes the properties that characterize the search. If the loop finds o, it invokes the private helper method shift to shift the elements above o down one position in the array (thereby overwriting o), then returns true. Note that the specification for shift expresses its postcondition in terms of the private concrete state, as opposed to the public abstract state (this is permissible because shift is private).

At this point Jahob must use the postcondition of the shift method and the other facts that it knows after the call to shift to prove the postcondition of the remove method. Unfortunately, the provers are unable to automatically prove the postcondition of remove. In part this is because irrelevant assumptions in verification conditions create a large search space that the provers fail to successfully explore in a reasonable amount of time. What makes the problem even more difficult is that the assumptions contain universally quantified formulas while the postcondition contains an existentially quantified formula. In the absence of developer guidance, the provers must therefore (in this case unsuccessfully) search a large space of possible witnesses for the existentially quantified formula as they attempt to prove the postcondition.

The developer first uses a note statement to instruct Jahob to prove an intermediate assertion (labeled ObjectRemoved in Figure 1). The ObjectRemoved assertion is simpler than the remove postcondition — it serves as a lemma that helps the provers structure the subsequent successful proof of the postcondition. To eliminate irrelevant assumptions, the developer uses the from clause to indicate that the proof of the lemma needs to use only 1) the postcondition of shift, 2) the loop invariant and exit condition from the closest enclosing loop, and 3) the definitions of content and csize. All of these facts are automatically available in the Jahob system and are accessible using standard names. With this guidance, the theorem provers easily establish the ObjectRemoved lemma, which is then available for subsequent reasoning.

```
public class ArrayList
{
 private Object[] elements;
 private int size;
 /*: public specvar init :: bool;
     public specvar content :: "(int * obj) set";
     vardefs "content == {(i,n). 0≤i∧i<size∧n=elements.[i]}"
     public specvar csize :: int;
     vardefs "csize == size" */
 public boolean remove(Object o)
 /*: requires "init"
     modifies content, csize
     ensures
  "(result → (∃ i. (i,o)∈old content ∧
          (¬∃ j. j<i ∧ (j,o)∈old content) ∧
          (∀ j e. 0≤j∧j<i→(j,e)∈content=(j,e)∈old content) ∧
          (i≤j∧j<csize→(j,e)∈content=(j+1,e)∈old content)) ∧
   (¬result → (content=old content∧¬∃i. (i,o)∈old content))" */
 {
  int index = 0;
  while /*: inv "(∀j. 0≤j∧j<index→o≠elements.[j]) ∧
                 0≤index ∧ size=old size" */
        (index < size) {
   if (elements[index] == o) {
    shift(index);
    /*: note ObjectRemoved:
     "∀j e. (0≤j∧j<index→(j,e)∈content=(j,e)∈old content) ∧
        (index≤j∧j<csize→(j,e)∈content=(j+1,e)∈old content)"
          from shift_Postcondition, LoopInv, LoopCondition
                                    content_def, csize_def;
        witness index for
     "∃i. (i,o)∈old content ∧ (¬∃j. j<i ∧ (j,o)∈old content) ∧
       (∀j e. (0≤j∧j<i→(j,e)∈content=(j,e)∈old content) ∧
        (i≤j∧j<csize→(j,e)∈content=(j+1,e)∈old content))" */
    return true;
   }
   index = index + 1;
  }
  return false;
 }
 private void shift(int index)
 /*: requires "init ∧ 0≤index ∧ index<size"
     modifies elements, size, content, csize
     ensures
  "(∀j. 0≤j∧j<index → elements.[j]=old elements.[j])) ∧
   (∀j. index≤j∧j<size → elements.[j]=old elements.[j+1])) ∧
   elements.[size]=null ∧ (size=old size−1)" */
 {...}
}
```

Figure 1. Array List Example

---

[2] The example uses mathematical notation for concepts such as set union (∪) and universal quantification (∀). Developers can enter these symbols in Jahob input files using X-Symbol ASCII notation, and view them in either ASCII or mathematical notation using the ProofGeneral editor mode for emacs [5].

[3] The ArrayList class also contains additional class invariants and methods, which, for clarity, we omit in Figure 1. We also omit certain conjuncts in loop invariants and postconditions. The complete source code for all of our benchmarks, including ArrayList, and the complete source code for Jahob itself are available at javaverification.org.

The developer next uses a witness statement to identify the witness index for the existentially quantified formula in the postcondition. In this way, the developer resolves the witness selection choice point in the proof, which eliminates the need for the provers to search for an appropriate witness. Given the resulting existentially quantified formula and the ObjectRemoved lemma, the provers easily prove the postcondition and all other proof obligations for remove.

As this example shows, the automated provers are very effective at performing the vast majority of the required proof steps. But insight from the developer at key choice points in the proof search space is occasionally required to enable the full proof to go through. In this example, this insight takes the form of a note statement that identifies a key lemma and the facts from which this lemma follows, and a witness statement that identifies a witness for an existentially quantified formula. In general, the developer can express this insight using other proof statements that draw on the developer's understanding of how the program works.

3. Program Verification in Jahob

This section briefly describes the Jahob verification system (for details, see [46]).

Specification Constructs. Following standard practice [9, 16, 46], developers specify Jahob programs using specification variable declarations, method contracts, class invariants, and loop invariants. Specification constructs contain formulas whose syntax and semantics follow Isabelle/HOL [35]. When an annotation containing a formula F occurs at program point q, v in F denotes the value of v at q; old v denotes the value of v at the entry of the currently verified procedure.

Verification Condition Generation. Jahob produces verification conditions by transforming the annotated Java code into extended guarded commands (Figures 2, 3) that contain both code and proof constructs. It then transforms extended guarded commands into simple guarded commands (Figures 4, 6, 8), and then finally generates verification conditions using weakest liberal precondition semantics (Figure 5).

From Java to Guarded Commands. Jahob simplifies code into three-address form to make the evaluation order in expressions explicit and inserts assertions that check for null dereferences, array bounds violations, and type cast errors. It converts field and array assignments into assignments of global variables whose right-hand side contains function update expressions. Having taken side effects into account, it transforms Java expressions into mathematical expressions in higher-order logic.

The Extended Guarded Command Language. Figures 2 and 3 present the syntax of the extended guarded command language. In addition to standard control-flow and state change constructs, the language contains assert, assume, and havoc. These constructs facilitate the staging of verification condition generation into multiple translation steps. They also provide the foundation for the integrated proof language described below in Section 4.

- **Assert:** An assert G annotation at program point q in the body of the method requires the formula G to be true at the program point q. Jahob assertions produce proof obligations that Jahob statically verifies to guarantee that G will be true in all program executions that satisfy the precondition of the method.
- **Assume:** An assume G statement is dual to the assert statement. Whereas an assert requires Jahob to demonstrate that G holds, an assume statement allows Jahob to assume that G is true at a given program point. In general, developer-supplied assume statements may violate soundness. The assume statement is therefore designed to support the automated translation

$$
\begin{array}{rcl}
c & ::= & p \\
& | & \mathsf{skip} \\
& | & x := F \\
& | & c_1 \,[\!]\, c_2 \mid c_1 \,;\, c_2 \\
& | & \mathsf{if}(F)\ c_1\ \mathsf{else}\ c_2 \\
& | & \mathsf{loop\ inv}(I)\ c_1\ \mathsf{while}(F)\ c_2 \\
& | & \mathsf{assume}\ l\!: F \\
& | & \mathsf{havoc}\ \vec{x}\ \mathsf{suchThat}\ F
\end{array}
$$

Figure 2. Extended Guarded Commands

$$
\begin{array}{rcl}
p & ::= & p_1 \,;\, p_2 \\
& | & \mathsf{assert}\ l\!: F\ \mathsf{from}\ \vec{h} \\
& | & \mathsf{note}\ l{:}F\ \mathsf{from}\ \vec{h} \\
& | & \mathsf{localize\ in}\ (p\,;\, \mathsf{note}\ l{:}F) \\
& | & \mathsf{mp}\ l{:}(F \rightarrow G) \\
& | & \mathsf{assuming}\ l_F{:}F\ \mathsf{in}\ (p\,;\,\mathsf{note}\ l_G{:}G) \\
& | & \mathsf{cases}\ \vec{F}\ \mathsf{for}\ l{:}G \\
& | & \mathsf{showedCase}\ i\ \mathsf{of}\ l : F_1 \vee \ldots \vee F_n \\
& | & \mathsf{byContradiction}\ l{:}F\ \mathsf{in}\ p \\
& | & \mathsf{contradiction}\ l{:}F \\
& | & \mathsf{instantiate}\ l{:}\forall \vec{x}.F\ \mathsf{with}\ \vec{t} \\
& | & \mathsf{witness}\ \vec{t}\ \mathsf{for}\ l{:}\exists \vec{x}.F \\
& | & \mathsf{pickWitness}\ \vec{x}\ \mathsf{for}\ l_F{:}F\ \mathsf{in}\ (p\,;\,\mathsf{note}\ l_G{:}G) \\
& | & \mathsf{pickAny}\ \vec{x}\ \mathsf{in}\ (p\,;\,\mathsf{note}\ l{:}F) \\
& | & \mathsf{induct}\ l{:}F\ \mathsf{over}\ n\ \mathsf{in}\ p
\end{array}
$$

Figure 3. Integrated Proof Language Constructs

$$
\begin{array}{rcl}
c & ::= & \mathsf{assume}\ l\!: F \\
& | & \mathsf{assert}\ l\!: F\ \mathsf{from}\ \vec{h} \\
& | & \mathsf{havoc}\ \vec{x} \\
& | & \mathsf{skip} \\
& | & c_1 \,[\!]\, c_2 \mid c_1 \,;\, c_2
\end{array}
$$

Figure 4. Simple Guarded Commands

$$
\begin{array}{rcl}
\mathsf{wlp}((\mathsf{assume}\ l\!: F), G) & = & F^{[l]} \rightarrow G \\
\mathsf{wlp}((\mathsf{assert}\ l\!: F\ \mathsf{from}\ \vec{h}), G) & = & F^{[l;\vec{h}]} \wedge G \\
\mathsf{wlp}((\mathsf{havoc}\ \vec{x}), G) & = & \forall \vec{x}.\, G \\
\mathsf{wlp}((\mathsf{skip}), G) & = & G \\
\mathsf{wlp}((c_1 \,[\!]\, c_2), G) & = & \mathsf{wlp}(c_1, G) \wedge \mathsf{wlp}(c_2, G) \\
\mathsf{wlp}((c_1 \,;\, c_2), G) & = & \mathsf{wlp}(c_1, \mathsf{wlp}(c_2, G))
\end{array}
$$

Figure 5. Weakest Preconditions for Simple Guarded Commands

$$
\begin{array}{l}
\text{for } v \text{ fresh variable, } [\![x := F]\!] = \mathsf{havoc}\ v\,;\, \mathsf{assume}\ (v = F)\,; \\
\qquad\qquad\qquad\qquad\qquad\quad \mathsf{havoc}\ x\,;\, \mathsf{assume}\ (x = v) \\
\qquad\qquad [\![\mathsf{if}(F)\ c_1\ \mathsf{else}\ c_2]\!] = (\mathsf{assume}\ F\,;\, [\![c_1]\!])\ [\!] \\
\qquad\qquad\qquad\qquad\qquad\quad (\mathsf{assume}\ \neg F\,;\, [\![c_2]\!]) \\
[\![\mathsf{loop\ inv}(I)\ c_1\ \mathsf{while}(F)\ c_2]\!] = \\
\text{(where } \vec{r} = \mathsf{mod}(c_1; c_2) \text{ denotes variables modified in } c_1, c_2) \\
\qquad\qquad\qquad \mathsf{assert}\ I\,;\, \mathsf{havoc}\ \vec{r}\,;\, \mathsf{assume}\ I\,; \\
\qquad\qquad\qquad [\![c_1]\!]\,; \\
\qquad\qquad\qquad (\mathsf{assume}\ (\neg F)\ [\!]\, (\mathsf{assume}\ F\,; \\
\qquad\qquad\qquad\qquad\quad [\![c_2]\!]\,;\, \mathsf{assert}\ I\,; \\
\qquad\qquad\qquad\qquad\quad \mathsf{assume\ false})) \\
\qquad [\![\mathsf{havoc}\ \vec{x}\ \mathsf{suchThat}\ F]\!] = \mathsf{assert}\ \exists \vec{x}.F\,; \\
\qquad\qquad\qquad\qquad\qquad\quad \mathsf{havoc}\ \vec{x}\,;\, \mathsf{assume}\ F
\end{array}
$$

Figure 6. Translating Code into Simple Guarded Commands

of higher-level constructs into a lower-level intermediate language, with soundness ensured by the form of the translation.

- **Non-deterministic change:** A statement $\mathsf{havoc}\, x\ \mathsf{suchThat}\ G$, where x is a variable and G is a formula, changes the value of x subject only to the constraint G (for example, the statement $\mathsf{havoc}\, x\ \mathsf{suchThat}\ 0 \leq x$ sets x to an arbitrary non-negative value). To ensure that the statement does not have the effect of assume(false), Jahob emits an assertion that verifies that at least one such value of x exists.

From Code to Simple Commands. Figure 6 presents the translation of the code portion of the extended guarded command language into simple guarded commands. Jahob translates assignments into a series of havoc statements and equality constraints, which reduces all state changes to havoc statements. Conditional statements become non-deterministic choice with assume statements, as in control-flow graph representations. The Jahob encoding of loops with loop invariants is standard and analogous, for example, to the sound version of the encoding in ESC/Java [20].

Proving Verification Conditions. Verification conditions generated using the rules in Figure 5 can typically be represented as a conjunction of a large number of formulas. Figure 7 describes Jahob's splitting process, which produces a list of implications whose conjunction is equivalent to the original formula. The individual implications correspond to different paths in the method, as well as different conjuncts of assert statements, operation preconditions, invariants, postconditions, and preconditions of invoked methods. The splitting rules in Jahob preserve formula annotations, which are used for assumption selection. During splitting Jahob also eliminates simple syntactically valid implications, such as those whose goal occurs as one of the assumptions, or those whose assumptions contain false.

4. The Integrated Proof Language

Our integrated proof language is designed to allow developers to provide additional guidance to the system to enable the automated provers to succeed in proving the desired verification conditions. Figure 3 presents the constructs in this language. These constructs appear in comments embedded within the Java source code, and are preserved by the translation to the extended guarded command language. Figure 8 presents the semantics of the proof language constructs as a translation into the simple guarded command language.

A primary goal of the design of the integrated proof language is to enable the developer to provide the system with as little or as much guidance as desired. At one extreme the provers should be able to prove verification conditions with no developer guidance at all if they have this capability. At the other extreme the language should enable the developer to perform every proof step explicitly if so desired. The language should also flexibly support intermediate points at which the developer and provers cooperate, with the developer providing only the minimal guidance needed to enable the provers to complete the proof. The Jahob proof language supports this wide range of behaviors by providing not only high-level constructs that leverage the substantial automated reasoning power of the Jahob system, but also low-level constructs that allow the developer to precisely control proof steps to enable a successful proof.

4.1 The Assumption Base

A verification condition in Jahob has the form of an implication $F \rightarrow G$ where the antecedent F is a conjunction of facts. This conjunction F is the *assumption base* that the provers use when they attempt to prove the consequent G.

$$
\begin{array}{rcl}
\vec{A} \rightarrow G_1 \wedge G_2 & \rightsquigarrow & \vec{A} \rightarrow G_1,\ \vec{A} \rightarrow G_2 \\
\vec{A} \rightarrow (\vec{B} \rightarrow G^{[p]})^{[q]} & \rightsquigarrow & (\vec{A} \wedge \vec{B}^{[q]}) \rightarrow G^{[pq]} \\
\vec{A} \rightarrow \forall x.G & \rightsquigarrow & \vec{A} \rightarrow G[x := x_{\mathsf{fresh}}]
\end{array}
$$

Figure 7. Splitting Rules for Converting a Formula into an Implication List ($F^{[c]}$ denotes a formula F annotated with a string c)

The translations of the proof language constructs use assume commands to add facts to the assumption base.[4] The soundness of the assume commands in this context is guaranteed by the form of the translation.

Some translations contain the following pattern:

$$(\mathsf{skip} \mathbin{[\!]} (c\,;[\![p]\!]\,;\mathsf{assert}\ F\,;\mathsf{assume\ false}))\,;\mathsf{assume}\ G$$

The net effect of this pattern is to soundly add G to the original assumption base. The pattern achieves this effect as follows. The first branch of the non-deterministic choice operator (skip) propagates the original assumption base. The second branch $(c\,;[\![p]\!]\,;\mathsf{assert}\ F\,;\mathsf{assume\ false})$ generates the proof obligations required to ensure that G actually holds. The assume false at the end of the second branch conceptually terminates the computation at the end of the branch so that the verification condition generator does not take the computational path through the second branch into account when generating the verification condition at the program point after the choice. This mechanism ensures that the second branch generates no proof obligations other than those required to ensure that G holds.

In effect, the second branch uses the assume false statement to create a new local assumption base in which the developer can guide the proof of the properties required to ensure that G holds. Because this assumption base is local, none of the assumptions or intermediate lemmas in the proof propagate through to the program point after the choice. This local assumption base mechanism therefore ensures that only G is added to the original assumption base at the program point after the translated proof language construct, and that local assumptions that are only sound in the context of the proof are not propagated to the original assumption base.

The command c contains statements introduced as part of the translation; p contains proof statements provided by the developer and originally nested inside the proof construct under translation. The command c can include constructs that may modify the program state, such as assume and havoc constructs. The form of the translation ensures that these constructs are used in a sound way.

4.2 The note Construct

The note construct translates into an assert followed by an assume of the same formula. The net effect is to identify a formula for Jahob to prove, then add the verified formula to the assumption base. Because Jahob proves the formula before adding it to the assumption base, the use of note is sound. A note statement of the form $\mathsf{note}\ l\!:\!F\ \mathsf{from}\ \vec{h}$ assigns a name l to the formula F and asks Jahob to prove F using the named formulas $\vec{h}$.

Proof Decomposition. The note statement enables the developer to guide the decomposition of the proof by instructing the combined proof system to prove certain lemmas. The availability of these lemmas is often sufficient to guide the provers through the (usually unbounded) proof search space to successfully find a proof for the verification condition of interest.

[4] Specifically, the verification condition generation rule in Figure 5 for statements of the form $\mathsf{assume}\ l\!:F$ produces a verification condition of the form $F^{[l]} \rightarrow G$, which, in effect, adds F to the set of facts available to the provers when they attempt to prove the consequent G.

$$\llbracket \mathsf{note}\ l{:}F\ \mathsf{from}\ \vec{h} \rrbracket = \mathsf{assert}\ l{:}\ F\ \mathsf{from}\ \vec{h}\ \mathbf{;}\ \mathsf{assume}\ l{:}\ F$$

$$\llbracket \mathsf{localize\ in}\ (p\,\mathbf{;}\,\mathsf{note}\ l{:}F) \rrbracket = (\mathsf{skip}\ [\!]\ (\llbracket p \rrbracket\,\mathbf{;}\,\mathsf{assert}\ F\,\mathbf{;}\ \mathsf{assume\ false}))\,\mathbf{;}\ \mathsf{assume}\ l{:}\ F$$

$$\llbracket \mathsf{mp}\ l{:}(F \rightarrow G) \rrbracket = \mathsf{assert}\ F\,\mathbf{;}\,\mathsf{assert}\ (F \rightarrow G)\,\mathbf{;}\ \mathsf{assume}\ l{:}\ G$$

$$\llbracket \mathsf{assuming}\ l_F{:}F\ \mathsf{in}\ (p\,\mathbf{;}\,\mathsf{note}\ l_G{:}G) \rrbracket = (\mathsf{skip}\ [\!]\ (\mathsf{assume}\ l_F{:}\ F\,\mathbf{;}\ \llbracket p \rrbracket\,\mathbf{;}\,\mathsf{assert}\ G\,\mathbf{;}\ \mathsf{assume\ false}))\,\mathbf{;}\ \mathsf{assume}\ l_G{:}\ (F \rightarrow G)$$

$$\llbracket \mathsf{cases}\ \vec{F}\ \mathsf{for}\ l{:}G \rrbracket = \mathsf{assert}\ F_1 \vee \ldots \vee F_n\,\mathbf{;}\ \mathsf{assert}\ (F_1 \rightarrow G)\,\mathbf{;}\ldots\mathbf{;}\,\mathsf{assert}\ (F_n \rightarrow G)\,\mathbf{;}\ \mathsf{assume}\ l{:}\ G$$

$$\llbracket \mathsf{showedCase}\ i\ \mathsf{of}\ l : F_1 \vee \ldots \vee F_n \rrbracket = \mathsf{assert}\ F_i\,\mathbf{;}\ \mathsf{assume}\ l{:}\ F_1 \vee \ldots \vee F_n$$

$$\llbracket \mathsf{byContradiction}\ l{:}F\ \mathsf{in}\ p \rrbracket = (\mathsf{skip}\ [\!]\ (\mathsf{assume}\ \neg F\,\mathbf{;}\ \llbracket p \rrbracket\,\mathbf{;}\,\mathsf{assert\ false}\,\mathbf{;}\ \mathsf{assume\ false}))\,\mathbf{;}\ \mathsf{assume}\ l{:}\ F$$

$$\llbracket \mathsf{contradiction}\ l{:}F \rrbracket = \mathsf{assert}\ F\,\mathbf{;}\,\mathsf{assert}\ \neg F\,\mathbf{;}\,\mathsf{assume\ false}$$

$$\llbracket \mathsf{instantiate}\ l{:}\forall \vec{x}.F\ \mathsf{with}\ \vec{t} \rrbracket = \mathsf{assert}\ \forall \vec{x}.F\,\mathbf{;}\ \mathsf{assume}\ l{:}\ F[\vec{x} := \vec{t}]$$

$$\llbracket \mathsf{witness}\ \vec{t}\ \mathsf{for}\ l{:}\exists \vec{x}.F \rrbracket = \mathsf{assert}\ F[\vec{x} := \vec{t}]\,\mathbf{;}\ \mathsf{assume}\ l{:}\ \exists \vec{x}.F$$

$$\llbracket \mathsf{pickWitness}\ \vec{x}\ \mathsf{for}\ l_F{:}F\ \mathsf{in}\ (p\,\mathbf{;}\,\mathsf{note}\ l_G{:}G) \rrbracket = \text{(where } \vec{x} \text{ is not free in } G\text{)}\ (\mathsf{skip}\ [\!]\ (\mathsf{assert}\ \exists \vec{x}.F\,\mathbf{;}\ \mathsf{havoc}\ \vec{x}\,\mathbf{;}\ \mathsf{assume}\ l_F{:}\ F\,\mathbf{;}\ \llbracket p \rrbracket\,\mathbf{;}\,\mathsf{assert}\ G\,\mathbf{;}\ \mathsf{assume\ false}))\,\mathbf{;}\ \mathsf{assume}\ l_G{:}\ G$$

$$\llbracket \mathsf{pickAny}\ \vec{x}\ \mathsf{in}\ (p\,\mathbf{;}\,\mathsf{note}\ l{:}G) \rrbracket = (\mathsf{skip}\ [\!]\ (\mathsf{havoc}\ \vec{x}\,\mathbf{;}\ \llbracket p \rrbracket\,\mathbf{;}\,\mathsf{assert}\ G\,\mathbf{;}\ \mathsf{assume\ false}))\,\mathbf{;}\ \mathsf{assume}\ l{:}\ \forall \vec{x}.G$$

$$\llbracket \mathsf{induct}\ l{:}F\ \mathsf{over}\ n\ \mathsf{in}\ p \rrbracket = (\mathsf{skip}\ [\!]\ (\mathsf{havoc}\ n\,\mathbf{;}\ \mathsf{assume}\ 0 \leq n\,\mathbf{;}\ \llbracket p \rrbracket\,\mathbf{;}\,\mathsf{assert}\ F[n := 0]\,\mathbf{;}\ \mathsf{assert}\ (F \rightarrow F[n := n{+}1])\ \mathsf{assume\ false}))\,\mathbf{;}\ \mathsf{assume}\ l{:}\ \forall n.(0 \leq n \rightarrow F)$$

Figure 8. Translating Proof Language Constructs into Simple Guarded Commands

Multiple Provers. The note statement also enables developers to decompose a proof obligation so that multiple provers (with arbitrarily narrow areas of specialization) can work together to prove it. Consider, for example, a proof obligation that involves both arithmetic reasoning and reasoning about the shape of a given data structure. By using one group of note statements to identify relevant arithmetic properties and another group of note statements to identify relevant data structure shape properties, the developer can decompose the proof obligation to expose specific parts of the proof obligation to different provers. A final note statement can then combine the results to deliver the complete proof obligation. A potential advantage of this approach is that the set of provers, when working together, may be able to provide sophisticated reasoning capabilities that are beyond the reach of any single general system.

Controlling the Assumption Base. Many provers perform a search over a (potentially unbounded) proof space. In practice, we have found that increasing the size of the assumption base may degrade the ability of the prover to find proofs for facts that it is otherwise perfectly capable of proving. Note statements allow developers to give names to specific facts (these facts can either be available directly in the assumption base or provable from the assumption base), then use the names to identify a specific set of facts that the prover should use when attempting to prove a new fact. The net effect is to eliminate irrelevant facts from the assumption base to productively focus provers on the specific facts they need to use.

4.3 The localize Construct

The localize construct allows the developer to create a new local assumption base for the proof of an arbitrary formula, then add only this proved formula back into the original assumption base. The construct therefore makes it possible to use intermediate lemmas to guide the proof of a formula without adding these intermediate lemmas back into the original assumption base. Excluding intermediate lemmas from the original assumption base when the lemmas are not relevant for subsequent verification conditions can help keep the assumption base (and resulting proof search space) small enough to enable the provers to find proofs of subsequent verification conditions in an acceptable amount of time. Note that because the local assumption base is initially the same as the original assumption base, any formulas verified in the local assumption base also hold in the original assumption base. This property ensures that the construct is sound.

$$\frac{F \quad G}{F \wedge G} \qquad \frac{F \wedge G}{F} \qquad \frac{F \wedge G}{G} \qquad \frac{\mathsf{false}}{F} \qquad \frac{}{\mathsf{true}}$$

Figure 9. Implicit First-Order Logic Rules

4.4 First-Order Logic Constructs

The first-order logic proof constructs encode standard rules of natural deduction systems [38, Section 2.12], [21, Section 5.4]. When combined with the proof rules implicit in the splitting process, these constructs give our system the completeness of first-order logic. (Note that for arbitrary higher-order logic formulas we cannot hope to obtain a proof system complete with respect to the standard models [2].) During splitting, Jahob implicitly splits top-level conjunctions of assumptions and goals, eliminating the need for conjunction introduction and elimination commands. Jahob also implicitly incorporates the standard rule for deriving any formula when false is one of the assumptions. Figure 9 shows the rules that Jahob automatically applies as part of the splitting process.

The assuming construct encodes the implication introduction rule for first-order logic. It enables the developer to guide the proof of a fact of the form $F \rightarrow G$. The fact F is first added to a new local assumption base, in which the developer can guide the proof of G. $F \rightarrow G$ is then added back into the original assumption base.

The assuming construct makes it possible to decompose the components of an implication. Without such a construct, many intermediate lemmas in a proof of $F \rightarrow G$ would themselves be implications that could not be decomposed, increasing the difficulty of the proof task. In practice, we have found the assuming construct particularly useful when G becomes complex, because the provers often fail to find the proof in such cases without guidance.

The mp construct encodes the *modus ponens* rule of inference. It enables the developer to conclude a goal G from known facts of the form F and $F \rightarrow G$.

The pickAny construct encodes the universal introduction rule. It enables the developer to guide the proof of a fact of the form $\forall \vec{x}.G$ by choosing arbitrary values for $\vec{x}$ in a new local assumption base, then allowing the developer to guide the proof of G. $\forall \vec{x}.G$ is then added to the original assumption base.

The instantiate construct encodes the universal elimination rule. It enables the developer to establish a fact of the form $G[\vec{x} := \vec{t}]$ by guiding the proof of a fact of the form $\forall \vec{x}.G$, then adding $G[\vec{x} := \vec{t}]$ to the assumption base.

The witness construct encodes the existential introduction rule. It enables the developer to establish a fact of the form $\exists \vec{x}.G$ by guiding the proof of a fact of the form $G[\vec{x} := \vec{t}]$, then adding $\exists \vec{x}.G$ to the assumption base.

The pickWitness construct encodes the existential elimination rule. It enables the developer to instantiate a formula of the form $\exists \vec{x}.F$ (i.e., eliminate the existential quantifier and name the values that satisfy the constraint F) in a new local assumption base, guide the proof of a formula G, then add the proved goal G back into the original assumption base. To ensure soundness, $\vec{x}$ — the variable(s) with which the developer is instantiating $\exists \vec{x}.F$ — must not be free in G.

By enabling the developer to name values of $\vec{x}$ for which the constraint F is true, the pickWitness construct makes it possible to replace an existentially-quantified formula with an instantiated version, then state additional facts about the named values. This functionality broadens the applicability of provers with limited ability to reason about existentially quantified formulas. Without the pickWitness construct, every subgoal that depended on the constrained values would have the form $\exists \vec{x}.(F \rightarrow G)$. Such a subgoal is beyond the reach of any prover that cannot reason effectively about existentially quantified formulas. The pickWitness construct enables the developer to soundly eliminate the existential quantifier, thereby transforming existentially quantified proof goals into a form that such provers can more effectively handle.

The cases construct enables the developer to decompose a goal using case analysis. It ensures that the set of cases fully cover the space of the proof, proves each case, then soundly concludes the goal.

The showedCase construct encodes the disjunction introduction rule. It enables the developer to establish a fact of the form $F_1 \vee \ldots \vee F_n$ by guiding the proof of a case F_i in the disjunction.

The byContradiction construct enables the developer to prove an arbitrary formula F using proof by contradiction. It allows the developer to add $\neg F$ to a new local assumption base, then use this assumption base to guide the proof of false. The verified formula F can then be soundly added to the original assumption base. The developer can also use this construct to perform negation introduction by directing Jahob to prove a formula of the form $\neg F$.

The contradiction construct enables the developer to derive false from a contradiction. It allows the developer to guide the proof of a formula F and its negation $\neg F$ to soundly conclude false.

4.5 The induct Construct

The induct construct enables the developer to prove facts of the form $\forall n.(0 \leq n \rightarrow F)$ using mathematical induction. As the translation in Figure 8 illustrates, the induct statement encodes mathematical induction by choosing an arbitrary value of n such that $0 \leq n$ holds, then allowing the developer to guide the proof of the base case $F[n := 0]$ and the inductive step $F \rightarrow F[n := n+1]$. The introduction of the constraint $0 \leq n$ makes it possible to simulate mathematical induction over natural numbers using integers.

The induct construct is particularly important because the fully automated provers that we use are generally unable to derive facts whose proofs require mathematical induction. Without this construct, the only recourse for the developer would be to perform the proof using an external interactive theorem prover.

4.6 Executable Code Inside Proof Constructs

Many proof language constructs can (recursively) contain other proof language constructs (see Figure 3). It is possible to generalize this formulation and introduce sound constructs that contain not only proof constructs, but also executable code that may mutate the Java program state. In particular, it is possible to generalize pickWitness and pickAny to enclose the general extended guarded command c (which may contain executable Java code as well as proof constructs) instead of just the proof command p. The generalization of pickWitness makes it possible to choose a witness for an existentially quantified formula in the assumption base, then use that witness at multiple program points throughout a sequence of Java statements. The dual generalization of pickAny makes it possible to prove a universally quantified formula at the end of a sequence of Java statements by 1) introducing a fresh variable that denotes an arbitrary value at the start of the sequence, 2) using this new variable to refer to its value at multiple program points throughout the sequence of statements, then 3) concluding the universally quantified formula at the end of the sequence. This approach is particularly useful in simplifying proofs of program properties that would otherwise require the reasoning systems to work with universally quantified loop invariants.

To enable the flexible combination of Java code and proof commands, we propose a proof language construct, fix, that subsumes both pickWitness and pickAny and can enclose executable code that may mutate the Java program state. Appendix B presents the fix construct, including its syntax, its semantics through a translation into simple guarded commands, and a proof of its soundness.

5. Soundness

The translation rules in Figure 8 define the semantics of our proof language constructs. We show the soundness of each of these rules using properties of weakest liberal preconditions [6]. We define the relation $\sqsubseteq$ such that $c \sqsubseteq c'$ if and only if $\mathsf{wlp}(c, F) \rightarrow \mathsf{wlp}(c', F)$ for all formulas F. In this case we say that c is stronger than c'.

Let skip be the no-op command. We use induction on p to show $p \sqsubseteq \mathsf{skip}$ for all p. This is sufficient for soundness because it ensures that any property provable for the annotated program containing proof language constructs also holds in the unannotated program (which is equivalent to the annotated program with all proof constructs replaced with skip).

The induction hypothesis is $\mathsf{wlp}([\![p]\!], H) \rightarrow H$, where H is an arbitrary formula. For each proof language construct p, we apply the translation rules in Figure 8, the rules of weakest liberal preconditions in Figure 5, the induction hypothesis, and the standard rules of logic to show that $\mathsf{wlp}([\![p]\!], H) \rightarrow H$ — i.e., that p is stronger than skip.

As a sample inductive step, consider the assuming construct. By applying the translation rule for assuming, the rules of weakest liberal preconditions, and the standard rules of logic, we obtain:

$$
\begin{array}{rl}
 & \mathsf{wlp}([\![\mathsf{assuming}\ F\ \mathsf{in}\ (p\,\mathbf{;}\,\mathsf{note}\ G)]\!], H) \\
= & \mathsf{wlp}(((\mathsf{skip}\ [\!]\ (\mathsf{assume}\ F\,\mathbf{;}[\![p]\!]\,\mathbf{;}\,\mathsf{assert}\ G\,\mathbf{;}\,\mathsf{assume\ false}))\,\mathbf{;} \\
 & \quad \mathsf{assume}\ (F \rightarrow G)), H) \\
= & ((F \rightarrow G) \rightarrow H) \wedge (F \rightarrow \mathsf{wlp}([\![p]\!], G))
\end{array}
$$

According to the induction hypothesis, $\mathsf{wlp}([\![p]\!], G) \rightarrow G$. Therefore, $((F \rightarrow G) \rightarrow H) \wedge (F \rightarrow \mathsf{wlp}([\![p]\!], G))$ implies the formula $((F \rightarrow G) \rightarrow H) \wedge (F \rightarrow G)$, which implies H. Consequently, assuming is stronger than skip and its translation is sound. The proofs for the other constructs, including induct, pickAny, and pickWitness, are similar and are given in Appendix A.

Data Structure	Java Methods	Java Statements	Verification Time (s)	Specification Variables	Local Specification Variables	Data Structure Invariants	Loop Invariants	note Statements	localize Statements
Hash Table	15	90	497.6	5	3	20	4	176 (93)	12
Priority Queue	12	60	130.2	5	0	9	2	105 (47)	0
Binary Tree	9	134	6519.9	2	4	7	6	98 (15)	2
Array List	23	121	194.5	4	0	10	10	27 (11)	0
Circular List	5	57	119.2	4	1	9	1	3 (0)	0
Cursor List	9	51	36.2	6	0	16	1	2 (0)	0
Association List	11	65	10.8	3	1	7	3	0 (0)	0
Linked List	6	38	5.3	3	0	8	2	0 (0)	0

Data Structure	assuming Statements	mp Statements	pickAny Statements	instantiate Statements	witness Statements	pickWitness Statements	cases Statements	induct Statements
Hash Table	26	3	17	8	0	0	3	0
Priority Queue	28	0	11	1	1	4	2	2
Binary Tree	0	0	0	0	0	0	0	0
Array List	3	0	1	0	1	0	0	0
Circular List	0	0	0	0	0	0	0	0
Cursor List	0	0	0	0	0	0	0	0
Association List	0	0	0	0	0	0	0	0
Linked List	0	0	0	0	0	0	0	0

Table 1. Method, Statement, Specification, and Integrated Proof Language Construct Counts for Verified Data Structures

6. Experimental Results

We next discuss our experience using our integrated proof language in the specification and verification of a collection of linked data structures. The complete source code for the data structures (including implementations and specifications) as well as the Jahob verification system (including source code) are all available at javaverification.org.

6.1 Construct Counts and Verification Times

Table 1 presents the verification times for the data structures as well as counts of various Java and Jahob constructs. The first and second columns present the number of Java methods and Java statements, respectively, in the data structure implementations. The third column presents the time required for Jahob to verify each implementation.

The remaining columns present the counts of the different Jahob constructs. Each method's specification typically contains requires, modifies, and ensures clauses, although some requires and modifies clauses are empty and therefore omitted from the specification. The remaining columns present counts of the various specification and proof language constructs, including the number of specification variables, local specification variables, data structure invariants, loop invariants, and proof statements. There is one column for each type of proof statement used in our data structures.

The note Statements column contains entries of the form $n(m)$. In these entries n counts the total number of note statements that appear in the data structure implementation. Of these n statements, m have a from clause that is used to identify a set of named facts for the provers to use when proving the new fact in the note statement. Because the typical motivation for including a from clause is to limit the size of the assumption base so that the provers can prove the new fact in a reasonable amount of time, these numbers provide some indication of how sensitive the provers are to the size of the assumption base in each data structure.

In general, the data structures use note statements much more extensively than any other proof language construct. This fact reflects the strength of the underlying provers — it is often possible to guide the provers to an effective proof by either providing a few lemmas that effectively guide the proof decomposition or by appropriately limiting the assumption base.

	Without Proof Language Constructs		With Proof Language Constructs	
Data Structures	Methods Verified	Sequents Verified	Methods Verified	Sequents Verified
Hash Table	6 of 15	949 of 982	15	1226
Priority Queue	9 of 12	555 of 563	12	792
Binary Tree	1 of 9	776 of 866	9	1294
Array List	18 of 23	886 of 891	23	928
Circular List	2 of 5	212 of 226	5	237
Cursor List	8 of 9	353 of 354	9	356
Association List	11 of 11	349 of 349	11	349
Linked List	6 of 6	168 of 168	6	168

Table 2. Effect of Proof Language Constructs on Verification

6.2 Effect of Proof Language Constructs

Table 2 presents numbers that summarize the effect that the proof language constructs have on the verification. The first two columns present the number of methods and sequents verified without proof language constructs. We obtained these numbers by removing all proof statements from the program, then attempting to verify the data structure. Each prover runs with a timeout — if the prover fails to prove the sequent within the timeout, Jahob terminates it and moves on to the next prover. In general, the more complex the data structure, the more guidance the provers need to verify the data structure.

The final column in Table 2 presents the total number of sequents required to fully verify the corresponding data structure implementations after adding the necessary proof language statements. Note that the number of sequents increases, in some cases significantly. This is because the proof statements force the provers to prove additional lemmas, which in turn correspond to additional sequents. The increase in the number of sequents reflects the difficulty of proving the complex sequents that failed to verify in the absence of developer guidance.

We next discuss the use of the proof language constructs in the Hash Table, Priority Queue, Binary Tree, and Array List data structures in more detail.

6.3 Hash Table

Hash Table implements a relation between keys and values. It uses the standard array of linked lists implementation, with each element in each list storing one of the key, value pairs in the relation.

The data structure uses note statements primarily for two purposes: to control the assumption base, and to instruct the provers to prove key lemmas involving the relationship between the concrete and abstract states. For example, the implementation often performs an operation on the linked list to which a specific key hashes. In this case a note statement often instructs the provers to prove a formula stating that if the abstract relation contains a specific key, value pair, then the element storing that pair is in the list stored at the offset in the array given by the key's hash value. Other note statements involve data structure consistency properties — for example, that once a given key, value pair has been removed from the hash table, there is no list element in the table whose next pointer refers to an element with that same key, value pair.

In comparison with the other data structures in our benchmark set, the hash table is a fairly complex data structure with many representation invariants. Even though the provers are capable of proving many of the desired properties, the presence of all of the invariants produces an assumption base large enough to significantly impair the ability of the provers to find the proofs within a reasonable amount of time. Many of the note statements are therefore present primarily to restrict the assumption base to relevant facts, thereby enabling the provers to focus their efforts on a productive part of the search space so that they can successfully prove the verification conditions within a reasonable amount of time.

In addition to the note statement, the hash table also uses the localize, assuming, mp, pickAny, instantiate, and cases statements. The hash table uses localize statements to limit the scope of intermediate lemmas, since adding extraneous facts to the assumption base can degrade the effectiveness of the provers in proving subsequent verification conditions. Note that localize statements also make certain aspects of the proof structure (specifically, the correspondence between intermediate lemmas and verification conditions) explicit. They therefore make the proofs easier to understand.

The hash table uses assuming and pickAny statements primarily in the proof of data structure invariants, which typically have the general form $\forall x.(x \in C \rightarrow P)$ (for all objects x that belong to the class C, the property P holds). As a result, the proof of a data structure invariant often has the following general form:

pickAny x in
  (assuming $x \in C$ in
    $(p$; note $P)$;
  note $P)$

The hash table uses instantiate statements to appropriately instantiate the universal quantifiers in data structure invariants to obtain intermediate lemmas to prove other goals.

6.4 Priority Queue

Priority Queue implements a priority queue with a set interface. The queue itself is implemented as a complete binary tree stored in a dense array. The children of a parent element stored at index i are stored at indices $2i+1$ and $2i+2$. An important ordering invariant is that each parent's key is greater than the keys of its two children. The greatest element is therefore the root of the tree.

The majority of the note statements appear in methods that update the tree, either to insert a new element or to remove the greatest element. During these methods some of the invariants are temporarily violated as the tree is updated. Many of the note statements appear in groups that identify these regions. Conceptually, the purpose of these statements is often to identify the updated region, instruct the provers to prove lemmas stating that the invariants hold outside of the updated region, and identify the properties that characterize the updated region. Given this guidance, the provers are then able to prove that a completed update restores the invariants to implement the desired operation.

The priority queue uses the induct statement to prove that the element stored at index 0 of the array is the maximal element in the tree. The provers are not able to prove this property without assistance, but by using the induct statement as well as other proof statements to resolve key choice points in the intermediate steps of the proof, it is possible to establish this property from the ordering invariant. Of the remaining constructs, the priority queue uses the assuming and pickAny statements the most, to establish data structure invariants and to establish equality between sets using proofs of the following form:

pickAny x in
  (assuming $x \in A$ in
    $(p_f$; note $x \in B)$;
  note l_f:$x \in B)$;
pickAny x in
  (assuming $x \in B$ in
    $(p_b$; note $x \in A)$;
  note l_b:$x \in A)$;
note l:$A = B$ from l_f, l_b

The pickAny and assuming statements establish that $\forall x.x \in A \rightarrow x \in B$ and $\forall x.x \in B \rightarrow x \in A$. The final note statement establishes from these two facts that the sets A and B are equal.

6.5 Binary Tree

Binary Tree stores a set of elements in a binary search tree. It exports a set interface to the elements in the tree. It is a challenging data structure to verify because the verification conditions involve a wide range of different kinds of properties: data structure shape properties, ordering properties involving the elements in the tree, and abstraction properties that relate the tree to the abstract set it implements. Moreover, the interactions between these properties make it difficult for any single prover to prove the generated verification conditions by itself.

The binary tree uses primarily note statements. The vast majority of the note statements in the data structure implementation are present to facilitate the interaction between the different provers that work together to verify the implementation. Specifically, these note statements identify specific shape properties that the Mona decision procedure is then able to prove. The first-order theorem provers use these shape properties as lemmas to establish the relationships between the shape properties, ordering properties, and abstraction properties required to prove the verification conditions. In effect, the note statements serve a dual purpose: guiding the provers to an effective proof decomposition by instructing them to prove data structure shape lemmas and enabling the successful application of multiple provers to a single verification problem by identifying relevant facts for the specialized Mona decision procedure to prove.

6.6 Array List

Array List implements a mapping from integers to objects. It stores the mapping in an array; the position of the element in the array corresponds to the element's index in the mapping. The note statements are often used to instruct the provers to prove lemmas that relate the contents of regions in the updated array to the contents of corresponding regions in the original array. For example, the method that adds an element at a specific index copies the block of items above the index up one position to make space for the new element.

The array list also contains a pickAny statement and several assuming statements. These are used to guide the proofs of several universally quantified lemmas that relate membership in different versions of the array with restrictions on the corresponding index of the element. For example, one of the lemmas states that if the element was in the array prior to the insertion of a new element, its index was less than the active size of the array. These formulas are used to prove a set equality relationship involving the old content before an insertion and the new content after the insertion.

As discussed above in Section 2, the array list also contains a witness statement that enables the developer to identify the witness for one of the clauses of the postcondition of the remove method.

7. Related Work

Interactive theorem provers include Isabelle/HOL [35], PVS [37], Boyer-Moore provers [11], and Coq [10]. Most of these provers provide facilities for exporting executable definitions of mathematical functions into purely functional programs. It is natural to consider combinations of automated techniques to increase the granularity of interactive proof steps in interactive provers. Such integration is used in PVS [37], Boyer-Moore provers [11], and Isabelle [33]. We adopt declarative-style proofs, which are also supported by the Isabelle's Isar notation [44] and are present in the Mizar [40] and Athena [4] systems. We adopted the names (but not the exact semantics) of terms such as pickWitness and assumption base from the Athena system [3, 4].

The difference between the approach used in interactive theorem provers and the approach used in our proof language is that the former aims to incorporate executable programs into the universe of proofs, while our proof language aims to bring proofs into the universe of executable programs. In particular, our proof language is able to naturally embed proofs into imperative programs. Most interactive theorem provers operate over functional programming languages in an environment designed for proofs. Our proof language is integrated into the underlying imperative programming language and naturally extends its assertion mechanism. This approach provides the developer with an accessible way of reasoning not only about the effect of executing a method but also about the intermediate states during the execution, which is often necessary when verifying complex program properties.

Although Jahob supports the use of interactive provers, its proof commands provide an alternative way of decomposing proof obligations without ever leaving the world of the original Java program. The fact that these proof constructs naturally translate into guarded commands suggests that they are intuitive for the verification of imperative programs.

We note that some theorem provers support primarily tactic-style proofs. In comparison with our declarative approach, tactics must be executed to show the intermediate facts that support the proof, which can lead to problems with robustness and maintainability in the presence of changes to the names of intermediate variables and theorem prover tactics.

Programs as proofs. The Jahob proof system differs from systems based on interpreting programs as proofs [13, 36, 45] and systems such as Ynot [34] based on monadic computations within Coq. The semantic basis of Jahob is the (first-order) guarded command language (as opposed to lambda calculus). Instead of introducing a fundamental distinction between programs as proofs and types as propositions, Jahob views programs as guarded commands and (through the weakest precondition rules) as the corresponding propositions. Jahob adopts the idea that certain propositions are simple enough to serve as their own proofs. The goal of Jahob's proof language then becomes soundly modifying the guarded commands to decompose the generated formulas into simpler ones until they become self-evident. The notion of self-evidence is in principle given by the rules of first-order and higher-order logic, but is in practice determined by the state of the art in automated reasoning. In comparison with the alternatives, we believe that this approach is easier for developers to use.

Software verification tools. Software verification tools that can prove properties of linked data structures include Hob [25, 30, 31], Spec# [9], ESC/Modula-3 [16], ESC/Java [18], ESC/Java2 [12], and Jahob [46], [24]. Jahob has already been used to verify a collection of linked data structures [46], but in certain cases relied on the developer to use Isabelle to interactively prove verification conditions that the provers were unable to verify. Our results in Section 6 show that our integrated proof language completely eliminates the need to use external interactive theorem provers even when proving such complex properties. LOOP, KIV, KeY, Jive, and Krakatoa have been used to verify a variety of Java programs [1, 8, 14, 14, 17, 42]. Several of these verification environments aim to avoid the disconnect between source code and the proof obligation by incorporating the notion of imperative programs into the notion of proof and building interactive interfaces that enable manipulations of annotated source code. Instead, Jahob shows how to introduce small extensions into a standard programming language to facilitate proof decomposition. To the best of our knowledge, no other system has been used to verify a collection of linked data structures of comparable sophistication to the ones in Section 6.

The working draft of the Boogie 2 language reference manual [32] presents the call-forall statement as a means to introduce lemmas for helping the program verifier in more advanced verifications. Lemma procedures, in conjunction with the call and call-forall constructs, make it possible to verify certain universally-quantified lemmas and implications.

Using specification variables to prove quantified properties. The idea of using specification variables with arbitrary values to verify universally quantified assertions appears in extensions of predicate abstraction [7, 19]. The fix construct makes this concept directly available to the developer, providing greater control over its use. We have also identified conditions under which this approach is sound.

Summary. Over the past several decades, researchers have proposed a range of program verification approaches and tools. Soundness is an essential property for any such approach and we have demonstrated the soundness of our approach. The next most important question in practice is the feasibility of the proof system in verifying complex program properties. Our experience shows that our system effectively supports this task for a wide range of data structures. We attribute the effectiveness of our system to 1) its integration with the underlying imperative programming language and 2) its ability to incorporate the full range of automated reasoning procedures based on fragments of well-understood classical logics. We believe our system is unique in the extent to which it supports these two features.

8. Conclusion

Automated reasoning systems are becoming increasingly powerful and therefore increasingly useful in a range of domains. But despite these advances, proofs of many complex properties remain beyond the reach of fully automated techniques. Our results show that our integrated proof language can enable developers to effectively resolve key proof choice points to obtain, with reasonable effort, proofs of very sophisticated program correctness properties that otherwise lie beyond the reach of automated techniques. These results suggest that the incorporation of a reasonable amount of developer guidance can dramatically increase the sophistication of the range of important program correctness properties that it is possible to formally prove. We anticipate that similar techniques would provide similar benefits in other domains.

References

[1] W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. H. Schmitt. The KeY tool. *Software and System Modeling*, 4:32–54, 2005.

[2] P. B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*. Springer (Kluwer), 2nd edition, 2002.

[3] K. Arkoudas. *Denotational Proof Languages*. PhD thesis, Massachusetts Institute of Technology, 2000.

[4] K. Arkoudas, K. Zee, V. Kuncak, and M. Rinard. Verifying a file system implementation. In *ICFEM*, volume 3308 of *LNCS*, 2004.

[5] D. Aspinall. Proof general: A generic tool for proof development. In *TACAS*, 2000.

[6] R.-J. Back and J. von Wright. *Refinement Calculus*. Springer-Verlag, 1998.

[7] I. Balaban, A. Pnueli, and L. Zuck. Shape analysis by predicate abstraction. In *VMCAI'05*, 2005.

[8] M. Balser, W. Reif, G. Schellhorn, K. Stenzel, and A. Thums. Formal system development with KIV. In *FASE*, number 1783 in LNCS, 2000.

[9] M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.

[10] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development–Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.

[11] R. S. Boyer and J. S. Moore. Integrating decision procedures into heuristic theorem provers: A case study of linear arithmetic. In *Machine Intelligence*, volume 11, pages 83–124. OUP, 1988.

[12] P. Chalin, C. Hurlin, and J. Kiniry. Integrating static checking and interactive verification: Supporting multiple theories and provers in verification. In *VSTTE*, 2005.

[13] T. Coquand and G. P. Huet. The calculus of constructions. *Inf. Comput.*, 76(2/3):95–120, 1988.

[14] A. Darvas and P. Müller. Formal encoding of JML Level 0 specifications in JIVE. Technical Report 559, Chair of Software Engineering, ETH Zurich, 2007.

[15] L. de Moura and N. Bjørner. Efficient E-matching for SMT solvers. In *CADE*, 2007.

[16] D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. Technical Report 159, COMPAQ Systems Research Center, 1998.

[17] J.-C. Filliatre. Verification of non-functional programs using interpretations in type theory. *Journal of Functional Programming*, 13(4):709–745, 2003.

[18] C. Flanagan, K. R. M. Leino, M. Lilibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended Static Checking for Java. In *Proc. ACM PLDI*, 2002.

[19] C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *Proc. 29th ACM POPL*, 2002.

[20] C. Flanagan and J. B. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *Proc. 28th ACM POPL*, 2001.

[21] J. Gallier. *Logic for Computer Science*. `http://www.cis.upenn.edu/~jean/gbooks/logic.html`, revised on-line edition, 2003.

[22] Y. Ge, C. Barrett, and C. Tinelli. Solving quantified verification conditions using satisfiability modulo theories. In *CADE*, 2007.

[23] N. Immerman, A. M. Rabinovich, T. W. Reps, S. Sagiv, and G. Yorsh. The boundary between decidability and undecidability for transitive-closure logics. In *Computer Science Logic*, pages 160–174, 2004.

[24] V. Kuncak. *Modular Data Structure Verification*. PhD thesis, EECS Department, Massachusetts Institute of Technology, February 2007.

[25] V. Kuncak, P. Lam, K. Zee, and M. Rinard. Modular pluggable analyses for data structure consistency. *IEEE Transactions on Software Engineering*, 32(12), December 2006.

[26] V. Kuncak and K. R. M. Leino. In-place refinement for effect checking. In *Second International Workshop on Automated Verification of Infinite-State Systems (AVIS'03), Warsaw, Poland*, April 2003.

[27] V. Kuncak, H. H. Nguyen, and M. Rinard. Deciding Boolean Algebra with Presburger Arithmetic. *J. of Automated Reasoning*, 2006. `http://dx.doi.org/10.1007/s10817-006-9042-1`.

[28] V. Kuncak and M. Rinard. Existential heap abstraction entailment is undecidable. In *Static Analysis Symposium*, 2003.

[29] V. Kuncak and M. Rinard. Towards efficient satisfiability checking for Boolean Algebra with Presburger Arithmetic. In *CADE-21*, 2007.

[30] P. Lam. *The Hob System for Verifying Software Design Properties*. PhD thesis, Massachusetts Institute of Technology, February 2007.

[31] P. Lam, V. Kuncak, and M. Rinard. Cross-cutting techniques in program specification and analysis. In *4th International Conference on Aspect-Oriented Software Development (AOSD'05)*, 2005.

[32] K. R. M. Leino. This is Boogie 2. http://research.microsoft.com/ leino/-papers/krml178.pdf, June 2008. (working draft).

[33] J. Meng and L. C. Paulson. Translating higher-order problems to first-order clauses. In *ESCoR: Empir. Successful Comp. Reasoning*, pages 70–80, 2006.

[34] A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: dependent types for imperative programs. In *ICFP*, pages 229–240, 2008.

[35] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer-Verlag, 2002.

[36] B. Nordstroem, K. Petersson, and J. Smith. *Programming in Martin-Loef's Type Theory: An Introduction*. Oxford University Press, 1990.

[37] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In *11th CADE*, 1992.

[38] L. C. Paulson. *Logic and Computation: Interactive Proof with Cambridge LCF*. CUP, 1987.

[39] S. Ranise and C. Tinelli. The SMT-LIB Standard: Version 1.2. Technical report, Department of Computer Science, The University of Iowa, 2006. Available at `www.SMT-LIB.org`.

[40] P. Rudnicki and A. Trybulec. On equivalents of well-foundedness. *J. Autom. Reasoning*, 23(3-4):197–234, 1999.

[41] S. Schulz. E – A Brainiac Theorem Prover. *Journal of AI Communications*, 15(2/3):111–126, 2002.

[42] J. van der Berg and B. Jacobs. The LOOP compiler for Java and UML. Technical Report CSI-R0019, Computing Science Institute, Univ. of Nijmegen, Dec. 2000.

[43] C. Weidenbach. Combining superposition, sorts and splitting. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume II, chapter 27. Elsevier Science, 2001.

[44] M. Wenzel. *Isabelle/Isar — a versatile environment for human-readable formal proof documents*. PhD thesis, TU München, 2002.

[45] H. Xi. Dependent ML: An approach to practical programming with dependent types. *J. Funct. Program.*, 17(2):215–286, 2007.

[46] K. Zee, V. Kuncak, and M. Rinard. Full functional verification of linked data structures. In *Proc. ACM PLDI*, June 2008.

A. Soundness Proofs

Using the proof methodology described in Section 5, we prove the soundness of the translations for each of the proof language constructs given in Figure 3. Figures 10 and 11 present these soundness proofs. The simplicity of these proofs illustrates the methodological advantages of defining proof constructs through a translation into guarded commands.

p	$\mathsf{wlp}(\llbracket p \rrbracket, H)$
$p_1 \,;\, p_2$	$\quad \mathsf{wlp}(\llbracket p_1 \,;\, p_2 \rrbracket, H)$ $= \mathsf{wlp}((\llbracket p_1 \rrbracket \,;\, \llbracket p_2 \rrbracket), H)$ $= \mathsf{wlp}(\llbracket p_1 \rrbracket, \mathsf{wlp}(\llbracket p_2 \rrbracket, H))$ $\rightarrow \mathsf{wlp}(\llbracket p_2 \rrbracket, H)$ $\rightarrow H$
$\mathsf{assert}\ F$	$\quad \mathsf{wlp}(\llbracket \mathsf{assert}\ F \rrbracket, H)$ $= \mathsf{wlp}((\mathsf{assert}\ F), H)$ $= F \wedge H$ $\rightarrow H$
$\mathsf{note}\ F$	$\quad \mathsf{wlp}(\llbracket \mathsf{note}\ F \rrbracket, H)$ $= \mathsf{wlp}((\mathsf{assert}\ F \,;\, \mathsf{assume}\ F), H)$ $= F \wedge (F \rightarrow H)$ $\rightarrow H$
$\mathsf{localize\ in}\ (p \,;\, \mathsf{note}\ F)$	$\quad \mathsf{wlp}(\llbracket \mathsf{localize\ in}\ (p \,;\, \mathsf{note}\ F) \rrbracket, H)$ $= \mathsf{wlp}((\mathsf{skip}\ [\!]\ (\llbracket p \rrbracket \,;\, \mathsf{assert}\ F \,;$ $\qquad\qquad \mathsf{assume\ false})) \,;$ $\qquad \mathsf{assume}\ F), H)$ $= (F \rightarrow H) \wedge \mathsf{wlp}(\llbracket p \rrbracket, F)$ $\rightarrow (F \rightarrow H) \wedge F$ $\rightarrow H$
$\mathsf{mp}\ (F \rightarrow G)$	$\quad \mathsf{wlp}(\llbracket \mathsf{mp}\ (F \rightarrow G) \rrbracket, H)$ $= \mathsf{wlp}((\mathsf{assert}\ F \,;\, \mathsf{assert}\ (F \rightarrow G) \,;\, \mathsf{assume}\ G), H)$ $= F \wedge (F \rightarrow G) \wedge (G \rightarrow H)$ $\rightarrow G \wedge (G \rightarrow H)$ $\rightarrow H$
$\mathsf{assuming}\ F\ \mathsf{in}\ (p \,;\, \mathsf{note}\ G)$	$\quad \mathsf{wlp}(\llbracket \mathsf{assuming}\ F\ \mathsf{in}\ (p \,;\, \mathsf{note}\ G) \rrbracket, H)$ $= \mathsf{wlp}(((\mathsf{skip}\ [\!]\ (\mathsf{assume}\ F \,;$ $\qquad\qquad \llbracket p \rrbracket \,;\, \mathsf{assert}\ G \,;\, \mathsf{assume\ false})) \,;$ $\qquad \mathsf{assume}\ (F \rightarrow G)), H)$ $= ((F \rightarrow G) \rightarrow H) \wedge (F \rightarrow \mathsf{wlp}(\llbracket p \rrbracket, G))$ $\rightarrow ((F \rightarrow G) \rightarrow H) \wedge (F \rightarrow G)$ $\rightarrow H$
$\mathsf{cases}\ \vec{F}\ \mathsf{for}\ G$	$\quad \mathsf{wlp}(\llbracket \mathsf{cases}\ \vec{F}\ \mathsf{for}\ G \rrbracket, H)$ $= \mathsf{wlp}((\mathsf{assert}\ F_1 \vee \ldots \vee F_n \,;$ $\qquad \mathsf{assert}\ (F_1 \rightarrow G) \,;\, \ldots \,;\, \mathsf{assert}\ (F_n \rightarrow G) \,;$ $\qquad \mathsf{assume}\ G), H)$ $= (F_1 \vee \ldots \vee F_n) \wedge (F_1 \rightarrow G) \wedge \ldots \wedge (F_n \rightarrow G) \wedge (G \rightarrow H)$ $\rightarrow G \wedge (G \rightarrow H)$ $\rightarrow H$
$\mathsf{showedCase}\ i\ \mathsf{of}\ F_1 \vee \ldots \vee F_n$	$\quad \mathsf{wlp}(\llbracket \mathsf{showedCase}\ i\ \mathsf{of}\ F_1 \vee \ldots \vee F_n \rrbracket, H)$ $= \mathsf{wlp}((\mathsf{assert}\ F_i \,;\, \mathsf{assume}\ F_1 \vee \ldots \vee F_n), H)$ $= F_i \wedge ((F_1 \vee \ldots \vee F_n) \rightarrow H)$ $\rightarrow H$
$\mathsf{byContradiction}\ F\ \mathsf{in}\ p$	$\quad \mathsf{wlp}(\llbracket \mathsf{byContradiction}\ F\ \mathsf{in}\ p \rrbracket, H)$ $= \mathsf{wlp}(((\mathsf{skip}\ [\!]\ (\mathsf{assume}\ \neg F \,;$ $\qquad\qquad \llbracket p \rrbracket \,;\, \mathsf{assert\ false} \,;\, \mathsf{assume\ false})) \,;$ $\qquad \mathsf{assume}\ F), H)$ $= (F \rightarrow H) \wedge ((\neg F) \rightarrow \mathsf{wlp}(\llbracket p \rrbracket, \mathsf{false}))$ $\rightarrow (F \rightarrow H) \wedge ((\neg F) \rightarrow \mathsf{false})$ $\rightarrow H$
$\mathsf{contradiction}\ F$	$\quad \mathsf{wlp}(\llbracket \mathsf{contradiction}\ F \rrbracket, H)$ $= \mathsf{wlp}((\mathsf{assert}\ F \,;\, \mathsf{assert}\ \neg F \,;\, \mathsf{assume\ false}), H)$ $= F \wedge \neg F$ $\rightarrow H$

Figure 10. Soundness Proofs for Translations of Proof Language Constructs (continued in Figure 11)

p	$\mathsf{wlp}([\![p]\!], H)$
$\mathsf{instantiate}\ \forall\vec{x}.F\ \mathsf{with}\ \vec{t}$	$\mathsf{wlp}([\![\mathsf{instantiate}\ \forall\vec{x}.F\ \mathsf{with}\ \vec{t}]\!], H)$ $=\ \mathsf{wlp}((\mathsf{assert}\ \forall\vec{x}.F\ ;\ \mathsf{assume}\ F[\vec{x} := \vec{t}]), H)$ $=\ (\forall\vec{x}.F) \land (F[\vec{x} := \vec{t}] \to H)$ $\to\ F[\vec{x} := \vec{t}] \land (F[\vec{x} := \vec{t}] \to H)$ $\to\ H$
$\mathsf{witness}\ \vec{t}\ \mathsf{for}\ \exists\vec{x}.F$	$\mathsf{wlp}([\![\mathsf{witness}\ \vec{t}\ \mathsf{for}\ \exists\vec{x}.F]\!], H)$ $=\ \mathsf{wlp}((\mathsf{assert}\ F[\vec{x} := \vec{t}]\ ;\ \mathsf{assume}\ \exists\vec{x}.F), H)$ $=\ F[\vec{x} := \vec{t}] \land ((\exists\vec{x}.F) \to H)$ $\to\ (\exists\vec{x}.F) \land ((\exists\vec{x}.F) \to H)$ $\to\ H$
$\mathsf{pickWitness}\ \vec{x}\ \mathsf{for}\ F\ \mathsf{in}\ (p\ ;\ \mathsf{note}\ G)$ (where $\vec{x}$ is not free in G)	$\mathsf{wlp}([\![\mathsf{pickWitness}\ \vec{x}\ \mathsf{for}\ F\ \mathsf{in}\ (p\ ;\ \mathsf{note}\ G)]\!], H)$ $=\ \mathsf{wlp}(((\mathsf{skip}\ [\!]\ (\mathsf{assert}\ \exists\vec{x}.F\ ;\ \mathsf{havoc}\ \vec{x}\ ;\ \mathsf{assume}\ F\ ;$ $[\![p]\!]\ ;\ \mathsf{assert}\ G\ ;$ $\mathsf{assume\ false}))\ ;$ $\mathsf{assume}\ G), H)$ $=\ (G \to H) \land \exists\vec{x}.F \land \forall\vec{x}.(F \to \mathsf{wlp}([\![p]\!], G))$ $\to\ (G \to H) \land \exists\vec{x}.F \land \forall\vec{x}.(F \to G)$ $\to\ (G \to H) \land G$ $\to\ H$
$\mathsf{pickAny}\ \vec{x}\ \mathsf{in}\ (p\ ;\ \mathsf{note}\ G)$	$\mathsf{wlp}([\![\mathsf{pickAny}\ \vec{x}\ \mathsf{in}\ (p\ ;\ \mathsf{note}\ G)]\!], H)$ $=\ \mathsf{wlp}(((\mathsf{skip}\ [\!]\ (\mathsf{havoc}\ \vec{x}\ ;$ $[\![p]\!]\ ;\ \mathsf{assert}\ G\ ;$ $\mathsf{assume\ false}))\ ;$ $\mathsf{assume}\ \forall\vec{x}.G), H)$ $=\ ((\forall\vec{x}.G) \to H) \land \forall\vec{x}.\mathsf{wlp}([\![p]\!], G)$ $\to\ ((\forall\vec{x}.G) \to H) \land \forall\vec{x}.G$ $\to\ H$
$\mathsf{induct}\ F\ \mathsf{over}\ n\ \mathsf{in}\ p$	$\mathsf{wlp}([\![\mathsf{induct}\ F\ \mathsf{over}\ n\ \mathsf{in}\ p]\!], H)$ $=\ \mathsf{wlp}(((\mathsf{skip}\ [\!]\ (\mathsf{havoc}\ n\ ;\ \mathsf{assume}\ 0 \leq n\ ;$ $[\![p]\!]\ ;\ \mathsf{assert}\ F[n := 0]\ ;\ \mathsf{assert}\ (F \to F[n := n{+}1])$ $\mathsf{assume\ false}))\ ;$ $\mathsf{assume}\ \forall n.(0 \leq n \to F)), H)$ $=\ ((\forall n.(0 \leq n \to F)) \to H) \land$ $\forall n.(0 \leq n \to \mathsf{wlp}([\![p]\!], (F[n := 0] \land (F \to F[n := n{+}1]))))$ $\to\ ((\forall n.(0 \leq n \to F)) \to H) \land \forall n.(0 \leq n \to (F[n := 0] \land (F \to F[n := n{+}1])))$ $\to\ ((\forall n.(0 \leq n \to F)) \to H) \land \forall n.(0 \leq n \to F)$ $\to\ H$

Figure 11. Soundness Proofs for Translations of Proof Language Constructs (continued from Figure 10)

B. Definition and Soundness of the Fix Construct

Figure 12 gives the translation for the fix construct, which enables the developer to establish a formula of the form $\forall x.(F' \to G)$. Unlike the other proof constructs we have presented, which can only enclose other proof statements, the fix construct may enclose statements that change the program state. In the statement $\mathsf{fix}\ \vec{x}\ \mathsf{suchThat}\ F\ \mathsf{in}\ (c\ ;\ \mathsf{note}\ G)$, F and G are formulas that may contain free occurrences of the variables $\vec{x}$. The command c is an extended guarded command that may include commands that modify the program state, including, for example, Java code that has been translated into the extended guarded command language. Of course, c may also contain proof language constructs, including nested fix constructs.

The fix construct enables the developer to select arbitrary values of $\vec{x}$ that satisfy F' (provided that such values exist), where F' is the formula F evaluated at the program point before c. The proof commands and loop invariants in c can refer to $\vec{x}$, but may not change $\vec{x}$ (e.g., assigning a variable in $\vec{x}$ to a new value is prohibited). Thus, the formula F' has the same meaning at all program points in c, even though the commands in c may change the program state.

$$
\begin{aligned}
[\![\mathsf{fix}\ \vec{x}\ \mathsf{suchThat}\ F\ \mathsf{in}\ (c\ ;\ \mathsf{note}\ G)]\!] = \\
& \vec{z}_0 := \vec{z}\ ; \\
& \mathsf{assert}\ \exists\vec{x}.F'\ ;\ \mathsf{havoc}\ \vec{x}\ ;\ \mathsf{assume}\ F'\ ; \\
& [\![c]\!]\ ; \\
& \mathsf{assert}\ G\ ; \\
& \mathsf{assume}\ \forall\vec{x}.(F' \to G)
\end{aligned}
$$

where:

- $\vec{x}$ does not appear outside F, G, and proof commands and loop invariants of c;
- $\vec{z} = \mathsf{mod}(c)$ denotes variables modified in c (disjoint from $\vec{x}$);
- $\vec{z}_0$ are fresh variables (used to save old values of $\vec{z}$);
- F' stands for $F[\vec{z} := \vec{z}_0]$.

Figure 12. The fix Construct and Its Translation

At the program point after c, the translation asserts that G holds for the arbitrary values of $\vec{x}$ selected to satisfy F'. If the assertion holds, the translation adds $\forall\vec{x}.(F' \rightarrow G)$ to the assumption base.

Note that fix can be viewed as a generalization of the pickAny construct from Figure 8 in two ways. First, it permits statements that change the program state in c. Second, it allows the quantification over $\vec{x}$ to bind only those values that satisfy F'. Thus, $\vec{x}$ can be assumed to satisfy F' in c. Note that, in contrast to the assuming construct (Figure 8), the translation of the fix construct (Figure 12) must ensure that the generated assume statement will not restrict the values of any variables other than $\vec{x}$, such as, for example, Java variables within c. Indeed, if, for example, the body c of fix were the statement `u.f = v`, and F' were the formula $(\mathtt{x} \neq \mathsf{null} \wedge \mathtt{u} \neq \mathsf{null})$, then blindly assuming F' would trivialize the null dereference check on u. To enforce soundness, it is therefore necessary to ensure, before assuming F', that, for the current values of the program variables, there exists at least one value for $\vec{x}$ that satisfies F'. The command assert $\exists\vec{x}.F'$ ensures that this is the case.

The use of assert $\exists\vec{x}.F'$ has, simultaneously, another role: it ensures that fix serves as a generalization of pickWitness. The fix construct can be viewed as a generalization of the pickWitness construct from Figure 8 by taking as G the formula true. In this case fix picks $\vec{x}$ as the witness for the existentially quantified statement assert $\exists\vec{x}.F'$. Note that, if G is true, then the meaning of the translation of fix from Figure 12 is precisely havoc $\vec{x}$ suchThat F'. Therefore, fix can be used as a generalization of pickWitness that allows the use of Java code in its body c.

Soundness. To show the soundness of the proof constructs together with fix, we build on and generalize the proof in Section A. We show that inserting a set of proof commands and fix commands into a piece of Java code generates weakest preconditions that imply the weakest precondition of the concrete semantics given by the conjunction of the weakest precondition over all possible paths in the program. The soundness then essentially reduces to positive conjunctivity of commands [6, 26].

Consider an innermost occurrence of fix that contains no nested fix commands within its body c. Then we can 1) eliminate any proof commands p from c, to obtain a weaker command, then 2) replace the loop desugarings of Figure 6 with the actual loop semantics, which (by soundness of the desugaring) produces another weaker command. We finally eliminate fix itself. To show soundness, it therefore suffices to show that

$$\mathsf{wlp}(\mathsf{fix}\,\vec{x}\,\mathsf{suchThat}\,F\,\mathsf{in}\,(c\,\mathbf{;}\,\mathsf{note}\,G), H_0) \;\rightarrow\; \mathsf{wlp}(c, H_0) \quad (1)$$

where c does not contain any proof language constructs p or fix. Here the postcondition H_0 of fix does not contain $\vec{x}$, which is justified by the assumption in Figure 12 that $\vec{x}$ does not occur outside F, c, and G. Furthermore, the loops within c are not desugared but remain in the syntax tree. The semantics of $\mathsf{loop}\ \mathsf{inv}(I)\ c_1\ \mathsf{while}(D)\ c_2$ is given by the exact fixpoint semantics that ignores loop invariants, that is, as countable non-deterministic choice $[]_n\, c_n$ over c_n for $n \geq 0$, where c_n is

$$c_1\,\mathbf{;}(\mathsf{assume}\ (D)\,\mathbf{;}\,c_2\,\mathbf{;}\,c_1)^n\,\mathbf{;}\,\mathsf{assume}\ (\neg D) \quad (2)$$

We represent the remaining (non-loop) executable code constructs from Figure 2 by their desugaring (Figure 6) into the simple guarded commands of Figure 4. We define the command change as follows:

$$\mathsf{change} \;\equiv\; \mathsf{assert}\ \exists\vec{x}.F'\,\mathbf{;}\,\mathsf{havoc}\ \vec{x}\,\mathbf{;}\,\mathsf{assume}\ F'$$

To show (1), we show

$$\begin{pmatrix} \mathsf{change}\,\mathbf{;} \\ c\,\mathbf{;} \\ \mathsf{assert}\ G\,\mathbf{;} \\ \mathsf{assume}\ \forall\vec{x}.(F' \rightarrow G) \end{pmatrix} \underset{S1}{\sqsubseteq} \begin{pmatrix} c\,\mathbf{;} \\ \mathsf{change}\,\mathbf{;} \\ \mathsf{assert}\ G\,\mathbf{;} \\ \mathsf{assume}\ \forall\vec{x}.(F' \rightarrow G) \end{pmatrix} \underset{S2}{\sqsubseteq} c \quad (3)$$

We first show the step $S1$ by showing, by induction on the syntax tree of c, that $\mathsf{change}\,\mathbf{;}\,c \;\sqsubseteq\; c\,\mathbf{;}\,\mathsf{change}$. We treat the loop command from Figure 2 as one abstract syntax tree node.

Case of non-structured commands. Let c be a havoc or assume command that occurs in the body of fix. We then show

$$\mathsf{change}\,\mathbf{;}\,c \;\sqsubseteq\; c\,\mathbf{;}\,\mathsf{change}$$

By definition, this condition reduces to showing that

$$(\exists\vec{x}.F') \;\wedge\; \forall\vec{x}.(F' \rightarrow \mathsf{wlp}(c, H))$$

implies $\mathsf{wlp}(c, (\exists\vec{x}.F') \;\wedge\; \forall\vec{x}.(F' \rightarrow H))$, where H may contain $\vec{x}$. First consider the case of a command assume A.

By the assumption that the body of fix cannot contain $\vec{x}$ except in loop invariants, $\vec{x}$ does not appear in A. The property thus reduces to showing that $(\exists\vec{x}.F') \;\wedge\; \forall\vec{x}.(F' \rightarrow (A \rightarrow H))$ implies $A \rightarrow ((\exists\vec{x}.F') \;\wedge\; \forall\vec{x}.(F' \rightarrow H))$, which easily follows for A not containing $\vec{x}$.

Next consider an occurrence of havoc $\vec{y}$ that occurs within the body of fix. In that case $\vec{y}$ belongs to the modified variables $\vec{z}$ of Figure 12, which means that F' does not contain any of the variables $\vec{y}$ and that $\vec{x}$ and $\vec{y}$ are disjoint variables. This case then reduces to showing that $(\exists\vec{x}.F') \wedge (\forall\vec{x}.(F' \rightarrow \forall\vec{y}.H))$ implies $\forall\vec{y}.((\exists\vec{x}.F') \wedge (\forall\vec{x}.F' \rightarrow H))$ under these assumptions on free variables, which is straightforward.

Case of sequential composition. If by the inductive hypothesis $\mathsf{change}\,\mathbf{;}\,c_1 \sqsubseteq c_1\,\mathbf{;}\,\mathsf{change}$ and $\mathsf{change}\,\mathbf{;}\,c_2 \sqsubseteq c_2\,\mathbf{;}\,\mathsf{change}$, then

$$\mathsf{change}\,\mathbf{;}\,c_1\,\mathbf{;}\,c_2 \;\sqsubseteq\; c_1\,\mathbf{;}\,\mathsf{change}\,\mathbf{;}\,c_2 \;\sqsubseteq\; c_1\,\mathbf{;}\,c_2\,\mathbf{;}\,\mathsf{change}$$

Case of non-deterministic choice. Similarly, by the induction hypothesis

$$\begin{aligned} \mathsf{change}\,\mathbf{;}(c_1 \,[]\, c_2) &= (\mathsf{change}\,\mathbf{;}\,c_1) \,[]\, (\mathsf{change}\,\mathbf{;}\,c_2) \\ &\sqsubseteq (c_1\,\mathbf{;}\,\mathsf{change}) \,[]\, (c_2\,\mathbf{;}\,\mathsf{change}) \\ &= (c_1 \,[]\, c_2)\,\mathbf{;}\,\mathsf{change} \end{aligned}$$

From the above steps, by induction we obtain $\mathsf{change}\,\mathbf{;}\,c \sqsubseteq c\,\mathbf{;}\,\mathsf{change}$ for all commands c without loops.

Case of loop. Consider L of the form $\mathsf{loop}\ \mathsf{inv}(I)\ c_1\ \mathsf{while}(D)\ c_2$. For each n in the semantics of loops (2) the condition $\mathsf{change}\,\mathbf{;}\,c_n \sqsubseteq c_n\,\mathbf{;}\,\mathsf{change}$ follows by the previous inductive proof for loop-free commands. From the definition of change we have that its weakest precondition is positively conjunctive, so $\mathsf{wlp}(\mathsf{change}, \wedge_{n\geq 0} P_n) = \wedge_{n\geq 0}\mathsf{wlp}(\mathsf{change}, P_n)$. Using this fact we have the following.

$$\begin{aligned} \mathsf{wlp}(\mathsf{change}\,\mathbf{;}\,[]_n\, c_n, H) &= \mathsf{wlp}(\mathsf{change}, \mathsf{wlp}([]_n\, c_n, H)) \\ &= \mathsf{wlp}(\mathsf{change}, \textstyle\bigwedge_{n\geq 0} \mathsf{wlp}(c_n, H)) \\ &= \textstyle\bigwedge_{n\geq 0} \mathsf{wlp}(\mathsf{change}, \mathsf{wlp}(c_n, H)) \\ &\rightarrow \textstyle\bigwedge_{n\geq 0} \mathsf{wlp}(c_n, \mathsf{wlp}(\mathsf{change}, H)) \\ &= \mathsf{wlp}([]_n\, c_n, \mathsf{wlp}(\mathsf{change}, H)) \end{aligned}$$

This shows $(\mathsf{change}\,\mathbf{;}\,L \sqsubseteq L\,\mathbf{;}\,\mathsf{change})$ for a loop L.

Change meets assert. It remains to show the step $S2$ of (3). Define $f_0 \;\equiv\; \mathsf{change}\,\mathbf{;}\,\mathsf{assert}\ G\,\mathbf{;}\,\mathsf{assume}\ \forall\vec{x}.(F' \rightarrow G)$. We show $\mathsf{wlp}(f_0, H_0) \rightarrow H_0$. Computing $\mathsf{wlp}(f_0, H_0)$ gives

$$(\exists\vec{x}.F') \wedge \forall\vec{x}.(F' \rightarrow (G \wedge ((\forall\vec{x}.(F' \rightarrow G)) \rightarrow H_0))$$

which, after splitting the conjunct $G \wedge ((\forall\vec{x}.(F' \rightarrow G)) \;\rightarrow\; H_0)$ implies first $\forall\vec{x}.\ F' \rightarrow H_0$. From this and $\exists\vec{x}.F'$, as well as the fact that $\vec{x}$ does not appear in H_0, we obtain H_0. ∎