Automated Program Verification Made SYMPLAR

SYMbolic Permissions for Lightweight Automated Reasoning

Kevin Bierhoff

Two Sigma Investments, New York City, USA

http://www.cs.cmu.edu/∼kbierhof/

Abstract

Research in automated program verification against specifications written in first-order logic has come a long way. Ever-faster Satisfiability Modulo Theories (SMT) solvers [Barrett et al. 2010] promise to verify program instructions quickly against specifications. Unfortunately, aliasing still prevents automated program verification tools from easily and soundly verifying interesting programs. This paper introduces the use of *symbolic permissions* as the basis for sound automated program verification. Symbolic permissions provide a simple alias control mechanism with expressiveness similar to the well-known fractional permissions [Boyland 2003]. The paper shows that symbolic permissions can be enforced with a linear refinement typechecking procedure. Once permissions are checked, aliasing can essentially be ignored for the purposes of program verification, which allows taking full advantage of SMT solvers for doing the heavy verification lifting. The paper shows that a verification tool based on symbolic permissions can easily verify a design pattern with inherent aliasing challenges.

Categories and Subject Descriptors D.2.4 [*Software/Program Verification*]

General Terms Languages, Verification

Keywords Symbolic permissions, SMT solvers, aliasing

1. Introduction

Research in automated program verification against specifications written in first-order logic has come a long way. Ever-faster Satisfiability Modulo Theories (SMT) solvers [Barrett et al. 2010] promise to verify program instructions quickly against preconditions, postconditions, loop invariants, and class invariants [Barnett et al. 2004; Flanagan et al. 2002].

Unfortunately, aliasing still prevents automated program verification tools from easily and soundly verifying interesting programs. Aliasing refers to the common situation in imperative programs where multiple variables and fields on the heap point to the same object. Aliasing complicates sound modular reasoning because of its non-local nature: every method invocation in a sequence of program instructions can render conclusions about the heap useless because we have to assume that the invoked method may manipulate the very objects touched by the instructions we are reasoning about.

State-of-the-art program verification tools such as Spec# address the aliasing problem by enforcing a certain form of ownership [Barnett et al. 2004; Müller 2002] called owners-as-modifiers. Owners-as-modifiers requires that a forest of references from objects to their "owned" children be exclusively used to modify objects. Program verification based on ownership can work very well [Bierhoff and Hawblitzel 2007]. But common design patterns such as iterators and observers do not fit the owners-as-modifiers paradigm (see Section 2 and Bierhoff and Aldrich [2008]). For instance, Spec# as available today cannot verify the absence of concurrent modification exceptions (CMEs) in the usage of iterators and collections, which is a well-established challenge problem for program verification approaches. CMEs occur when a collection is modified directly while an iterator over the collection is in use.

Separation logic [Reynolds 2002] and other substructural logics have proved to be viable alternatives to first-order logic plus ownership for verifying programs manually [Parkinson and Bierman 2008]. However, SMT solvers cannot easily be made to decide separation logic assertions, complicating automation substantially. In addition, separation logic is again not able to reason about concurrently (separately) used iterators over the same collection. This is because access to the shared collection has to be explicitly passed between the different iterators [Krishnaswami 2006].

Separation logic has been combined with fractional permissions [Boyland 2003] to enable modular reasoning about parts of a program that share read access to the same memory

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Onward! 2011, October 22–27, 2011, Portland, Oregon, USA.

Copyright © 2011 ACM 978-1-4503-0941-7/11/10. . . $10.00

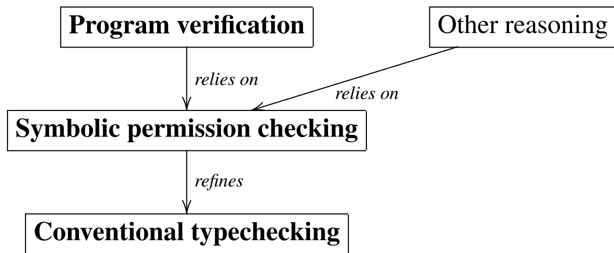

Figure 1. Refinement approach proposed in this paper

[Bornat et al. 2005]. While fractions finally allow reasoning about iterators individually [Bierhoff and Aldrich 2007; Haack and Hurlin 2008], fractional permission checking is in the author's and others' experience very difficult to automate [Bierhoff 2009; Terauchi 2008].

To summarize, aliasing still makes it very difficult for machines and humans alike to reason about complex programs. Automation is essential for lightweight program verification but with many aliasing control techniques difficult to achieve.

This paper introduces a simple type system for controlling aliasing with *symbolic permissions*. The paper shows that symbolic permissions enable sound modular automated program verification and reports on a prototype implementation of this approach. A verified iterator implementation over an array list suggests that the proposed approach allows verifying challenging programs quickly.

Symbolic permissions provide a simple alias control mechanism with expressiveness approaching that of fractional permissions [Boyland 2003]. The paper shows that symbolic permissions can be enforced as a typechecking procedure that refines conventional typechecking. Program verification of a program module can then be performed using an SMT decision procedure while *ignoring* aliasing for all variables for which a permission is available, which allows taking full advantage of SMT solvers for doing the heavy verification lifting. Implementing a verification tool based on symbolic permissions is therefore much "symplar" than previously possible. Benefits of symbolic permissions for program verification therefore include:

- *Automation.* Program verification can be effectively automated with the use of SMT solvers; symbolic permissions are easily enforced with a typechecker.
- *Express and enforce complex protocols.* Permissions offer flexibility to express and reason about properties of complex protocols such as the absence of CMEs [c.f. Bierhoff et al. 2009].

To verify these claims, I developed JavaSyp[1], a program verification tool for Java based on a symbolic permissions that uses an off-the-shelf SMT solver. JavaSyp is able to automatically verify the absense of CMEs in use and implementation of iterators over an array list (Section 2), which no existing automated behavioral program verification tool I am aware of can do. More specialized typestate checkers, including my own, are able to verify the absence of CMEs [Bierhoff et al. 2009; Bodden et al. 2007; Ramalingam et al. 2002]; this paper attempts to simplify my own previous work and take it beyond typestate checking to the more challenging context of behavioral program verification.

My proposal to separate permissions tracking using a typechecker from logical reasoning about program correctness using a decision procedure is pragmatically motivated (Figure 1):

- *Separation of concerns.* Aliasing control and reasoning about program behavior are handled separately, opening up the possibility to reason about other aspects of program behavior on top of permissions, such as typestates [Bierhoff and Aldrich 2007].
- *Simplicity.* Permission tracking by itself can be implemented as a linear typechecking procedure. At the same time, aliasing concerns do not have to encoded in logic and reasoned about with a decision procedure (as in Spec#, see Barnett et al. [2004]). This should simplify the task of program verification itself and possibly increase the degree to which it can be automated.

As has been done with fractional permissions [Bornat et al. 2005], I believe symbolic permissions could be embedded into separation logic. Such an embedding would add expressive power but would arguably also forego separation of concerns, increase complexity, and decrease chances for automation. Therefore, this paper attempts to get by without a logical embedding approach by supporting common programming idioms directly. The proposed symbolic permission type system (Section 4) to this end supports two programming idioms—borrowing and capture/release—which I previously have found to be crucial in using permissions for reasoning about real programs [Bierhoff et al. 2009].

Contributions of this paper include:

- Symbolic permissions (Section 2) as a new basis for sound automated program verification (Section 3).
- Formalization of symbolic permissions in a core object-oriented langauge with an argument for soundness of verification based on symbolic permissions (Section 4).
- Validation of the presented ideas with a working prototype implementation for Java that can verify the iterator implementation and client shown in this paper in under 2 seconds (Section 5).

Furthermore, Section 6 discusses related work and Section 7 concludes. Like most work on automated program verification with first-order logic in the last decade, examples, formalisms, and implementations are based on object-oriented programming. Subtyping and inheritance can be handled using standard techniques [Fähndrich and Xia 2007;

[1] Available at http://code.google.com/p/syper

name	fract.	access to referenced obj.	annot.
unique	1	exclusive read-write	@Excl
immutable	$(0, 1)$	shared, read-only	@Imm

Table 1. Permissions, their meaning (after Boyland [2003]), and corresponding annotations for JavaSyp

Liskov and Wing 1994] and are therefore not discussed in this paper.

2. Symbolic Permissions

This section introduces the idea of symbolic permissions. Symbolic permissions are a simplification of fractional permissions, which I will first briefly recount. I will then explain what it means to do away with fractions and instead track permissions *symbolically* based on their kind. Finally, I will introduce the concepts of borrowing, capturing, and releasing permissions, which are key to making symbolic permissions practical. A Java-like array list implementation (Figure 2) with read-only iterators (Figure 3) serves as a running example through this and the following section.

2.1 Recap: Fractional Permissions

Boyland [2003] initially proposed fractional permissions for ensuring race freedom of data that could at different times be modified by only one thread or only read by multiple threads in parallel. In other words, they formalize that it is safe to concurrently access a single object in memory if none of the accessing threads modify that object; while an object is modified it is only accessible by one thread.

This idea can be formalized by associating a *fraction* $k \in (0, 1]$ with every program reference (variable or memory location), often written $k \cdot x$. Boyland coined a reference with a fraction of 1 a unique permission; references with any fraction strictly less than 1 (but greater than 0) are called immutable permissions (Table 1).

A permission can be *split* ($\Rrightarrow$) by dividing up its fraction among two references so that they new fractions sum up to the split fraction. Permissions also can be *joined* ($\Lleftarrow$) by associating the sum of their fractions with a single reference:

$$k \cdot x \Lleftarrow\!\Rrightarrow k_1 \cdot x_1, k_2 \cdot x_2 \;\; (\textit{if } k = k_1 + k_2)$$

A non-deterministic type system ensures that unique and immutable permissions do not co-exist (which would represent data races); essentially the type system guarantees that fractions associated with a single object do not sum up to more than 1. The type system is non-deterministic because it "guesses" how fractions are split up among references.

In addition to being used for avoiding race conditions [Heule et al. 2011; Terauchi and Aiken 2008; Zhao 2007], fractions have also been used for verification [Bierhoff and Aldrich 2007; Bornat et al. 2005] and combinations of the two [Beckman et al. 2008].

2.2 We Don't Need Fractions

While formally appealing, automated reasoning about fractions requires significant engineering and a sophisticated decision procedure for rationals or integers [Bierhoff 2009; Heule et al. 2011; Terauchi 2008]. One reason is that permissions can be split among references (e.g., at method call sites) in infinitely many ways; which split should be chosen depends on what fractions the different references need later on in the program.

Fortunately, the "concrete" fractions associated with immutable permissions do not matter: by design it is irrelevant for reasoning about object accesses through a reference whether that reference has a 0.5 or 0.25 fraction. Fractions are in fact only relevant for joining permissions to regain a unique permission for modifying a previously immutable object.

Furthermore, a programmer likewise does not care about fractions. It is far more convenient to simply declare a reference as needing a unique or any immutable permission than to declare it as $0.25 \cdot x$! Not only does using immutable$\cdot x$ introduce a certain amount of "polymorphism" over fractions [Bierhoff 2009], but it is also stable against changes in the program that would otherwise require changing $0.25 \cdot x$ into $0.5 \cdot x$ because some additional method is now called that happens to require a larger fraction than is available. A single such signature change could have ripple effects through the entire program even through nothing substantially changed: the program still works with read-only access to the referenced object. Previously I and others have approached this problem by somehow "hiding" concrete fractions from programmers using inter-procedural inference or fraction polymorphism [Bierhoff 2009; Heule et al. 2011; Terauchi 2008].

The idea of symbolic permissions is to instead represent and track permissions symbolically by their "kind"—i.e., as unique or immutable—rather than with fractions and fraction variables. This approach dramatically simplifies automation to an almost-trivial procedure that can be performed by a refinement type checker. In JavaSyp, the Java annotations @Imm and @Excl signify references with immutable and unique permission, respectively.

2.3 Borrow, Capture, Release

The good news is that in most all cases we don't need the full generality of fractions anyway, as suggested by my experience with using permissions for typestate verification [Bierhoff et al. 2009]. But, we need *borrowing* [Boyland and Retert 2005] as well as *capture* (similar to adoption [Fähndrich and DeLine 2002], although we don't use "focus" here) and *release* [Bierhoff et al. 2009]. In a nutshell, a method that *borrows* a permission returns that permission exactly the way it was passed into the method. An object *captures* a permission in a field if all subsequent accesses to the field borrow that permission. The type system is then allowed to *release* (return back to the client) captured permis-

```
1   public class ArrayList<T> {
2
3   @Excl private T[] a;
4   private int size;
5   //@ invariant 0 <= size & a != null & size <= a.length;
6
7   //@ ensures size == 0;
8   public ArrayList() {
9       this(16);
10  }
11
12  //@ requires 0 <= initialCapacity;
13  //@ ensures size == 0;
14  public ArrayList(int initialCapacity) {
15      super();
16      a = (T[]) new Object[initialCapacity];
17      size = 0;
18  }
19
20  //@ requires 0 <= index & index < size;
21  @Imm public T get(int index) {
22      imm: return a[index];
23  }
24
25  //@ ensures size == \old(size) + 1;
26  @Excl(mod = {"a", "size"}) public void add(T e) {
27  excl: {
28      if (a.length <= size) {
29          @Excl(borrowed = false) final T[] newA =
30              (T[]) new Object[a.length * 2 + 1];
31          final int oldSize = size;
32          final T[] oldA = a;
33          int i = 0;
34          //@ maintaining 0 <= i;
35          //@ size == oldSize & a == oldA;
36          for (; i < size; ++i)
37              newA[i] = a[i];
38          a = newA;
39      }
40      a[size++] = e;
41  } }
42
43  //@ ensures \result == size;
44  @Imm public int size() {
45      imm: return size;
46  }
47
48  //@ ensures \result != null;
49  @Imm(released = true)
50  @ReturnExcl public ArrIterator<T> iterator() {
51      imm: {
52          @Excl final ArrIterator<T> result =
53              new ArrIterator<T>(a, size);
54          return result;
55      }
56  }
57  }
```

Figure 2. Verified ArrayList implementation

sions when the capturing object becomes garbage. Objects become garbage when unique permissions are abandoned [Bierhoff 2009].

Recall that from a technical perspective, fractions let us join permissions after they were split. Borrowing and capture/release let us do just that with symbolic permissions for objects shared on the stack and in the heap, respectively.

2.3.1 Borrowing vs. Permission Consumption

In order to explain borrowing, let's first look at its opposite in the constructor of the following code snippet. It assigns the given receiver permission to a field:

```
1   class Consumer {
2   @Imm ArrayList list;
3   Consumer(@Imm(borrowed = false) ArrayList l) {
4       list = l;
5   } }
6   // client code
7   @Excl(borrowed = false) ArrayList<Object> l =
8       new ArrayList<Object>();
9   int zero = l.size();
10  l.add(new Object());
11  @Excl Consumer c = new Consumer(l);
```

I will call this permission *consumption*: a permission to a constructor argument is consumed by that constructor by holding on to it in one of the fields of the new object. Methods can consume permissions in the very same way. In JavaSyp we explicitly indicate consumed permissions by setting the borrowed attribute in permission annotations to **false**.

Borrowing represents the absence of permission consumption in a method and transitively in all methods it calls. The method size() in Figure 2 is an example of such a method. Here, @Excl and @Imm annotations on methods define receiver permissions.

As the vast majority of method arguments are borrowed in practice [Bierhoff et al. 2009], the borrowed attribute is true by default, so get() and size() both borrow a receiver permission. The same annotations on fields such as ArrayList.a mean that these fields hold the respective permission to the referenced object (if not **null**). I will refer to such permissions as *field permissions*.

Now consider typechecking the "client code" starting at line 7 above. Assuming **new** ArrayList() yields a unique permission, our caller can clearly call size on l: we can split the unique permission for l into two immutable permissions, one of which we pass into the method. But can we call add afterwards? Only if we can join the two permissions for l back together. The fact that method size borrows the receiver permission allows us to do just that! Since the permission we pass into that method comes back exactly the way it was we are allowed to reinstate the unique permission for l after size returns. This allows calling add, and by similar reasoning to before, we can afterwards create a Consumer object. That call, however, leaves us only with an immutable per-

mission for l, which still allows calling size, but calling add is no longer allowed. Also, we now consumed part of the original permission to l, and we are therefore required to declare l as consumed on line 7.

It is instructive to consider how this example would be handled with fractions.

- size would be declared to require some fraction and yield the same fraction for the receiver.
- add would require and yield a 1 fraction for the receiver.
- Consumer's constructor would require some fraction and not yield any permission for the receiver.

Thus, borrowing is characterized by yielding the same fraction that was passed in; consuming means yielding no permission. Notice that permission consumption encodes most methods that requires some fraction and yields a different (smaller) fraction without loss of expressiveness: intuitively, one can subtract the smaller fraction from both sides, unless the larger fraction was 1 (which we could encode by requiring and yielding different symbolic permission kinds).

2.3.2 Capture and Release

Where borrowing represents temporary sharing on the stack, capture and release allow substantially the same thing in the heap. Consider an instance i of class ArrIterator that requires a permission to a helper array snapshot during construction (Figure 3). The iterator can then use the helper array throughout and does not further split the permission to the array. We will say that the permission for snapshot is *captured* by i. When i is no longer in use then we can *release* its captured permission to snapshot back to the client (in this example, the array list being iterated).

This reasoning allows the sane method in Figure 4 to typecheck and in fact consider the parameter l as borrowed: The locally used objects it1 and it2 each capture a immutable permission to l.a. These l.a permissions in turn capture immutable permissions to l. Since it1 and it2 are abandoned at the end of the method and not consumed anywhere (the local variables are declared as borrowed), l's originally permission can be returned to the caller at the end of sane.

2.4 Exposure: Field Access

Ultimately we want to use permissions to access and modify fields. Unsurprisingly, a unique permission will be required to assign to fields of an object; immutable permissions will allow reading values. These rules are sufficient for reading and writing primitive fields (such as size in Figure 2).

But fields will often have object type themselves, in which case they are associated with a *field permission*. For instance, the a field in Figure 2 is declared to hold a unique permission to the referenced array object.

Given a permission to an object o, an interesting question is then what permission will be available to access that object's fields $o.f$ with object type. Since immutability is

```
1   public class ArrIterator <T> {
2
3   @Imm private final T[] snapshot;
4   private final int count;
5   private int index;
6   //@ invariant snapshot != null & 0 <= count &
7   //@ count <= snapshot.length & 0 <= index & index <= count;
8
9   //@ requires snapshot != null && 0 <= count &&
10  //@       count <= snapshot.length;
11  public ArrIterator (@Imm(released = true) T[] snapshot,
12          int count) {
13      this.snapshot = snapshot;
14      this.count = count;
15      this.index = 0;
16  }
17
18  //@ ensures \result == index < count;
19  @Imm public boolean hasNext() {
20      imm: return index < count;
21  }
22
23  //@ requires index < count;
24  @Excl(mod = "index") public T next() {
25      excl: return snapshot[index++];
26  }
27  }
```

Figure 3. Verified iterator over ArrayList in Figure 2

a "deep" property that is supposed to guarantee that an object does not change in any way, we have to *weaken* field permissions read through immutable permissions to always be immutable. Reading through a unique permission will yield the original permission assigned to the field.

Because field reads yield permissions, we have to be able to track for how long those field permissions are used after reading a field. In particular, if a field carries a unique permission, as a in Figure 2, then we don't want that permission to disappear on us. More to the point, we should not be able to use an object if one of its field permissions is consumed (that would break the declared "invariant" of expected field permissions).

To simplify detecting these sorts of problems, we will use *exposure blocks* to delineate field access. Receiver fields will be accessible inside exposure blocks, but not outside. An exposure block captures a permission to the receiver; hence we can distinguish unique and immutable exposure of the receiver depending on what kind of permission the exposure block captures. The permission used to expose the receiver is inaccessible inside the exposure block but may be returned to the client at the end of the block (see below). By the reasoning from above, fields can be assigned inside unique exposure blocks, and field reads yield the field's original permission. Reading fields inside immutable

```
1   //@ requires l != null;
2   public <T> boolean sane(@Imm ArrayList<T> l) {
3       @Excl ArrIterator<T> it1 = l. iterator ();
4       @Excl ArrIterator<T> it2 = l. iterator ();
5       assert it1 != it2;
6       while ( it1 .hasNext() & it2 .hasNext()) {
7           // it1 . index < it1 . count & it2 . index < it2 . count
8           if ( it1 . next () /* it2 . index<it2 . count*/
9                       != it2 . next ())
10              return false;
11      }
12      if (l . size () % 2 == 1)
13          l .add(null); // ERROR: Need Excl but have Imm
14      return ! it1 .hasNext() & ! it2 .hasNext();
15  }
```

Figure 4. Sample client for classes in Figures 2 and 3 with facts known between loop statements in comments and seeded permission error that exposes Concurrent Modification Exception (CME)

exposure results in weakened field permissions as discussed above.

If at the end of an exposure block field permissions are missing then the permission that was exposed will be deemed consumed; otherwise the exposed permission is released and can be used again. While that seems like an odd rule, it is sound while flexible: the rule does the obvious and desirable if no field permissions are consumed inside an exposure block, but it does allow field permissions to be consumed.[2]

Figure 2 illustrates exposure blocks for unique and immutable receiver permissions, which we signify to JavaSyp with block or statement labels named excl: and imm:, respectively (this is legal Java [Beckman et al. 2008]!). For simplicity, JavaSyp does not support exposing references other than the receiver, but there is no theoretical problem exposing other references. Exposure blocks are in fact inspired by Spec#'s expose blocks, which can be used with an arbitrary reference [Barnett et al. 2004]. Just as in Spec#, exposure blocks not only help tracking field permissions but also simplify verifying that class invariants hold (Section 3.2).

Notice that it is never a problem to expose multiple permissions at once: Because there can be only one unique permission per object, two unique exposures at the same time can never refer to the same object (which would be bad because we could get two unique permissions to fields).

3. Program Verification with Permissions

This section will discuss the kinds of assertions we want to verify, how permissions help ensuring that class invariants hold, outline a procedure for verifying a method against contracts, and illustrate how some of the code from the running array list example will be verified. As we will see, once permissions are in place, we will pretty much be able to read off verification conditions (VCs) from code, feed them to an SMT solver, and get sound program verification.

3.1 Assertions and Contracts

We can specify the array list's expected behavior using *contracts*. Figure 2 shows JML contracts [Leavens et al. 1999] that can be understood by JavaSyp. The array list implementation in Figure 2 with its iterator (Figure 3) and a sample client (Figure 4) continues to serve as a running example thoughout this section. Array list and iterator are simplified versions of the ArrayList class in the Java standard library.

An array list maintains a list of objects ("elements") in the order they were added (using the add method) in a backing array. The backing array is grown as needed to add more elements. The get method returns the element with the given index, and size returns the number of elements in the list. Notice that the backing array can have more cells than the list has elements because the array's size doubles every time it has to grow to accommodate more elements. The iterator method returns a Java-like iterator of the array list whose implementation is shown in Figure 3. Figure 4 shows a simple array list client that uses two iterators to make sure they return the same objects.

Let's consider some example contracts:

- *Pre-conditions* that callers have to meet. get's pre-condition, for instance, is that the index parameter be between 0 (inclusive) and the list's size field (exclusive).
- *Post-conditions* have to hold at the end of a method. For instance, the ArrayList constructors promise that the size field will be 0. add's post-condition means that when add returns, size will be one larger than when add was called.
- *Loop invariants* are declared using **maintaining** JML clauses and have to hold at the beginning and end of every loop iteration. JavaSyp does not perform any inference of loop invariants, so the invariant in add's loop has to include the rather obvious lower bound for i.
- *Assertions* are encoded with Java's built-in **assert** statement. The assertion in Figure 4, for instance, requires proving that it1 and it2 do not reference the same object.
- *Assumptions* allow programmers to "inject" knowledge into the verification process. This paper does not resort to assumptions, but JavaSyp supports them.

The iterator implementation in Figure 3 likewise uses JML contracts to define how to properly use hasNext and next. Notice how the constructor's pre-condition becomes the iterator's class invariant. With these contracts, no further specifications are required for JavaSyp to check the client in Figure 4 (the **assert** statement is for illustrative purposes).

[2] In the spirit of Beckman [2010], immutable field permissions marked with borrowed = **false** can be consumed without losing the exposed permission. For soundness, such field permissions cannot be released.

3.2 Class Invariants

Class invariants are assertions over a class's instance fields that must hold for a given instance of the class whenever a client of that instance could assume the invariant is holding. Making sure of that is indubitably the hardest aspect of contract-based program verification because class invariants typically should continue to hold beyond the point in the program execution where we proved them so we can assume they still hold the next time we access a field.

With permissions, we can ensure the class invariant of an object holds as long as a client has a permission to the object. We can do so by requiring that the invariant holds unless the object's unique permission is exposed. This satisfies our requirement that invariants always hold when a client has a permission: while the unique permission is exposed, no client has a permission to the object (since the exposed permission is captured during exposure), and hence no client could assume the invariant is holding. On the other hand, if the object's unique permission is not exposed, clients can have a permission to the object and we are guaranteed that the invariant holds.

Class invariants can then be operationalized (analogously to Barnett et al. [2004] and Bierhoff and Aldrich [2007]) as follows:

- We get to assume the class invariant at the beginning of an exposure block.
- We have to prove that the class invariant holds at the end of excl: exposure blocks. (Since fields transitively cannot change in imm: exposure blocks, we don't have to prove the invariant at the end of those blocks.)
- We also have to prove class invariants at the end of constructors (which mark the end of exposures as well).

Crucially, this approach requires that invariants only mention fields of objects reachable through permissions. For instance, a class invariant `f.x == g.h.y` is only legal if fields `f`, `g`, and `g.h` are declared with immutable or unique field permissions. Under this condition, the field permissions for `f`, `g`, and `g.h` are *necessary* to modify `f.x` and `g.h.y`, which can only be acquired by exposing a unique permission to an instance of the class defining the invariant, making this approach sound. (Recall from the previous chapter that exposing a immutable permission results in "weakened" permissions for fields.)

3.3 Verification Procedure

Verifying a method now becomes a collaboration between a programming-language specific tool and an SMT solver: the language-specific tool produces verification conditions (VCs) from contracts as well as the semantics of the program instructions encountered. VCs encode the program and the assertion that is to prove as a first-order logic predicate. Once we have constructed a VC we ask a decision procedure, such as an SMT solver, whether the VC holds, which allows us to conclude the truth of the assertion.

Deriving VCs requires maintaining knowledge about local variables as well as (portions of) the heap, with the ability to *havoc* individual variables or the heap. "Havocking" variables means assuming that the havocked variables (or heap portions) could now hold any values, effectively invalidating the knowledge about those variables (or heap portions).

Verification of a method is pretty standard and begins by assuming the method's pre-condition. We then process program instructions in the order of execution, with special treatment of control flow structures (loops and conditionals).

- Whenever we encounter a method (or constructor) invocation, we have to prove the method's pre-condition and havoc the parts of the heap that could be modified by the method. Then we get to assume the method's post-condition. We do not have to havoc objects for which the caller retains a permission (see below).
- Whenever we encounter a programmer-provided assertion we prove it.
- We get to assume programmer-provided assumptions as we encounter them.
- Verification of loop invariants requires a combination of proving and assuming the invariant, as previously described for Spec# [Barnett et al. 2004].
- Conditional branches can be processed one by one from their common initial knowledge. After the conditional, the knowledge is a disjunction of the knowledge we had at the end of each branch.
- Assignments havoc the assigned variable or field and then equate it with whatever is known about the new value.
- Class invariants are handled as described in the previous section based on exposure blocks.

A crucial advantage of using permissions for verification comes from the details of handling method invocations. Conventionally, method invocations result in information loss about objects accessible through local variables and fields *not involved in the invocation*. For instance, when the sane method in Figure 4 calls the next method that may modify it1 then previous verification tools can get into trouble proving predicates involving the local variables it2 and l. This is because those locals may alias with it1 (reference the same object), and so the method call may have modified the object pointed to by it2, resulting in spurious warnings about the precondition of `it2.next()` not holding.

Conversely, with permissions, any locals and fields for which the caller retains a permission during a method invocation will not be modified by that invocation. In other words, any permissions a caller does *not* pass into a method allow that caller to assume that the object accessible through the unused permission is not modified. (JavaSyp encodes this non-interference in VCs with inequalities and "protec-

tion" predicates derived from field permission declarations in assumptions for havocking the heap.)

In separation logic, this effect is provided by the *frame rule*, which can be extended to verification with fractional permissions (as in Bornat et al. [2005]). Likewise, the frame rule could be extended to symbolic permissions, but in this paper we investigate tracking symbolic permissions with a separate type system rather than directly in the logic used for program verification.

Unlike Spec# [Barnett et al. 2004] and JML [Leavens et al. 1999] we do not need "deep" modifies clauses that describe all heap portions modified by a method because our permission-defined frame of untouched variables serves a similar purpose without the overhead. For added convenience we only havoc those fields of objects passed into a method that are listed in mod attributes. For instance, calling next in Figure 3 does not havoc what we know about count. This is simply to avoid x == \old(x) assertions, which I believe reduces annotation burden. I plan to investigate the impact of encoding mod attributes with the JML in future work.

3.4 Examples

To illustrate verification with symbolic permissions, let's consider the sane method in Figure 4. The knowledge available to JavaSyp after various instructions is shown in comments (omitting non-nullness predicates for readability).

The method creates two iterators it1 and it2 over the given list l. These iterators each *capture* an immutable permission to l. Inside the loop, calling hasNext *ensures* the exact predicate *required* for calling next on the same iterator. Notice that we can call next on it1 without problem for calling the same method on it2. This is because we retain a unique permission to it2 during the (modifying) call to it1, allowing us to maintain the knowledge we gained from calling it2.hasNext() until we need it. While we use iterators, we cannot modify the array list, which is a good thing because such a modification would represent a CME.[3]

At the end of the method, the two iterators become unavailable (because their unique permissions are declared as borrowed and indeed are not consumed anywhere in the method), which allows releasing the list permissions they capture. That, in turn, means the method parameter l's permission is also borrowed, as declared in sane's signature.

Notice that we do not need a loop invariant in this method, as the loop's condition establishes everything we need to know inside the loop body. Conversely, a loop invariant appears in the add method in Figure 2. The loop invariant helps proving the array bounds needed in the loop as well as the class invariant and post-condition: JavaSyp, like Spec# [Barnett et al. 2004], for simplicity assumes that the fields size and a mentioned in the method's mod attributes could change in loops (even though that's not the case here). Assuming the declared class invariant at the onset of the excl: block, the loop invariant allows proving that the class invariant still holds at the end of that block.

Using JavaSyp helped me find a bug in my ArrayList code of which I was previously unaware: it is not enough to double the length of the underlying array when growing it—one must also add a constant factor. Otherwise the array would not actually grow if it initially was of zero length, violating the class invariant that size < a.length. java.util.ArrayList in the Java standard library employs the same strategy of adding 1 when growing the underlying array that my implementation (now) uses.

4. Formalization of Symbolic Permissions

Symbolic permission checking can be implemented as a simple, syntax-directed typechecking procedure. This section formalizes symbolic permission checking for a core object-oriented language that is similar to my previous work [Bierhoff and Aldrich 2007] and inspired by Featherweight Java (FJ) [Igarashi et al. 1999]. It formalizes permission checking as a *refinement type system*: it assumes that conventional (class) types have already been established, e.g., as formalized in FJ. In a few places, permission checking relies on class types to look up type declarations for fields and methods, but it proceeds otherwise independently from conventional typechecking.

4.1 Syntax

Syntactic forms are summarized in Figure 5. Expressions M are unsurprising and include variable access, object creation (new), method invocation, a let binding construct, field assignment, and field access. For simplicity, programs are written in A-normal form: expressions are not allowed as part of other expressions except in a binding. This is a common trick that simplifies the theory [Bierhoff and Aldrich 2007]. Assignments will evaluate to the field's *previous* value (and its permission). To keep typechecking syntax-directed, field reads $x : p.f^c$ are annotated with the permission expected for the accessed object (this helps deciding which permissions are available for fields). Instead of the "expose" blocks that JavaSyp supports, field reads will be bounded by a surrounding let binding, and field assignments can happen whenever a unique permission is available. This simplifies the formalization. Field reads and method invocations are annotated with the class the field or method belongs to—this information is assumed to come from conventional typechecking.

Classes C are lists of fields $f : p$ and methods B, and a program consists of the "class table", $C_1 \cdots C_n$ (the list of all classes), and a main expression. Permissions p include unique and immutable. Borrowing is indicated with a 0 be-

[3] Note that JavaSyp can in also handle modifying iterators (that include a remove method), although they have to be created using a separate method. This is because modifying iterators need a unique permission to the iterated list and JavaSyp does not support multiple signatures for a single method as the author's previous work does [Bierhoff et al. 2009].

$$
\begin{array}{rcl}
\textit{prog. } R & ::= & \langle C_1 \cdots C_n, M\rangle \\
\textit{class } C & ::= & \texttt{class}\ c\ \{f_1 : p_1; \ldots; f_k : p_k; B_1 \cdots B_l\} \\
\textit{meth. } B & ::= & u_0 p_0\ m(x_1 : u_1 p_1, \ldots, x_n : u_n p_n) : T = M \\
\textit{exprs. } M & ::= & x \\
& | & \texttt{new}\ c(x_1, \ldots, x_n) \\
& | & x_0.m^c(x_1, \ldots, x_n) \\
& | & \texttt{let}\ x : up = M_1\ \texttt{in}\ M_2 \\
& | & x.f^c := y \\
& | & x : p.f^c \\
\textit{perms. } p & ::= & \mathsf{unique} \mid \mathsf{immutable} \\
\textit{borrow } u & ::= & 0 \mid 1 \\
\textit{ctxs. } \Gamma & ::= & \cdot \mid \Gamma, x : P \\
\textit{var. ts. } P & ::= & T \mid P \backslash p \\
\textit{ex. ts. } T & ::= & p[x_1 : p_1, \ldots, x_n : p_n]
\end{array}
$$

variables x, y *methods* m *fields* f *classes* c

Figure 5. Syntax

fore types T in method declarations and `let` bindings. Consumed permissions are indicated with a 1. Types T include permissions with a list of captured variables; p is a shorthand for $p[\cdot]$, which types a variable that does not capture any permissions. Additionally, variable contexts Γ can mark consumed permissions with $x : T\backslash p$, meaning that p was consumed from T, leaving potentially no or a less powerful permission for x.

4.2 Typing

Typing rules are shown in Figure 6, with helper rules in Figure 7. Typechecking proceeds with the following judgment:

$$\Gamma \vdash M : T \mid \Gamma'$$

This can be read as, "under variable context Γ, expression M is assigned type T and yields context Γ' for typechecking subsequent expressions". The "output" context Γ' reflects permissions consumed by M. Implicity, the class table is available when typechecking expressions.

Variable contexts Γ are assumed to not include the same variable name twice, which can be achieved with alpha-conversion. Let's discuss each rule in turn.

T-VAR types a variable and consumes the required permission in the process. T-BORROW and T-CALL ensure that borrowed permissions do not remain consumed, but it is easier to always assume that a variable's permission will be consumed in this rule.

T-NEW ensures that the arguments provided match the permissions needed for fields of the newly constructed object. The new object is typed as unique.

T-CALL likewise ensures that the arguments provided match the method's signature for all arguments. In a "second pass", the rule limits itself to consuming only consumed ($u = 1$) permissions, leaving borrowed permissions untouched in the resulting context Γ''. The first pass is nonetheless necessary to catch situations in which the same variable is used in two argument positions and cannot satisfy the permissions needed for both parameters.

T-BORROW makes sure that the initial permission for the borrowed variable x is fully returned by M_2 and not consumed. The rule then releases any permissions captured by x back into the context Γ_2. By contrast, T-CONSUME has no requirement on x's permission after checking M_2 but also does not release captured permissions.

T-ASSIGN requires a unique permission for the object whose field is assigned. It also requires the assigned field's permission for the new field value. While the latter permission is consumed, the former is not.

T-READEXCL types a field read with the accessed field's permission and captures the accessed object's (x) permission with it. T-READIMM works similarly but weakens any field permission to immutable.

T-METHOD unsurprisingly checks a method B by typechecking the method body under the context formed by the method parameters. Borrowed variables have to be available in their original form in the output context Γ.

T-CLASS and T-PROGRAM are standard and check each method in each class before checking the program expression M under empty variable context.

Notice that these rules support returning captured variables from method calls, which would come from field accesses inside that method. Permissions captured in fields of new objects are not formalized but could be added to T-NEW.

4.3 Soundness

This section discusses the soundness of the presented approach in two parts. We first briefly review existing permission-related soundness results. Afterwards we sketch a proof of soundness for permission-based verification. The soundness argument follows Barnett et al. [2004]. Formally proving soundness of the system presented in this paper is future work.

4.3.1 Symbolic Permission Tracking

Fractional permissions have been repeatedly proven sound in the context of typestate verification [Beckman et al. 2008; Bierhoff and Aldrich 2007] as well as for concurrent programs [Terauchi and Aiken 2008]. Borrowing and adoption (similar to capture) have however been proven sound in isolation [Aldrich et al. 2002; Boyland and Retert 2005].

4.3.2 Verification

To show soundness of the Symplar verification approach presented in Section 3 our biggest concern must be that class invariants continue to hold during evaluation [Barnett et al. 2004]. This is because while pre-conditions, post-conditions, and other assertions only have to hold at the point they are proved, we prove class invariants at the end

T-VAR
$$\frac{P \vdash T \qquad T = p[x_1 : p_1, \ldots, x_n : p_n]}{\Gamma_1, x : P, \Gamma_2 \vdash x : T \mid \Gamma_1, x : P \backslash p, \Gamma_2}$$

T-NEW
$$\frac{\Gamma \vdash x_1 : p_1, \ldots, x_n : p_n \mid \Gamma' \quad \mathsf{fields}(c) = f_1 : p_1, \ldots, f_n : p_n}{\Gamma \vdash \mathtt{new}\ c(x_1, \ldots, x_n) : \mathsf{unique} \mid \Gamma'}$$

T-CALL
$$\frac{\Gamma \vdash x_0 : p_0, x_1 : p_1, \ldots, x_n : p_n \mid \Gamma' \quad \Gamma \vdash x_0 : u_0 p_0, x_1 : u_1 p_1, \ldots, x_n : u_n p_n \mid \Gamma'' \quad \mathsf{mtype}(c.m) = u_0 p_0 (u_1 p_1, \ldots, u_n p_n) \to T}{\Gamma \vdash x_0.m^c(x_1, \ldots, x_n) : T \mid \Gamma''}$$

T-BORROW
$$\frac{\Gamma \vdash M_1 : T_1 \mid \Gamma_1 \qquad \Gamma_1, x : p \vdash M_2 : T \mid \Gamma_2, x : p \qquad T_1 = p[x_1 : p_1, \ldots, x_n : p_n] \qquad x \textit{ not occurring in } T}{\Gamma \vdash \mathtt{let}\ x : 0p = M_1\ \mathtt{in}\ M_2 : T \mid \Gamma_2 + x_n : p_n + \ldots + x_1 : p_1}$$

T-CONSUME
$$\frac{\Gamma \vdash M_1 : T_1 \mid \Gamma_1 \qquad \Gamma_1, x : p \vdash M_2 : T \mid \Gamma_2, x : P \qquad T_1 = p[x_1 : p_1, \ldots, x_n : p_n] \qquad x \textit{ not occurring in } T}{\Gamma \vdash \mathtt{let}\ x : 1p = M_1\ \mathtt{in}\ M_2 : T \mid \Gamma_2}$$

T-ASSIGN
$$\frac{\Gamma \vdash x : \mathsf{unique} \mid \Gamma_1 \qquad \Gamma_1 \vdash y : p \mid \Gamma_2 \qquad f : p \in \mathsf{fields}(c)}{\Gamma \vdash x.f^c := y : p \mid \Gamma_2 + x : \mathsf{unique}}$$

T-READEXCL
$$\frac{\Gamma \vdash x : \mathsf{unique} \mid \Gamma' \qquad f : p \in \mathsf{fields}(c)}{\Gamma \vdash x : \mathsf{unique}.f^c : p[x : \mathsf{unique}] \mid \Gamma'}$$

T-READIMM
$$\frac{\Gamma \vdash x : \mathsf{immutable} \mid \Gamma' \qquad f : p \in \mathsf{fields}(c)}{\Gamma \vdash x : \mathsf{immutable}.f^c : \mathsf{immutable}[x : \mathsf{immutable}] \mid \Gamma'}$$

T-METHOD
$$\frac{\textit{this} : p_0, x_1 : p_1, \ldots, x_n : p_n \vdash M : T \mid \Gamma \qquad \Gamma = \textit{this} : P_0, x_1 : P_1, \ldots, x_n : P_n \qquad \forall i.u_i = 0 \textit{ implies } P_i = p_i}{u_0 p_0\ m(x_1 : u_1 p_1, \ldots, x_n : u_n p_n) : T = M\ \mathsf{ok\ in\ c}}$$

T-CLASS
$$\frac{B_1\ \mathsf{ok\ in\ c} \qquad \cdots \qquad B_l\ \mathsf{ok\ in\ c}}{\mathtt{class}\ c\,\{f_1 : p_1; \ldots; f_k : p_k; B_1 \cdots B_l\}\ \mathsf{ok}}$$

T-PROGRAM
$$\frac{C_1\ \mathsf{ok} \qquad \cdots C_n\ \mathsf{ok} \qquad \cdot \vdash M : T \mid \Gamma}{\langle C_1 \cdots C_n, M\rangle\ \mathsf{ok}}$$

Figure 6. Symbolic permission typechecking rules (helper judgments in Figure 7)

$$\frac{}{T \vdash T} \qquad \frac{}{T \vdash \mathsf{immutable}[\cdot]}$$

$$\frac{P \vdash \mathsf{immutable}[\cdot]}{P \backslash \mathsf{immutable} \vdash \mathsf{immutable}[\cdot]}$$

$$\frac{\Gamma_0 \vdash x_1 : 1T_1, \cdots, x_n : 1T_n \mid \Gamma_n}{\Gamma_0 \vdash x_1 : T_1, \cdots, x_n : T_n \mid \Gamma_n}$$

$$\frac{\Gamma_0 \vdash x_1 : u_1 T_1 \mid \Gamma_1 \qquad \cdots \qquad \Gamma_{n-1} \vdash x_n : u_n T_n \mid \Gamma_n}{\Gamma_0 \vdash x_1 : u_1 T_1, \cdots, x_n : u_n T_n \mid \Gamma_n}$$

$$\frac{\Gamma \vdash x : T \mid \Gamma'}{\Gamma \vdash x : 1T \mid \Gamma'} \qquad \frac{}{\Gamma \vdash x : 0T \mid \Gamma}$$

$$\frac{}{\Gamma_1, x : T \backslash p, \Gamma_2 + x : p = \Gamma_1, x : T, \Gamma_2}$$

$$\frac{\Gamma + x_1 : p_1 = \Gamma' \qquad \Gamma' + x_2 : p_2 + \ldots + x_n : p_n = \Gamma''}{\Gamma + x_1 : p_1 + \ldots + x_n : p_n = \Gamma''}$$

$$\frac{\mathtt{class}\ c\,\{f_1 : p_1; \ldots; f_n : p_n; \ldots\} \in CT}{\mathsf{fields}(c) = f_1 : p_1, \ldots, f_n : p_n}$$

$$\frac{\mathtt{class}\ c\,\{\ldots u_0 p_0\ m(x_1 : u_1 p_1, \ldots, x_n : u_n p_n) : T = M \ldots\} \in CT}{\mathsf{mtype}(c.m) = u_0 p_0 (u_1 p_1, \ldots, u_n p_n) \to T}$$

Figure 7. Helper judgments for Figure 6

of unique exposuse blocks and assume that they still hold at the beginning of the next exposure.

To show that class invariants are preserved during evaluation we first re-state the well-formedness condition for class invariants mentioned in Section 3.

Definition 1 (Well-formed invariant)**.** *A class invariant is* well-formed *if every field it references is either one of the class's own instance fields or is (transitively) reachable through field permissions.*

Note that this definition trivially permits invariants over a class's own fields. This leads to the following heap invariant that we wish to preserve:

Invariant 1 (Heap invariant)**.** *During evaluation, for every object o, either its invariant* $\mathrm{inv}(o)$ *holds or it is exposed with a* unique *permission.*

Finally, we can state that this heap invariant is preserved under evaluation:

Proposition 1 (Invariant preservation)**.** *For every object o, if* $\mathrm{inv}(o)$ *is* well-formed *and* $\mathrm{inv}(o)$ *is proved when o is constructed and at the end of every* unique *exposuse block for o, then Invariant 1 is preserved during every evaluation step.*

Proof. By induction over small-step evaluation judgment with case analysis on last rule applied.

- *New object o*: The class invariant must be proved for the field values provided, establishing $inv(o)$ for the new object o.
- *Entering* unique *exposure of o*: $inv(o)$ holds by assumption.
- *Leaving* unique *exposure of o*: The class invariant must be proved at this point, to establish that $inv(o)$ holds again.
- *Other evaluation rules*:
  - while unique permission for o is exposed: fields mentioned in $inv(o)$ can freely change.
  - otherwise: o's fields cannot be changed because o is not exposed with unique permission (typechecking guarantees this). Every other object mentioned in o's invariant must be reachable transitively from o through field permissions (because the invariant is well-formed). None of these objects can therefore be exposed with unique permissions because some permission to them is held by another field. Therefore, none of these objects' fields can change, preserving o's invariant.

□

As we can see, the soundness argument for the approach presented in this paper is ultimately quite simple and relies on our symbolic permission type system to only allow field modifications for objects whose unique permission is exposed. As long as an object's unique permission is not exposed it must hold the field permissions declared for its fields, which in turn guarantees that none of the objects referenced through fields change during that time. This, finally, means that class invariants will continue to hold for all objects while their unique permission is not exposed as long as these class invariants are *well-formed*.

5. JavaSyp: Implementation for Java

JavaSyp contains a prototype implementation of the permission tracking algorithm shown in Figure 6 for the Java programming language enriched with Java 5 annotations as shown in Section 2. In a linear pass it assigns "effective permissions" to every program expression. JavaSyp also derives verification conditions (VCs) from programmer-defined JML contracts and the semantics of program instructions and invokes a theorem prover whenever a precondition, invariant or "assert" statement has to be proved (see Section 3). JavaSyp encodes VCs according to the SMT-LIB standard [Barrett et al. 2010] and currently uses Z3 [de Moura and Bjørner 2008] to discharge VCs.

JavaSyp and is available open-source [4]. JavaSyp is a plugin to the Eclipse development environment[5] and is implemented as a visitor over Eclipse's AST. The Crystal[6] static analysis framework is used to help parsing annotations and provide feedback to the developer where permissions are missing or assertions could not be proved. JML support is currently based on the JML4 project[7] with some extensions (details on the JavaSyp website). JavaSyp can also be used without JML4 and to this end supports its own, pure-Java declarations of method contracts and invariants. Examples are available with JavaSyp and include the array list example shown in this paper.

JavaSyp verifies the code shown in Figures 2, 3, and 4 in about 1.9 seconds on a 2GHz Intel Core Duo laptop with 3GB RAM running Eclipse 3.5.2 and Java 1.6_18 (x86); 1.4 seconds of those are spent in the theorem prover, Z3 2.7 (x64). The following subsections touch on features of JavaSyp that appear in the array list example but haven't been discussed so far.

5.1 Constructors

JavaSyp handles Java constructors soundly at the price of currently not allowing method calls in constructors that require a receiver permission. This is achieved by treating constructor bodies as "expose" blocks for an injected unique receiver permission. Constructors hence cannot use receiver permissions for calling methods and can also not consume a receiver permission. Without an explicit annotation, constructors therefore always return a unique permission, which is consistent with the T-NEW rule in Figure 6. Method calls in constructors could for instance be allowed with the use of delayed types [Fähndrich and Xia 2007].

5.2 Arrays

Arrays are handled in JavaSyp just like fields are handled in Figure 6 (that is, without "expose" blocks): Storing into an array cell is allowed with unique permission to the array. All elements of an array can uniformly carry a permission that can be specified with the optional elems attribute in JavaSyp's permission annotations. For instance, `@Excl(elems = "excl") Object[]` defines a unique permission to an array holding unique permissions to objects.

This declaration can in fact be used for a more sophisticated array list implementation that keeps track of permissions for elements as well. Capture/release can be used to make element permissions temporarily available to iterator clients. JavaSyp supports *forall* and *exists* quantifiers that can be used to encode properties of array contents.

Assigning to an array cell will consume the permission needed for the cell. Reading from an array cell captures

[4] http://code.google.com/p/syper/

[5] http://www.eclipse.org

[6] http://code.google.com/p/crystalsaf

[7] http://sourceforge.net/apps/trac/jmlspecs/wiki/JML4

the permission used to access the array until the resulting value releases it (if the value is borrowed); thus, reading and writing array cells is similar to field reads and writes in Figure 6.

6. Related Work

Program verification research has a history too long to recount here. Over the last decade, automated program verification for mainstream programming languages has made very exciting advances with comprehensive systems such as ESC/Java [Flanagan et al. 2002] and Spec# [Barnett et al. 2004]. Methodologies based on the owners-as-modifiers paradigm have been key to achieving soundness [Barnett et al. 2004; Müller 2002]. These advances have greatly benefitted from the performance gains of first-order logic SMT solvers [Barrett et al. 2010] at the same time [e.g., de Moura and Bjørner 2008]. Reasoning about concurrent modifications in iterators over a collection is difficult in these systems because no single iterator can claim ownership of the collection. The "friends" methodology offers a potential solution [Barnett and Naumann 2004], but it has to my knowledge never been implemented. Iterators and collections can be made "peers" with universe types [Dietl and Müller 2005], but in Dietl and Müller [2005] it appears that this does not preclude the possibility that collections are modified while iterated, allowing concurrent modification exceptions at runtime in verified code.

This paper takes advantage of fast SMT solvers to reason about program behavior while using symbolic permissions for achieving soundness under aliasing. Permissions can express heap structures incompatible with ownership, such as iterators over collections, and do not impose an overall tree-like heap structure (as ownership does). Ownership systems also have considerable trouble allowing "ownership transfer" [Müller and Rudich 2007]—Spec #, for instance, does not implement it—while that's trivial to do with permissions. (Ownership transfer is possible, but transferring permissions from one owner to another is simpler compared to ownership transfer.) Spec# enforces ownership as part of the verification conditions it discharges, while this paper enforces permissions with a typechecker (universe types are enforced with a typechecker as well).

Separation logic [Reynolds 2002] and other substructural logics have also received a lot of attention as reasoning frameworks for program verification. These "resource-aware" logics essentially build aliasing control directly into the logic, which enables elegant manual proofs of program correctness even in object-oriented languages [Parkinson and Bierman 2008]. Promising inroads into automated verification have been made, but to date automation seems to rely on custom provers [Calcagno et al. 2009; Smans et al. 2009] and can involve significant additional input from the programmer [Jacobs and Piessens 2008], neither of which has been shown to scale.

Symbolic permissions as proposed in this paper are inspired by fractional permissions [Boyland 2003] for separation logic [Bornat et al. 2005] but separate permission tracking into a type system, easing program verification and allowing the use of SMT solvers. As a corollary, this paper allows programmers to write their assertions about program behavior (behavioral contracts) in first-order logic, which is arguably more familiar to most programmers.

Because of the aliasing flexibility they offer, fractional permissions [Boyland 2003] have seen a variety of uses for verifying program behavior [Bierhoff and Aldrich 2007], ensuring the absence of race conditions [Heule et al. 2011; Terauchi and Aiken 2008; Zhao 2007], or both [Beckman et al. 2008]. The author found them well-suited for reasoning about a variety of existing open-source programs [Bierhoff et al. 2009], but automated reasoning about fractions is stubbornly difficult by itself: fraction inference [Terauchi 2008] and fraction polymorphism [Bierhoff 2009; Heule et al. 2011] require substantial engineering and computational resources before one can even start proving assertions about the program. Without inference or polymorphism, however, programmers are left with annotating their programs with concrete fractions such as 0.5, which is not only awkward but also hard to maintain: changing the signature of one function can easily have ripple effects through an entire program.

This paper proposes using symbolic permissions, which the paper shows can be checked with a linear-scan refinement typechecker. Symbolic permissions are simpler to reason about than fractions and save programmers from writing concrete fractions. The proposed typechecker supports the notions of borrowing as well as capture & release [Aldrich et al. 2002; Boyland and Retert 2005; Fähndrich and DeLine 2002], which the author previously found to be critical for capturing realistic programs [Bierhoff et al. 2009]. Note that Plural, the tool used in the author's previous work, can verify clients of interfaces using capture and release, but Plural is currently not equipped to verify that permissions are properly captured and released by an implementing class, while JavaSyp is able to do so. Independently, Saini et al. [2010] have also developed the idea of symbolic permissions as the basis of typestate-based programming, although their calculus does not address program verification and does not include borrowing or capture & release. Program verification in this paper happens separately from permission checking and can mostly ignore aliasing once permissions are checked.

7. Conclusion

This paper introduces symbolic permissions for lightweight automated program verification in sequential programs. The paper shows that permission tracking can be separated into a linear refinement type system that allows program verification to mostly ignore aliasing challenges: specifically, vari-

ables, fields, and array cells with permissions cannot change during method invocations. This allows the use of SMT solvers for doing the heavy verification lifting while keeping verification conditions free of predicates for controlling aliasing. A prototype implementation, JavaSyp, illustrates this approach for Java and is able to verify a conventional array list, an iterator implementation over that array list, as well as clients of these iterators to guarantee the absence of concurrent modification exceptions and other common errors in using this data structure such as accessing a list element that does not exist.

The symbolic permission type system includes two programming idioms, called *borrowing* and *capture & release*, that are needed to reason about common programs. More such common idioms could conceivably be added. Likewise, this paper limits itself to unique and immutable permissions, and other kinds of permissions such as the ones I have proposed previously [Bierhoff and Aldrich 2007]—i.e., full, share, and pure—could be supported easily.

Symbolic permissions are also compatible with race condition avoidance as has been done with fractional permissions [Boyland 2003; Heule et al. 2011; Terauchi and Aiken 2008]. In particular, fork-join-style multi-threading can be seen as the forked thread capturing permissions in scope and releasing them upon joining with the parent thread. In Java and C#, this is particularly apparent as threads are represented as objects themselves. Additionally,verifying concurrent programs seems to require havocking unprotected heap portions after every program instruction [Beckman 2010; Beckman et al. 2008].

While the annotation overhead shown in this paper is significant, the principle of annotating (relevant) variable and field declarations makes for very transparent and easy-to-implement permission tracking, unlike in my and others' previous, more inference-based approaches [Bierhoff et al. 2009; Terauchi 2008]. I will note that permission annotations appear in strictly fewer places than type annotations required in Java (because primitives need not be annotated), but inferring annotations certainly seems desirable where it doesn't confuse programmers.

This paper attempts to make things simpler: symbolic permissions are much easier to track than fractional permissions, and the paper shows that permissions simplify automation of program verification while giving programmers simple tools to work with. I believe we are only at the beginning of exploring what permissions can do for automated reasoning about programs, and this paper attempts to provide a glimpse at what we can do with them.

Acknowledgments

The author thanks Ciera Jaspan and Nels Beckman for valuable feedback on earlier drafts of this paper.

References

J. Aldrich, V. Kostadinov, and C. Chambers. Alias annotations for program understanding. In *ACM Conference on Object-Oriented Programming, Systems, Languages & Applications*, pages 311–330, Nov. 2002.

M. Barnett and D. A. Naumann. Friends need a bit more: Maintaining object invariants over shared state. In *Mathematics of Program Construction*, pages 54–84. Springer, 2004.

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, June 2004.

C. Barrett, A. Stump, and C. Tinelli. *The SMT-LIB Standard, Version 2.0*, Mar. 2010. URL `http://goedel.cs.uiowa.edu/smtlib/`.

N. E. Beckman. *Types for Correct Concurrent API Usage*. PhD thesis, Carnegie Mellon University, Dec. 2010.

N. E. Beckman, K. Bierhoff, and J. Aldrich. Verifying correct usage of Atomic blocks and typestate. In *ACM Conference on Object-Oriented Programming, Systems, Languages & Applications*, pages 227–244, Oct. 2008.

K. Bierhoff. *API Protocol Compliance in Object-Oriented Software*. PhD thesis, Carnegie Mellon University, Apr. 2009.

K. Bierhoff and J. Aldrich. Modular typestate checking of aliased objects. In *ACM Conference on Object-Oriented Programming, Systems, Languages & Applications*, pages 301–320, Oct. 2007.

K. Bierhoff and J. Aldrich. Permissions to specify the composite design pattern. In *7th International Workshop on Specification and Verification of Component-Based Systems*, Nov. 2008.

K. Bierhoff and C. Hawblitzel. Checking the hardware-software interface in Spec#. In *4th Workshop on Programming Languages and Operating Systems*. ACM Digital Library, Oct. 2007. doi: http://doi.acm.org/10.1145/1376789.1376802.

K. Bierhoff, N. E. Beckman, and J. Aldrich. Practical API protocol checking with Access Permissions. In *European Conference on Object-Oriented Programming*, pages 195–219. Springer, July 2009.

E. Bodden, L. Hendren, and O. Lhoták. A staged static program analysis to improve the performance of runtime monitoring. In *European Conference on Object–Oriented Programming*, pages 525–549. Springer, 2007.

R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission accounting in separation logic. In *ACM Symposium on Principles of Programming Languages*, pages 259–270, Jan. 2005. doi: http://doi.acm.org/10.1145/1047659.1040327.

J. Boyland. Checking interference with fractional permissions. In *International Symposium on Static Analysis*, pages 55–72. Springer, 2003.

J. T. Boyland and W. Retert. Connecting effects and uniqueness with adoption. In *ACM Symposium on Principles of Programming Languages*, pages 283–295, Jan. 2005.

C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. In *ACM Symposium on Principles of Programming Languages*, pages 289–300. ACM, Jan. 2009. ISBN 978-1-60558-379-2. doi: http://doi.acm.org/10.1145/1480881.1480917.

L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology*, 4(8):5–32, 2005. URL `http://www.jot.fm/issues/issues 2005 10/article1`.

M. Fähndrich and R. DeLine. Adoption and focus: Practical linear types for imperative programming. In *ACM Conference on Programming Language Design and Implementation*, pages 13–24, June 2002.

M. Fähndrich and S. Xia. Establishing object invariants with delayed types. In *ACM Conference on Object-Oriented Programming, Systems, Languages & Applications*, pages 337–350, Oct. 2007. ISBN 978-1-59593-786-5. doi: http://doi.acm.org/10.1145/1297027.1297052.

C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. Saxe, and R. Stata. Extended static checking for Java. In *ACM Conference on Programming Language Design and Implementation*, pages 234–245, May 2002.

C. Haack and C. Hurlin. Resource usage protocols for iterators. In *International Workshop on Aliasing, Confinement and Ownership*, July 2008.

S. Heule, K. R. M. Leino, P. Müller, and A. J. Summers. Fractional permissions without the fractions. In *Workshop on Formal Techniques for Java-like Programs*, July 2011.

A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *ACM Conference on Object-Oriented Programming, Systems, Languages & Applications*, pages 132–146, 1999.

B. Jacobs and F. Piessens. The VeriFast program verifier. Technical Report CW-520, Department of Computer Science, Katholieke Universiteit Leuven, Aug. 2008.

N. Krishnaswami. Reasoning about iterators with separation logic. In *5th International Workshop on Specification and Verification of Component-Based Systems*, pages 83–86. ACM Press, Nov. 2006.

G. T. Leavens, A. L. Baker, and C. Ruby. JML: A notation for detailed design. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, Boston, 1999.

B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16 (6):1811–1841, Nov. 1994.

P. Müller. *Modular Specification and Verification of Object-Oriented Programs*. Springer, 2002.

P. Müller and A. Rudich. Ownership transfer in Universe types. In *ACM Conference on Object-Oriented Programming, Systems, Languages & Applications*, pages 461–478, Oct. 2007. ISBN 978-1-59593-786-5. doi: http://doi.acm.org/10.1145/1297027.1297061.

M. J. Parkinson and G. M. Bierman. Separation logic, abstraction and inheritance. In *ACM Symposium on Principles of Programming Languages*, pages 75–86, Jan. 2008.

G. Ramalingam, A. Warshavsky, J. Field, D. Goyal, and M. Sagiv. Deriving specialized program analyses for certifying component-client conformance. In *ACM Conference on Programming Language Design and Implementation*, pages 83–94, 2002. ISBN 1-58113-463-0. doi: http://doi.acm.org/10.1145/512529.512540.

J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *IEEE Symposium on Logic in Computer Science*, pages 55–74, 2002.

D. Saini, J. Sunshine, and J. Aldrich. A theory of Typestate oriented programming. In *Workshop on Formal Techniques for Java-like Programs (FTfJP)*, 2010.

J. Smans, B. Jacobs, and F. Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In *European Conference on Object-oriented Programming*, pages 148–172. Springer, July 2009.

T. Terauchi. Checking race freedom via linear programming. In *ACM Conference on Programming Language Design and Implementation*, pages 1–10, June 2008.

T. Terauchi and A. Aiken. A capability calculus for concurrency and determinism. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30(5):1–30, Aug. 2008. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/1387673.1387676.

Y. Zhao. *Concurrency Analysis Based on Fractional Permission System*. PhD thesis, University of Wisconsin-Milwaukee, Aug. 2007.