Verification of Equivalent-Results Methods

K. Rustan M. Leino and Peter Müller

Microsoft Research, Redmond, WA, USA
{leino,mueller}@microsoft.com

Abstract. Methods that query the state of a data structure often return identical or equivalent values as long as the data structure does not change. Program verification depends on this fact, but it has been difficult to specify and verify such equivalent-results methods and their callers.
This paper presents an encoding from which one can determine equivalent-results methods to be deterministic modulo a user-defined equivalence relation. It also presents a technique for checking that a query method returns equivalent results and enforcing that the result depends only on a user-defined influence set.
The technique is general, for example it supports user-defined equivalence relations based on $Equals$ methods and it supports query methods that return newly allocated objects. The paper also discusses the implementation of the technique in the context of the Spec# static program verifier.

0 Introduction

Computer programs contain many methods that query the state of a data structure and return a value based on that state. As long as the data structure remains unchanged, one expects different invocations of the query method to produce equivalent return values. For methods returning scalar values, the return values are expected to be the same. For methods returning object references, the most interesting equivalences are reference equality and equivalence based on the $Equals$ method.

A simple and common example of a query method is the $Count$ method of a collection class, like $List$ in Fig. 0, where for a given collection the method returns the number of elements stored in the collection. Obviously, one expects $Count$ to return identical values when called twice on the same collection. Another example is shown in the $Calendar$ class in Fig. 2, where invocations of the $GetEarliestAppointment$ will yield equivalent results as long as the state of the calendar does not change. However, since $GetEarliestAppointment$ returns a newly allocated object, the results will not be identical. Due to object-allocation, query methods cannot be expected to be deterministic. Nevertheless, their results are expected to be equivalent. Therefore, we shall refer to such query methods as *equivalent-results methods*.

Query methods (also called *pure methods*) are particularly important in assertion languages such as JML [15] or Spec# [1] because they allow assertions to be expressed in an abstract, implementation-independent way. For instance, $Count$ is used in the precondition of $GetItem$ (Fig. 0) to refer to the number of elements in the list without revealing any implementation details. However, reasoning about assertions that contain query methods is difficult. The client program in Fig. 1 illustrates the problem. It uses a

```
class List⟨T⟩ {
  int Count()
    ensures 0 ⩽ result;
  { ... }

  T GetItem(int n)
    requires 0 ⩽ n < Count();
  { ... }
  ⋮
}
```

Fig. 0. A *List* class whose *Count* method returns the number of elements in a given list and whose *GetItem* method returns a requested element of the list. The postcondition of *Count* promises the return value to be non-negative, and the precondition of *GetItem* requires parameter n to be less than the value returned by *Count*.

```
List⟨T⟩ list;
⋮
if (n < list.Count()) {
  S   // some statement that changes the state, but not the list
  t = list.GetItem(n);
}
```

Fig. 1. A code fragment that uses the *List* class from Fig. 0. The if statement guards the invocation of *GetItem* to ensure that *GetItem*'s precondition is met. To verify the correctness of this code, one needs to be able to determine that the two invocations of *Count* return the same value.

conditional statement to establish the precondition of *GetItem*. We assume that statement S does not change the list structure. Therefore, we expect that the condition still holds when *GetItem* is called, that is, that the two calls to *Count* yield the same result. There are essentially three approaches for a program verifier to conclude this fact.

The first approach is to require that the postcondition of the query method is strong enough for a caller to determine exactly what value is returned. Typically, this can be achieved by having a postcondition of the form $result = E$. In our example, this postcondition would allow the verifier to compare the state affected by S to the state read by E to determine whether the two calls to *Count* return the same result. However, requiring such strong postconditions may entail a dramatic increase in the complexity of the specification. For *Count*, one would have to axiomatize mathematical lists and use that mathematical abstraction in the specification of the *List* class. We consider this burden too high, in particular for the verification of rather simple properties.

The second approach is to define the return value of the method to be a function of the program state. If the program state has not changed by the time the method is invoked again, this approach allows one to conclude the return value is the same as before. But this approach is too brittle, for two reasons. First, it treats state changes too coarsely. For example, statement S in Fig. 1 may change the program state, but as long as it does not change the state of the list, we want to be able to conclude that the result

of $Count$ is unchanged. Second, this approach is too precise about the return value. For example, the object references returned by two calls to $GetEarliestAppointment$ in Fig. 2 are not identical, yet the data they reference are equivalent. Queries that return newly allocated objects are very common, especially in JML's model classes [16].

The third approach is to require that all query methods used in specifications are equivalent-results methods whose results depend only on certain heap locations. We call this set of locations the *influence set* of a query method. With this approach, the code in Fig. 1 can be verified by showing that the locations modified by S are not in the influence set of $Count$. From the equivalent-results property and the fact that $Count$ returns an integer, we can conclude that the two calls to $Count$ yield the same results.

Existing program verifiers such as the Spec# static program verifier Boogie [0] and ESC/Java2 [14] apply the third approach. However, these systems do not enforce that query methods actually are equivalent-results methods and that their result actually depends only on the declared influence set. Blindly assuming these two properties is unsound. Checking the properties is not trivial, even for methods that return scalar values. For instance, $GetHashCode$ is an equivalent-results method and should be permitted in assertions, but returning the hash code of a newly allocated object leads to nondeterminism and must be prevented.

In this paper, we present a simple technique to check that a query method is an equivalent-results method and that its result depends only on its parameters and the declared influence set. This technique supports user-defined equivalence relations based on, for instance, $Equals$ methods. We use self-composition [2, 20] to simulate two executions of the method body from start states that coincide in the influence set and to prove that the respective results are indeed equivalent. We also present axioms that enable reasoning about equivalent-results methods and argue why they are sound. Our technique is very general: it supports user-defined equivalence relations, it does not require a particular way of specifying influence sets, and it uses a relaxed notion of purity. In particular, implementations of query methods may use non-deterministic language features and algorithms, and may return newly allocated objects. We plan to implement our technique for pure methods in Boogie, but our results do not rely on the specifics of Spec#. Therefore, they can be adopted by other program verifiers.

Outline. Section 1 provides the background on program verification that is needed in the rest of this paper. Section 2 presents an encoding of equivalent-results methods that enables the kind of reasoning discussed above. Section 3 explains our technique for checking the equivalence of results. Section 4 discusses the application of our technique to Spec#. The remaining sections summarize related work and offer conclusions.

1 Background on Program Verification

In this section, we review details of program verification relevant to our paper. For a more comprehensive and tutorial account of this material, we refer to some recent Marktoberdorf lecture notes [19].

```
class Appointment {
  int time;
  // ... more fields here

  pure override bool Equals(object o)
    ensures GetType() = typeof(Appointment) ⇒
        (result ⟺
            o ≠ null ∧ GetType() = o.GetType() ∧
            time = ((Appointment)o).time ∧ ... more comparisons here);
  { ... }
}

class Calendar {
  pure Appointment GetEarliestAppointment(int day) {
    Appointment a;
    // find earliest appointment on day day
    ...
    return a.Clone();
  }

  void ScheduleMorningMeeting(int day, List⟨Person⟩ invitees)
    requires 10 ⩽ GetEarliestAppointment(day).time;
  { ... }
}

class Person {
  void Invite(Calendar c, ...) {
    if (10 ⩽ c.GetEarliestAppointment(5).time) {
      // compute invitees
      List⟨Person⟩ invitees = new List⟨Person⟩();
      while (...) {
        ...
        invitees.Add(p);
      }
      // schedule those invitees
      c.ScheduleMorningMeeting(5, invitees);
    }
  }
}
```

Fig. 2. A *Calendar* program whose *GetEarliestAppointment* method returns an equivalent value as long as the calendar does not change. The correctness of the code fragment at the bottom of the figure depends on that the call to *GetEarliestAppointment* in the precondition of *ScheduleMorningMeeting* returns a value that is equivalent to the one returned by the call to *GetEarliestAppointment* in the guard of the if statement.

Architecture of Program Verifiers. To verify a program, the program's proof obligations (*e.g.*, that preconditions are met) are encoded as logical formulas called *verification conditions*. The verification conditions are valid formulas if and only if the program is correct with respect to the properties being verified. Each verification condition is fed to a theorem prover, such as an SMT solver or an interactive proof assistant, which attempts to ascertain the validity of the formula or construct counterexample contexts that may reveal errors in the source program. As has been noted by several state-of-the-art verifiers, it is convenient to generate verification conditions in two steps: first encode the source program in an intermediate verification language, and then generate input for the theorem prover from the intermediate language [0,11, 4]. Since the second step concerns issues that are orthogonal to our focus in this paper, we look only at the first step. The notation we will use for the intermediate verification language is BoogiePL [0,10]. A BoogiePL program consists of a first-order logic theory, which in particular specifies the heap model of the source language, and an encoding of the source program. We explain these two parts in the following subsections.

Heap Model. We model the heap as a two-dimensional array that maps object identities and field names to values [23], so a field selection expression $o.f$ is modeled as $\$Heap[o, f]$. By making the heap explicit, we correctly handle object aliases, as is well known [3, 23]. In the encoding, we use a boolean field $\$alloc$ in each object to model whether or not the object has been allocated. The subtype relation is denoted by $<:$.

For any set S of locations (that is, of object-field pairs), we define a relation $\equiv_S$ that relates two heaps if they have the same values for all locations in S. More precisely:

$$(\forall\, H, K, S \bullet\ (H \equiv_S K \iff (\forall\, o, f \bullet\ (o, f) \in S \Rightarrow H[o, f] = K[o, f]\ ))\ )$$

Note that $\equiv_S$ is an equivalence relation: it is reflexive, symmetric, and transitive. If $H \equiv_S K$, we say that H and K are *equivalent modulo* S.

We assume that pure methods do not modify the state of any object that is allocated in the pre-state of the method execution. This definition allows a pure method to allocate and modify new objects such as iterators [24]. More precisely, if $H0$ and $H1$ denote the heaps immediately before and after the call to a pure method, and S is a set of locations of objects that are allocated in $H0$, the following property holds:

$$H0 \equiv_S H1 \tag{0}$$

Encoding of Source Programs. Each source-language method is encoded as a procedure in the intermediate verification language. To understand the basic encoding, consider a method M in a class C with a field y, shown in Fig. 3.

The specification of M has a precondition that obligates the callers of M to pass a non-negative argument value. In turn, the precondition lets the implementation of M assume x to be non-negative on entry. The specification also has a modifies clause and a postcondition that obligate the implementation to make sure that its return value, parameter x, and the y field of the method's receiver object are related as specified, and to modify only $\mathbf{this}.y$. A caller can assume these properties upon return of a call.

```
class C {
  int y;
  int M(int x)
    requires 0 ⩽ x;
    modifies this.y;
    ensures result + x ⩽ this.y;
  { ... }
```

Fig. 3. An example class in the source language, showing an instance field y and a method M with a method specification.

```
procedure C.M(this, x) returns (result);
  requires this ≠ null;
  free requires $Heap[this, $alloc] ∧ $typeof(this) <: C;
  ensures result + x ⩽ $Heap[this, C.y];
  ensures (∀ o, f • o ≠ this ∧ old($Heap)[o, $alloc] ⇒
        $Heap[o, f] = old($Heap)[o, f] ∨ (o = this ∧ f = y) );
  free ensures (∀ o • old($Heap)[o, $alloc] ⇒ $Heap[o, $alloc] );
```

Fig. 4. A BoogiePL procedure declaration that encodes the signature and specification of the example method $C.M$.

A representative encoding of M as a BoogiePL procedure is shown in Fig. 4. The procedure declaration makes the implicit receiver parameter **this** explicit, and the anonymous return value is encoded as a named out-parameter. The types in BoogiePL are more coarse-grained than those in the source language, and for the purposes of this paper, they are only a distraction, so we omit them altogether. Three things are worth noting about the procedure specification.

First, method M's pre- and postconditions have direct analogs in the BoogiePL procedure, where the implicit dereferencing of the heap in a field selection expression is made explicit in the BoogiePL encoding.

Second, the method's modifies clause is encoded as a BoogiePL postcondition that dictates which locations in the heap are allowed to change. The latter says that for any non-null object o allocated on entry to the method and for any field f, the heap at location $o.f$ is unchanged except possibly at location $\mathbf{this}.y$.

Third, to verify a program, one often needs to know some properties that are guaranteed by the source language. For example, the static type of the receiver parameter of method M is C and the source-language type checker thus guarantees that the allocated type of the receiver is some subtype of C. The source language also guarantees that all object references in use by a program are allocated and (thanks to the fiction created by the garbage collector) remain allocated forever. To incorporate these guaranteed conditions in the encoding, BoogiePL conveniently offers *free pre- and postconditions* as part of a procedure declaration. Free preconditions are assumed on entry to a procedure implementation, but not checked at call sites, and analogously for free postconditions.

Proof Obligations and Soundness. Proving the correctness of a BoogiePL program amounts to statically verifying that the program does not abort due to a violated assertion (such as a precondition or postcondition). To do that, each assertion is turned into a proof obligation. One can then use an appropriate program logic to show that the assertions hold. For the proof, one may assume the conditions expressed as free preconditions, free postconditions, and explicit **assume** statements. The verification is sound if all of these assumptions actually hold.

2 Encoding of Equivalent-Results Methods

Our idea is to define an equivalence class of return values for each equivalent-results method. We define the equivalence class via a programmer-defined *similarity relation*. Typical choices for the similarity relation are reference equality and the $Equals$ method. Rather than letting the similarity relation be the equivalence relation, we define the equivalence class to be those values that are related by the similarity relation to a particular element, called the *anchor element*. This has the advantage that the similarity relation need not be symmetric and transitive, which in practice the $Equals$ method often is not [25]. Another advantage is that using an anchor element allows us to state axioms that are handled more efficiently by the theorem prover.

In this section, we explain similarity relations, anchor elements, and the influence sets that define the dependencies of method results.

Similarity Relations. For a method M, we let $\mathcal{R}_M(H, r, H', r')$ denote M's similarity relation, relating r whose state is evaluated in heap H and r' whose state is evaluated in heap H'. For example, if $\mathcal{R}_M$ denotes equality of scalar values or reference equality for object values, we have:

$$\mathcal{R}_M(H, r, H', r') \iff r = r' \tag{1}$$

and if $\mathcal{R}_M$ uses the $Equals$ method, we have:

$$\mathcal{R}_M(H, r, H', r') \iff @Equals(H, r, H', r') \tag{2}$$

where $@Equals$ is a function automatically generated from the specification of $Equals$. Value r is always a return value of the method; r' is either a return value, in which case $H = H'$ or the anchor element, in which case H' is a special heap $AnchorHeap_M(p)$ where we evaluate anchor elements. The similarity relation defines an equivalence class of values that are related to the anchor element.

For the $Appointment.Equals$ method in Fig. 2, the following axiom is automatically generated for function $@Equals$:

$$\begin{array}{l}(\forall\, H, this, K, o \bullet \\ \quad this \neq \mathbf{null} \wedge \$typeof(this) <: Appointment \wedge \$typeof(o) <: Object \Rightarrow \\ \qquad (@Equals(H, this, K, o) \iff \\ \qquad\quad o \neq \mathbf{null} \wedge \$typeof(this) = \$typeof(o) \wedge \\ \qquad\quad H[this, time] = K[o, time] \wedge \ldots \text{more comparisons here}))\end{array} \tag{3}$$

where, here and throughout, quantifications over H and K range over well-formed heaps. It is not the subject of our paper to describe how axioms for pure methods are described, but see our previous work with Ádám Darvas [9, 8]; the difference is that here we use one heap argument for each of the two parameters to $Equals$.

Influence Sets. The influence set is a set of locations in the heap. Let $\mathcal{F}_M(H, p)$ denote the influence set of M as computed for parameters p in a heap H. Note that the computation of the influence set may depend on the heap. For example, consider a class $Schedule$ with an $Appointment$ field a. Suppose the influence set for some method applied to a schedule s is given by the set of path expressions $\{s.a,\ s.a.time\}$. Viewed in the intermediate-language notation, these path expressions denote the following object-field pairs: $(s, a),\ (\$Heap[s, a], time)$.

We require every influence set to be *self-protecting* [13], which means that any two heaps equivalent modulo the influence set compute the influence set the same way:

$$(\,\forall\, H, K, p \bullet \quad H \equiv_{\mathcal{F}_M(H,p)} K \quad \Rightarrow \quad \mathcal{F}_M(H, p) = \mathcal{F}_M(K, p)\,) \tag{4}$$

Self-protection can be enforced by requiring the set of path expressions that specify the influence set to be prefix closed: if it contains a path expression $E.x.y$, then it must also contain the path expression $E.x$. Therefore, the expression $E.x.y$ denotes the same location in heaps H and K.

The influence set specifies which parts of the program state are allowed to influence the return value. To a first order of approximation, the influence set is the *read set* or *read effect* of the method [5], but, technically, we actually allow methods to read any part of the state, as long as the values of things outside the influence set have no bearing on the return value.

Anchor Elements. The encoding of equivalent-results methods has to allow us to prove that two calls to an equivalent-results method M return equivalent results if the two heaps before the calls are equivalent modulo the influence set of M. We reach this conclusion in two steps. First, we encode by an axiom that the anchor element remains the same as long as the program state indicated by the influence set does not change. Second, we encode by a free postcondition that the actual return value of M is related to the anchor element by the similarity relation. Hence, the results of the two calls to M are in the same equivalence class.

Step A: In our intermediate-language encoding, we introduce a function $Anchor_M$ that yields an anchor element for the equivalence class of the return values of M. We axiomatize $Anchor_M$ as follows:

$$(\,\forall\, p, H, K \bullet \quad H \equiv_{\mathcal{F}_M(H,p)} K \quad \Rightarrow \quad Anchor_M(H, p) = Anchor_M(K, p)\,) \tag{5}$$

The axiom says that we pick the same anchor element whenever M is invoked with the same arguments p in two heaps H and K that are equivalent modulo $\mathcal{F}_M(H, p)$. In other words, the anchor element is a function of the program state projected onto the influence set.

```
H0 := $Heap;
call r := GetEarliestAppointment(c, 5);
H1 := $Heap;
if (10 ⩽ $Heap[r, time]) {
  // code to compute invitees ...
  K0 := $Heap;
  call r' := GetEarliestAppointment(c, 5);
  K1 := $Heap;
  assert 10 ⩽ $Heap[r', time];
  ...
}
```

Fig. 5. A sketch of the code fragment from the bottom of Fig. 2, giving the names $H0$, $H1$, $K0$, and $K1$ to the intermediate values of the heap, and giving the names r and r' to the return values of the two calls to $GetEarliestAppointment$. The assert statement at the end shows the condition that we want to prove.

Step B: We add to our encoding the following free postcondition:

$$\textbf{free ensures } \mathcal{R}_M(\$Heap, result, AnchorHeap_M(p), Anchor_M(\$Heap, p)); \tag{6}$$

To make sure the anchor object always denotes the same equivalence class, we evaluate its state in a special, constant heap $AnchorHeap_M$. We postpone until Section 3 how to justify this free postcondition.

Example. To prove the correctness of method $Invite$ in Fig. 2, it suffices to show that the two invocations of $GetEarliestAppointment$ return equivalent values. Recall, the second invocation takes place during the evaluation of the precondition of $ScheduleMorningMeeting$. Fig. 5 shows a BoogiePL encoding of that fragment. As illustrated by the assert statement in Fig. 5, we wish to prove that $H1[r, time]$ equals $K1[r', time]$.

The influence set of $GetEarliestAppointment$ contains the fields that make up the representation of the $Calendar$ object. Let $H0$ and $H1$ denote the heaps immediately before and after the first call to $GetEarliestAppointment$, and let $K0$ and $K1$ denote the heaps immediately before and after the second call.

Since $GetEarliestAppointment$ is pure, it does not change the values of any previously allocated locations (see condition (0)), so $H0$ and $H1$ are equivalent modulo $\mathcal{F}(H0, c, 5)$, and $K0$ and $K1$ are equivalent modulo $\mathcal{F}(K0, c, 5)$ (we drop the subscript $GetEarliestAppointment$ in this example). Assuming that the code that computes $invitees$ has no effect on the values of the locations in the influence set, we also have that $H1$ and $K0$ are equivalent modulo $\mathcal{F}(H1, c, 5)$. By self-protection (4), we know that the three influence sets are equal. Thus, we can conclude by transitivity:

$$H1 \equiv_{\mathcal{F}(H1,c,5)} K1 \tag{7}$$

By axiom (5) and equation (7), we conclude that the anchor elements for the two calls are the same:

$$Anchor(H1, c, 5) = Anchor(K1, c, 5) \tag{8}$$

```
procedure M(p) returns (result)
  requires P($Heap, p);
  free requires Q($Heap, p);
  ensures S(old($Heap), $Heap, p, result);
  free ensures T(old($Heap), $Heap, p, result);
  free ensures R_M($Heap, result, AnchorHeap_M(p), Anchor_M($Heap, p));
{
  var locals;
  Body
}
```

Fig. 6. A procedure in the intermediate verification language, illustrating the general form of the procedure into which the method translates.

Now let r and r' denote (as indicated in Fig. 5) the values returned by the two calls to $GetEarliestAppointment$. The similarity relation is given by the $Equals$ method. Thus, we conclude from postcondition (6):

$$@Equals(H1,\ r,\ AnchorHeap(c,5), Anchor(H1,c,5)) \quad \text{and}$$
$$@Equals(K1,\ r',\ AnchorHeap(c,5), Anchor(K1,c,5))$$

By axiom (3) and property (8), we have

$$H1[r, time] = AnchorHeap(c,5)[Anchor(H1,c,5), time] \wedge$$
$$K1[r', time] = AnchorHeap(c,5)[Anchor(H1,c,5), time]$$

from which we conclude $H1[r, time] = K1[r', time]$, as required to establish the precondition of the call to $ScheduleMorningMeeting$.

3 Verifying Equivalence of Results

As we mentioned in Section 1, soundness of a verification system comes down to justifying every assumption that the proof system allows a proof to make use of. In the previous section, we introduced three conditions that we used as assumptions in the proof. The first assumption is the axiom of self-protection (4). It can be justified by a syntactic check on the path expressions used to define the influence set. The second assumption is the axiom about $Anchor_M$ (5). It is justified on the basis that there exists a function $Anchor_M$ that satisfies the axiom, for example any constant function. The third assumption is the free postcondition (6). In this section, we present a proof technique based on self-composition that justifies this assumption.

Ordinarily, a method M gives rise to a verification condition prescribed by a BoogiePL procedure implementation like procedure M in Fig. 6, where p denotes the in-parameters, $\mathcal{P}$ and $\mathcal{S}$ denote some checked pre- and postconditions, $\mathcal{Q}$ and $\mathcal{T}$ denote some free pre- and postconditions (*cf.* Fig. 4), $locals$ are local variables, and $Body$ is the BoogiePL encoding of the implementation of method M.

For every equivalent-results method M, we will now prescribe a second BoogiePL procedure, whose validity will justify the free postcondition (6). The key idea is to

```
procedure M'(p) returns (result) {
  var locals;

  var $oldHeap := $Heap;
  assume P($Heap, p) ∧ Q($Heap, p);
  Body'
  assume S($oldHeap, $Heap, p, result) ∧ T($oldHeap, $Heap, p, result);

  assume Anchor_M($Heap, p) = result ∧ AnchorHeap_M(p) = $Heap;    // L0

  havoc $Heap, locals, result;
  assume $Heap ≡_{F_M($oldHeap,p)} $oldHeap;

  $oldHeap := $Heap;
  assume P($Heap, p) ∧ Q($Heap, p);
  Body'
  assume S($oldHeap, $Heap, p, result) ∧ T($oldHeap, $Heap, p, result);

  assert R_M($Heap, result, AnchorHeap_M(p), Anchor_M($Heap, p));    // L1
}
```

Fig. 7. A procedure that checks by assertion (L1) that M satisfies its free postcondition (6).

execute the method body twice starting in states that agree on the values of the in-parameters and all objects in the influence set. We then prove that the two executions yield equivalent results. This second procedure has the form shown by M' in Fig. 7 and is described as follows:

- The body of M' starts off with $\$Heap$, $locals$, and $result$ set to arbitrary values, saves the value of $\$Heap$ in $\$oldHeap$, and assumes the preconditions $\mathcal{P}$ and $\mathcal{Q}$.
- It then performs $Body'$, which is $Body$ with occurrences of $\mathbf{old}(\$Heap)$ replaced by $\$oldHeap$ and occurrences of assert statements (*i.e.*, checked conditions) replaced by assume statements. These assume statements are justified by the fact that procedure M already prescribes checks for them, so if the conditions do not hold, the program verifier will generate appropriate errors when attempting to verify M.
- Upon termination of $Body'$, the postconditions $\mathcal{S}$ and $\mathcal{T}$ are assumed. Again, $\mathcal{S}$ can be assumed here because it is checked by M.
- We explain the assume statement (L0) below.
- Next, the code prepares for another execution of $Body'$. The second execution of $Body'$ is to start in a state where all locations of the influence set have the same values as in the first execution. Thus, $\$Heap$, $locals$, and $result$ are set to arbitrary values (using a **havoc** statement) and the value of $\$Heap$ is constrained (using an assume statement) to be equivalent to $\$oldHeap$ modulo the influence set.
- The preconditions are assumed, $Body'$ is executed a second time, and the postconditions are assumed.
- We explain the assert statement (L1) below.

The first half of M' culminates in assume statement (L0), which has the effect of defining $Anchor_M(\$Heap, p)$ and $AnchorHeap_M(p)$ to be the result value and result

heap of an arbitrary execution of the method (namely, the first execution of $Body'$). In fact, by axiom (5), (L0) defines $Anchor_M(\$Heap, p)$ for all heaps that are equivalent to $\$Heap$ modulo the influence set. The second half of M' checks that (6) is indeed a postcondition of the method for all those equivalent heaps.

With that, we have justified all the assumptions that our technique introduces, and thus we have established that our technique is sound.

4 Application to Spec#

In verifying Spec# programs, we have run across scores of examples like the one in Fig. 0, where in Spec# the $Count$ method tends to be a *property getter*, which is a form of parameter-less method. By default, property getters are treated as pure methods that read only the ownership cone of the receiver object. The *ownership cone* of an object is the set of locations that make up the object's representation [6]. Previously, our best solution for dealing with this situation in the Spec# program verifier was to introduce an axiom that says the return value of the method is a function of the ownership cone. But such an axiom is not sound if a pure method returns newly allocated object or values that are derived from such objects. Our technique in this paper gives a sound solution to the problem, and we intend to implement it. In this section, we describe some issues that pertain to the practical implementation of equivalent-results methods in Spec#.

We intend to restrict the choices for $\mathcal{R}_M$ in Spec# to support only the two choices (1) and (2). This will simplify the implementation while supporting the most common similarity relations. (The only other useful similarity we found puts all non-null references in one equivalence class.) To select between the two choices, we will introduce a default choice and a method annotation (a *custom attribute*) that can override the default.

For the influence set, we will only support the union of the ownership cones for some subset of the parameters. Ownership provides a form of abstraction, allowing one to specify influence sets without being specific about implementation details. There is already a notion of *confined* in Spec# that says that a pure method reads the ownership cone of a parameter. Moreover, the Spec# program verifier already has an encoding that lets one deduce, for *valid* objects, whether or not the ownership cone of the object has changed. The encoding is simply to inspect the object's ghost field $snapshot$ [8]. An object is valid when its object invariant holds [18]. Since this is the precondition of almost all methods, we will not attempt to prove ownership cones to be the same other than via the $snapshot$ field. Because of the snapshot encoding, we can write axiom (5) as:

$$\begin{array}{l}(\,\forall\, p, H, K \;\bullet\; H[p, valid] \,\wedge\, K[p, valid] \,\wedge\, H[p, snapshot] = K[p, snapshot] \\ \qquad\qquad \Rightarrow\; Anchor_M(H, p) = Anchor_M(K, p)\;)\end{array}$$

(We have abused notation slightly: by $H[p, valid]$ and $H[p, snapshot]$, we really mean to refer to the $valid$ and $snapshot$ fields of all the parameters in p that contribute to the influence set, and likewise for K.) In fact, there is an alternative way to encode this property that is significantly more efficient for the SMT solver because it avoids quantification over pairs of heaps. The alternative encoding [8] introduces an uninterpreted

function A_M and uses it to more directly say that $Anchor_M(H, p)$ is a function of p and $H[p, snapshot]$:

$$(\forall\, p, H \bullet \;\; H[p, valid] \;\; \Rightarrow \;\; Anchor_M(H, p) = A_M(p, H[p, snapshot])\;)$$

With the restriction to influence sets based on ownership cones and our focus on reasoning about these via snapshots, axiom (4) becomes trivial, so we omit it.

5 Related Work

The Java Modeling Language (JML) requires pure methods to be deterministic [17]. This requirement is not practical since pure methods often need to return newly allocated objects, which is illustrated by many pure methods in JML's model library [16]. Our notion of equivalent-results methods allows pure methods to return newly allocated objects. Since our axioms are based on a user-defined similarity relation such as an $Equals$ method, determinism is not required for soundness.

The axiomatization of pure methods consists of two groups of axioms: method-specific axioms that specify the behavior of each individual method and general axioms that describe common properties of all pure methods. Previous work by Darvas and Müller [9] focuses on the method-specific axioms, but does not discuss the general axioms that we provide in this paper. Their axiomatization is sound, but too weak for many interesting examples. Darvas and Leino [8] present general axioms that are used in the Spec# verifier Boogie. Some of their work assumes that a pure method is deterministic and that its result depends only on a specified influence set, but these assumptions are not checked. Therefore, their axiomatization is unsound for pure methods that return newly allocated objects or whose result depends on locations outside the influence set. Our work eliminates both sources of unsoundness.

Jacobs developed SpecLeuven, a variant of Spec# for multi-threaded programs. In his work [12], *inspector methods* are syntactically enforced to be deterministic, which is sound but overly restrictive. Influence sets are checked by an extension of the Boogie methodology [18], which requires an object to be unpacked before its state is read. Our verification technique based on self-composition does not require any particular methodology.

ESC/Java2 [14, 7] also operates under the unchecked assumption that pure methods are deterministic, which is unsound if they are not. Moreover, since JML specifications typically do not declare an influence set, ESC/Java2 has but limited support for reasoning about the effect of a heap modification on the result of a pure method.

The influence sets we use in this paper are similar to read effects. However, read effects constrain the whole execution of a method, whereas our influence sets only constrain the method result. We allow methods to read arbitrary locations as long as the result depends only on the declared influence set. Clarke and Drossopoulou [5] show how to declare and check read effects in an ownership type system. We use self-composition to verify influence sets, which is in general more fine-grained than type checking and does not require a particular ownership scheme.

Self-composition has been applied to prove secure information flow [2, 20]. In fact, proving that a method result depends only on a specified influence set can be seen as an

instance of secure information flow, where the method result, the method parameters, and the locations in the influence set have a low security level and all other locations have a high security level. In addition to information flow, we use self-composition to prove that two executions of a method yield equivalent results.

Separation logic [21] provides a powerful and elegant way to reason about the effects of heap modifications. The effect of pure methods can be achieved by introducing abstract predicates [22]. The influence set of a pure method corresponds to the footprint of the predicate. The frame rule can be used to show that certain heap modifications do not affect the truth value of the abstract predicate. However, even if pure methods are not used in contracts, the correctness of some programs relies on the equivalent-results property. We believe that our verification technique is also applicable to separation logic in order to verify such programs.

6 Conclusions

In this paper, we introduced the notion of equivalent-results methods and explained their usefulness for program specification: equivalent-results methods are expressive, for instance, they may return newly-allocated objects, and they permit an axiomatization that is sound and strong enough to verify interesting programs. We showed that the equivalent-results property can be checked by an automatic program verifier using self-composition. Our technique is very flexible: it does not require a particular programming methodology, uses a relaxed notion of purity, and even handles non-deterministic language features and algorithms. As future work, we plan to implement our technique in the Spec# verifier Boogie.

Acknowledgments. The idea of using self-composition was inspired by a discussion with Anindya Banerjee. We thank David Naumann and the anonymous reviewers for helpful comments, one of which led to a simplification of Fig. 7.

References

0. M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO*, volume 4111 of *LNCS*, pages 364–387. Springer, 2006.
1. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *CASSIS 2004*, volume 3362 of *LNCS*, pages 49–69. Springer, 2005.
2. G. Barthe, P. R. D'Argenio, and T. Rezk. Secure information flow by self-composition. In *Computer Security Foundations (CSFW)*, pages 100–114. IEEE, 2004.
3. R. M. Burstall. Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence*, 7:23–50, 1972.
4. S. Chatterjee, S. K. Lahiri, S. Qadeer, and Z. Rakamarić. A reachability predicate for analyzing low-level software. In *TACAS*, volume 4424 of *LNCS*, pages 19–33. Springer, 2007.
5. D. G. Clarke and S. Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *OOPSLA*, volume 37(11) of *SIGPLAN Notices*, pages 292–310. ACM, 2002.
6. D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *OOPSLA*, volume 33(10) of *SIGPLAN Notices*, pages 48–64. ACM, 1998.

7. D. Cok. Reasoning with specifications containing method calls and model fields. *Journal of Object Technology*, 4(8):77–103, 2005.
8. Á. Darvas and K. R. M. Leino. Practical reasoning about invocations and implementations of pure methods. In *FASE*, volume 4422 of *LNCS*, pages 336–351. Springer, 2007.
9. Á. Darvas and P. Müller. Reasoning about method calls in interface specifications. *Journal of Object Technology*, 5(5):59–85, 2006.
10. R. DeLine and K. R. M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, Mar. 2005.
11. J.-C. Filliâtre. Why: a multi-language multi-prover verification tool. Research Report 1366, LRI, Université Paris Sud, Mar. 2003.
12. B. Jacobs. *A Statically Verifiable Programming Model for Concurrent Object-Oriented Programs*. PhD thesis, Katholieke Universiteit Leuven, 2007.
13. I. T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In *Formal Methods*, volume 4085 of *LNCS*, pages 268–283. Springer, 2006.
14. J. R. Kiniry and D. R. Cok. ESC/Java2: Uniting ESC/Java and JML. In *CASSIS 2004*, volume 3362 of *LNCS*, pages 108–128. Springer, 2005.
15. G. T. Leavens, A. L. Baker, and C. Ruby. JML: A notation for detailed design. In *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer, 1999.
16. G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. *Science of Computer Programming*, 55(1–3):185–208, Mar. 2005.
17. G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, and J. Kiniry. JML reference manual. Dept. Comp. Sci., Iowa State University. Available from www.jmlspecs.org, 2007.
18. K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In *ECOOP*, volume 3086 of *LNCS*, pages 491–516. Springer, 2004.
19. K. R. M. Leino and W. Schulte. A verifying compiler for a multi-threaded object-oriented language. In *2006 Marktoberdorf Summer School on Programming Methodology*, NATO ASI Series. Springer, 2007. To appear. Available at research.microsoft.com/~leino/papers.html.
20. D. A. Naumann. From coupling relations to mated invariants for checking information flow. In *ESORICS*, volume 4189 of *LNCS*, pages 279–296. Springer, 2006.
21. P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *POPL*, pages 268–280, 2004.
22. M. J. Parkinson and G. M. Bierman. Separation logic and abstraction. In *POPL*, pages 247–258. ACM, 2005.
23. A. Poetzsch-Heffter and P. Müller. Logical foundations for typed object-oriented languages. In *Programming Concepts and Methods (PROCOMET)*, pages 404–423, 1998.
24. A. Salcianu and M. Rinard. Purity and side effect analysis for java programs. In *VMCAI*, volume 3385 of *LNCS*, pages 199–215. Springer, 2005.
25. D. E. Stevenson and A. T. Phillips. Implementing object equivalence in Java using the template method design pattern. *SIGCSE Bulletin*, 35(1):278–282, 2003.