

**Implicit Dynamic Frames:
Combining dynamic frames and
separation logic (soundness proof)**

Jan Smans

Bart Jacobs

Frank Piessens

Report CW 542, June 2009



Katholieke Universiteit Leuven
Department of Computer Science

Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Implicit Dynamic Frames: Combining dynamic frames and separation logic (soundness proof)

Jan Smans

Bart Jacobs

Frank Piessens

Report CW 542, June 2009

Department of Computer Science, K.U.Leuven

Abstract

The dynamic frames approach has proven to be a powerful formalism for specifying and verifying object-oriented programs. However, it requires writing and checking many frame annotations. In this paper, we propose a variant of the dynamic frames approach that eliminates the need to explicitly write and check frame annotations.

In this paper, we improve upon the classical dynamic frames approach in two ways: (1) method contracts are more concise and (2) fewer proof obligations must be discharged by the verifier. We have proven soundness, implemented the approach in a verifier prototype and demonstrated its expressiveness by verifying several challenging examples from related work.

Implicit Dynamic Frames: Combining dynamic frames and separation logic (soundness proof)

Jan Smans Bart Jacobs Frank Piessens

Abstract

The dynamic frames approach has proven to be a powerful formalism for specifying and verifying object-oriented programs. However, it requires writing and checking many frame annotations. In this paper, we propose a variant of the dynamic frames approach that eliminates the need to explicitly write and check frame annotations.

In this paper, we improve upon the classical dynamic frames approach in two ways: (1) method contracts are more concise and (2) fewer proof obligations must be discharged by the verifier. We have proven soundness, implemented the approach in a verifier prototype and demonstrated its expressiveness by verifying several challenging examples from related work.

1 Introduction

In the dynamic frames approach [2, 22, 21, 1, 26, 43, 46], the programmer specifies upper bounds on the locations that can be read or written by a method in terms of expressions denoting sets of locations. To preserve information hiding, these expressions can involve dynamic frames, special pure methods that abstract over sets of locations. A disadvantage of this approach is that frame annotations must be provided for each method, and that they must be checked explicitly at verification time.

In this paper, we improve dynamic frames-based approaches in two ways: (1) method contracts are more concise and (2) fewer proof obligations must be discharged by the verifier. More specifically, we propose a variant of the dynamic frames approach inspired by separation logic that eliminates the need to explicitly write and check frame annotations. Instead, frame information is inferred from access assertions in pre- and postconditions. Disjointness restrictions can elegantly be expressed via separating conjunction. We have proven the soundness of our approach, implemented it in a verifier prototype and demonstrated its expressiveness by verifying several challenging examples from related work. This technical report is based on and extends a paper published at the European Conference on Object Oriented Programming 2009 [45].

The remainder of this paper is structured as follows. Section 2 describes how the implicit dynamic frames approach solves the frame problem in the presence of data abstraction. We demonstrate the approach’s expressive power using various example programs in Section 3. In Section 4, we explain how one can automatically verify whether a program satisfies its implicit dynamic frames specification. We prove the soundness of the verification technique in Section 5. In Section 6, we shortly explain how the implicit dynamic frames approach deals with inheritance and subclassing. Finally, we discuss experience with the verifier prototype, compare with related work and conclude in Sections 7, 8 and 9.

2 Implicit Dynamic Frames

This section describes how the implicit dynamic frames approach solves the frame problem (Section 2.1) and how it deals with data abstraction via pure methods (Section 2.2).

2.1 Framing

To reason modularly about a method invocation, one should not rely on the callee’s implementation, but only on its specification. For example, consider the code in Figure 1(b). To prove that the assertion at the end of the code snippet holds in every execution, one should only take into account *Cell*’s method contracts. However, the given contracts are too weak to prove the assertion. Indeed, *setX*’s implementation is allowed to change the state arbitrarily, as long as it ensures that **this.x** equals *v* on exit. In particular, the contract does not prevent *c2.setX(10)*; from modifying *c1.x*.

```

class Cell {
  int x;

  Cell()
    ensures this.x = 0;
  { this.x := 0; }

  void setX(int v)
    ensures this.x = v;
  { this.x := v; }
}
(a)

Cell c1 := new Cell();
c1.setX(5); //A

Cell c2 := new Cell();
c2.setX(10);

assert c1.x = 5;
(b)

```

Figure 1: A class *Cell* and some client code.

To prove the assertion at the end of Figure 1(b), we must strengthen *Cell*’s method contracts. More specifically, the contracts should additionally specify

an upper bound on the set of memory locations modifiable by the corresponding method. This problem is called the *frame problem*.

Various solutions to the frame problem have been proposed in the literature (see Section 8 for a detailed comparison). The solution proposed in this paper is as follows. A method may only access a memory location $o.f$ if it has permission to do so. More specifically, writing to or reading from a memory location $o.f$ requires $o.f$ to be *accessible*. Accessibility of $o.f$ is denoted $\mathbf{acc}(o.f)$. Method implementations are not allowed to mention $\mathbf{acc}(o.f)$. In particular, they are not permitted to branch over accessibility of a memory location. As a consequence, a location $o.f$ that was allocated before execution of a method m is only known to be accessible during execution of m if m 's precondition requires accessibility of $o.f$. In other words, a method's precondition provides an upper bound on the set of memory locations modifiable by the corresponding method: a method can only modify an existing location $o.f$ if that location is required to be accessible by its precondition. As an example, consider the revised version of the class *Cell* of Figure 2. *setX* can only modify $\mathbf{this.x}$, since its precondition only requires accessibility of $\mathbf{this.x}$. Similarly, *Cell*'s constructor does not require access to any location, and can therefore only assign to fields of the new object.

```

class Cell {
  int x;

  Cell()
    ensures acc(this.x) ∧ this.x = 0;
  { this.x := 0; }

  void setX(int v)
    requires acc(this.x);
    ensures acc(this.x) ∧ this.x = v;
  { this.x := v; }
}

```

Figure 2: A revised version of the class *Cell* from Figure 1(a).

The accessibility of a memory location can change over time. For example, when a new object is created, the fields of the new object become accessible. How does a method invocation affect the set of accessible memory locations? Since Java does not provide a mechanism for explicit deallocation and assertions can only mention allocated locations, it would be safe to assume that the set of accessible locations only grows across a method invocation. However, this assumption would rule out interesting specification patterns, where a method “captures” accessibility of a location. Furthermore, this assumption would break in the presence of concurrency, where accessibility of memory locations is passed on to other threads (cfr. [16, 17]). Therefore, we use the following rule instead: a memory location $o.f$ that is known to be accessible before a method invocation

is still accessible after the invocation, if $o.f$ was not required to be accessible by the callee’s precondition. An existing location $o.f$ that is required to be accessible by a method’s precondition is only accessible after invocation of that method, if the postcondition ensures $o.f$ is still accessible. Note that if the postcondition does not return permission to $o.f$, then that permission is lost forever.

Given the new method contracts for *Cell* of Figure 2 together with the rules for framing outlined above, we can now prove the assertion at the end of Figure 1(b). Informally, the reasoning is as follows. At program location *A*, the postcondition of $c_1.setX(5)$; holds: $c_1.x$ is accessible and its value is 5. Since c_2 ’s constructor does not require access to any location, it can modify neither the accessibility nor the value of any existing location. In particular, $c_1.x$ is still accessible and still holds 5. Similarly, the call $c_2.setX(10)$ only requires $c_2.x$ to be accessible, and hence $c_1.x$ is not affected. We may conclude that the assertion, $c_1.x = 5$, holds in any execution.

2.2 Data Abstraction

Data abstraction is crucial in the construction of modular programs, since it ensures that internal changes in one module do not propagate to other modules. However, the class *Cell* of Figure 2 and its specifications were not written with data abstraction in mind. More specifically, (1) client code must directly access the field x to query a *Cell* object’s internal state and (2) *Cell*’s method contracts are not implementation-independent as they mention the internal field x . Any change to *Cell*’s implementation, such as renaming x to y , would break or at least oblige us to reconsider the correctness of client code.

Developers typically solve issue (1) by adding “getters” to their classes. For example, the class *Cell* of Figure 3(a) defines a method *getX* to query a *Cell*’s internal state. The method is marked **pure** to indicate it does not have side-effects. As shown in Figure 3(b), the assertion of Figure 1(b) can now be rephrased in terms of *getX*.

To complete the decoupling between *Cell*’s implementation and client code, we should also solve issue (2) and make *Cell*’s method contracts implementation-independent. In this paper, we solve the latter issue by allowing getters to be used inside specifications. That is, we allow the effect of one method to be specified in terms of other methods. For example, the behavior of *setX* in Figure 3(a) is described in terms of its effect on *getX*.

In this paper, methods used within contracts are called *pure methods*. We distinguish two kinds of pure methods: normal pure methods (annotated with **pure**) and predicates (annotated with **predicate**). A pure method’s body consists of a single return statement, returning either a side-effect free expression (in case of a normal pure method) or an assertion (in case of a predicate). That is, a normal pure method abstracts over an expression, while a predicate abstracts over an assertion. Since assertions and expressions are side-effect free, execution of a pure method never modifies the state. Since we disallow mentioning assertions inside method bodies, predicates can only be called from contracts and

from the bodies of predicates. Furthermore, predicates are not allowed to have preconditions. In our running example, both *getX* and *valid* are pure methods. The former is a normal pure method, while the latter is a predicate. Predicates are typically used to represent invariants and to abstract over accessibility of memory locations.

To prove the assertion at the end of Figure 3(b), one must show that c_2 's constructor and $c_2.setX(10)$ do not affect the return value of $c_1.getX()$. In other words, it suffices to show that the set of locations modified by those statements is disjoint from the set of locations that $c_1.getX()$ depends on. But how can we determine which locations influence the return value of *getX*? The answer is simple.

```

class Cell {
  int x;

  Cell()
    ensures valid() ∧ getX() = 0;
  { this.x := 0; }

  void setX(int v)
    requires valid();
    ensures valid() ∧ getX() = v;
  { this.x := v; }

  predicate bool valid()
  { return acc(this.x); }

  pure int getX()
    requires valid();
  { return this.x; }

  void swap(Cell c)
    requires c ≠ null;
    requires valid() * c.valid();
    ensures valid() * c.valid();
    ensures getX() = old(c.getX());
    ensures c.getX() = old(getX());
  { int i := x; x := c.getX(); c.setX(i); }
}

```

(a)

```

Cell c1 := new Cell();
c1.setX(5); //A

Cell c2 := new Cell();
c2.setX(10);

assert c1.getX() = 5;

```

(b)

Figure 3: A revised version of class *Cell* with data abstraction.

We can deduce from the precondition of a normal pure method an upper bound on the set of locations readable by that method: a pure method p can

only read $o.f$ if p 's precondition requires $o.f$ to be accessible. In other words, the return value of a normal pure method only depends on locations required to be accessible by its precondition. A predicate does not have a precondition, so what locations does its return value depend on? We say a predicate is *self-framing*. That is, the return value of a predicate q only depends on locations that q itself requires to be accessible.

Given these properties of pure methods, we can now prove the assertion at the end of Figure 3(b). Informally, the reasoning is as follows. At program location A , the postcondition of $c_1.setX(5)$; holds: $c_1.valid()$ is true and $c_1.getX()$ returns 5. Because c_2 's constructor does not require access to any existing location, it can only modify fresh locations (i.e. c_2 's fields and fields of objects allocated within the constructor itself). Since $c_1.valid()$ only requires access to non-fresh locations, both its own return value and the return value of $c_1.getX()$ are not affected by c_2 's constructor. In addition, the set of memory locations required to be accessible by $c_1.valid()$ is disjoint from the set of locations required to be accessible by $c_2.valid()$, since the latter set only contains fresh locations (follows from the swinging pivot property, see Section 4.5). $c_2.setX()$; can only modify locations covered by $c_2.valid()$. The latter set of locations is disjoint from $c_1.valid()$, hence the return values of $c_1.valid()$ and $c_1.getX()$ are not affected by $c_2.setX(10)$; . We may conclude that the assertion, $c_1.getX() = 5$, holds in any execution.

Note that the correctness proof outlined above does not depend on internal implementation details of the class *Cell* but only on the class' specification. Therefore, changing *Cell*'s internal representation (within the boundaries set by its method contracts) does not endanger the correctness of the client code in Figure 3(b).

Implicit dynamic frames specifications support separation logic's separating conjunction (*). To illustrate the use of the separating conjunction, consider the method *swap* of Figure 3(a). *swap*'s precondition requires that the receiver and c are "separately" valid, i.e. that both **this.valid()** and $c.valid()$ hold and that the set of locations required to be accessible by **this.valid()** is disjoint from the set of locations required to be accessible by $c.valid()$. If we would have used a normal conjunction instead of a separating conjunction, we would not be able to prove $c.valid()$ holds after the assignment to x . In particular, the separating conjunction ensures that $c.valid()$ does not depend on **this.x**.

The specification of *Cell* shown in Figure 3(a) is idiomatic in the implicit dynamic frames approach. More specifically, the invariant of a class C is typically expressed in terms of a predicate declared in C . Each of C 's methods requires that the invariant holds and each mutator additionally ensures that it is preserved. Note that pure methods do not need to explicitly specify in their contract that they preserve the invariant, since they cannot modify the program state. Finally, mutators specify in their postcondition how they affect the return values of pure methods.

3 Examples

In this section, we show a number of programs annotated with implicit dynamic frame specifications and explain why they verify. All programs shown below have been verified using our verifier prototype (see Section 7).

3.1 Interval

Objects of the *Interval* class (Figure 4) represent intervals with a lower and upper bound.

```
class Interval {
  Cell low, high;

  Interval(int l, int h)
    requires  $l \leq h$ ;
    ensures valid();
    ensures  $getLow() = l \wedge getHigh() = h$ ;
    { low := new Cell(l); high := new Cell(h); }

  void shift(int d)
    requires valid();
    ensures valid();
    ensures  $getLow() = \text{old}(getLow()) + d$ ;
    ensures  $getHigh() = \text{old}(getHigh()) + d$ ;
    { low.setX(low.getX() + d);
      high.setX(high.getX() + d); }

  predicate bool valid()
  { return acc(low)  $\wedge$  low  $\neq$  null *
    acc(high)  $\wedge$  high  $\neq$  null *
    low.valid() * high.valid() *
    low.getX()  $\leq$  high.getX(); }

  pure int getLow()
    requires valid();
    { return low.getX(); }

  pure int getHigh()
    requires valid();
    { return high.getX(); }
}
```

Figure 4: A class *Interval*.

Interval is implemented in terms of two *Cell* objects referred to by *low* and

high. The invariant (the predicate *valid*) states that the locations *low* and *high* must be accessible and hold non-null values, that both *Cell* objects referred to by those fields must be valid and that the lower bound is less or equal to the upper bound. In addition, the invariant imposes some disjointness restrictions via the separating conjunction. In particular, the validity of the *Cell* object referred to by *low* must neither depend (1) on the values of *low* or *high* nor (2) on values of locations covered by *high.valid()*. A similar restriction applies to the *Cell* object referred to by *high*. If one would omit these disjointness restrictions, the mutator *shift* would not verify. More specifically, if one would omit restriction (1), *low.valid()* may demand access to *low*. Therefore, *low.setX* is allowed to set *low* to *null* and violate the invariant. If one would omit restriction (2), *low.valid()* might not be disjoint from *high.valid()*. Therefore, *low.setX* can violate *high*'s invariant which would cause the precondition of *high.setX* to fail.

The approach proposed here does not impose any built-in aliasing restrictions. In particular, it does not forbid an aggregate object from leaking references to its internal helper objects. For instance, *Interval* is allowed to leak references to its internal *Cell* objects. However, since the sets of locations covered by the invariants of the helper objects are not disjoint from the set of locations covered by the invariant of the aggregate object, modifying the helper object automatically invalidates the aggregate object.

In ownership-based approaches, an *Interval* object would own its *Cell* objects. For example, *low* and *high* would be rep-fields in Boogie methodology [3].

3.2 ConnectionPool

A *ConnectionPool* object (Figure 5) pools a number of connections. That is, client code can obtain a connection from the pool via the *getConnection* method. If the pool is empty, a new connection is created; otherwise, one of the connections is removed from the pool and returned to the client. Vice versa, *freeConnection* returns a connection to the pool. If the pool already holds the maximum number of connections, then the connection is closed; otherwise, it is added to the pool.

The interesting part of the example is *ConnectionPool*'s invariant which requires all connections in *conns* to be non-null, valid and mutually disjoint. This invariant is expressed using a special kind of assertion, called iterated star. In general, an iterated star has the form $(\forall^* x \in (min : max) \bullet \phi)$, where *min* and *max* are integer expressions. Informally, the latter assertion states that ϕ holds for all integers between *min* (inclusive) and *max* (exclusive) and that for any two different integers in that range, the locations required to be accessible by ϕ are disjoint.

Note that *getConnection* and *freeConnection* perform “ownership transfer”. More specifically, *getConnection* transfers ownership of the returned connection from the pool to the client. Vice versa, *freeConnection* captures ownership of a *Connection* object. No special methodological rules are needed to ensure safety of ownership transfer. The *ConnectionPool* example was taken from [13].

```

class ConnectionPool {
  List < Connection > conns;

  Connection getConnection()
  requires valid();
  ensures valid() * result.valid();
  {
    if(conns.size() = 0) {
      return new Connection();
    } else {
      return conns.removeFirst();
    }
  }

  void freeConnection(Connection c)
  requires valid() * c.valid();
  ensures valid();
  {
    if(conns.size() = 20) {
      c.close();
    } else {
      conns.add(c);
    }
  }

  predicate bool valid()
  { return acc(conns) ∧ conns ≠ null * conns.valid() *
    (∀*i ∈ (0 : conns.size()) •
      conns.get(i) ≠ null ∧ conns.get(i).valid()); }
}

```

Figure 5: A class *ConnectionPool*.

3.3 Iterator

Figure 6 shows an implementation of the iterator design pattern. Reasoning about the iterator pattern is challenging because of sharing: each iterator of a list l requires access to the locations covered by l 's invariant. In other words, if two iterators i_1 and i_2 of the same list are valid, then $i_1.valid() \wedge i_2.valid()$ holds but $i_1.valid() * i_2.valid()$ does not! However, disjointness is crucial for framing. How can we ensure that $i_1.valid()$ is preserved when $i_2.next()$ is called?

The key to solving the problem is the fact that the iterators share only locations of their list and that those locations are never modified by *next*. To indicate the locations covered by the list are not modified, *next* has a postcondition **untouched**(*getList().valid()*). That is, $i_1.valid()$ is preserved by $i_2.next()$,

since the locations which are shared between $i_1.valid()$ and $i_2.valid()$ are not modified by $next$.

```

class ArrayList {
  int n;
  Object[] items;

  ArrayList()
    ensures valid() ∧ size() = 0;
  { items := new Object[10]; }

  void add(Object o)
    requires valid();
    ensures valid();
    ensures size() =
      old(size()) + 1;
    ensures
      (∀*i ∈ (0 : size() - 1) •
        get(i) = old(get(i)));
    ensures get(size() - 1) = o;
  { ... }

  predicate bool valid()
  { return acc(n) * acc(items) *
    items ≠ null *
    accElms(items) *
    0 ≤ n ≤ items.length; }

  pure int size()
    requires valid();
  { return n; }

  pure Object get(int index)
    requires valid();
    requires 0 ≤ index < size();
  { return items[index]; }
}

class Iterator {
  ArrayList list; int index;

  Iterator(List l)
    requires l ≠ null ∧ l.valid();
    ensures valid() ∧ getList() = l;
    ensures untouched(
      getList().valid());
  { list := l; }

  Object next()
    requires valid() ∧ hasNext();
    ensures valid() ∧ getList() =
      old(getList());
    ensures untouched(
      getList().valid());
  { return list.items[index++]; }

  Object reset()
    requires valid();
    ensures valid() ∧ getList() =
      old(getList());
    ensures untouched(
      getList().valid());
  { index := 0; }

  predicate bool valid()
  { return acc(list) * acc(index) *
    list ≠ null ∧ list.valid() *
    0 ≤ index ≤ list.size(); }

  pure bool hasNext()
    requires valid();
  { return index < list.size(); }

  pure bool getList()
    requires valid();
  { return list; }
}

```

Figure 6: The iterator design pattern.

Note that the conjunct $\mathbf{acc}_{\text{Elms}}(\text{items})$ in *ArrayList*'s invariant is a special access assertion that gives permission to access the elements of the array. Also, it is ok for the invariant to read *items.length* without demanding access since *length* is immutable.

The client code of Figure 7 shows that multiple iterators can iterate over the same list at the same time. Note that loop modifies annotations and the swinging pivot property must not be specified explicitly by the developer, but can instead be inferred from loop invariants. More specifically, our tool provides two ways of framing loops. The first option is to verify the body of the loop assuming only the loop invariant. In particular, one must prove that the body of the loop preserves the invariant, when the loop targets and the entire heap are havocked (i.e. assigned random values) on entry. Alternatively, instead of havocking the entire heap, one can choose to havoc only the subset of the heap required to be accessible by the loop invariant. This technique is sound only if one proves that the loop's body modifies only locations required to be accessible by the loop invariant. The latter technique typically leads to more concise invariants, as locations that are accessible before entering the loop and which are only read in the loop's body must not be mentioned in the invariant. A minor disadvantage is that one has to discharge an additional proof obligation (i.e. the loop frame condition, which comes for free in the first approach). Developers can select the type of framing to be used on a case-by-case basis via an annotation.

```

ArrayList l := new ArrayList();

Iterator i1 := new Iterator(l);
Iterator i2 := new Iterator(l);

while(i1.hasNext())
  invariant i1.valid() ∧ i1.getList() = l;
  invariant i2.valid() ∧ i2.getList() = l;
  invariant untouched(l.valid());
{
  Object o1 := i1.next();
  while(i2.hasNext())
    invariant i2.valid() ∧ i2.getList() = l;
    invariant untouched(l.valid());
    {
      Object o2 := i2.next();
      ...
    }
  i2.reset();
}

```

Figure 7: Client code for *Iterator*.

3.4 Linked List

Figure 8 shows the specification and implementation of a linked list.

```

class Node { Node next; int value; }

class LinkedList {
  Node head; ghost seq<Node> nodes;

  void add(int x)
    requires valid();
    ensures valid()  $\wedge$  size() = old(size()) + 1;
    ensures get(size() - 1) = x;
    ensures ( $\forall^* i \in (0 : \text{size}() - 1) \bullet \text{get}(i) = \text{old}(\text{get}(i))$ );
  {
    Node current := head; int k := 0;
    while(current.next  $\neq$  null)
      invariant  $0 \leq k < \text{nodes.length} \wedge \text{current} = \text{nodes}[k]$ ;
      invariant current  $\neq$  null  $\wedge$  acc(current.next);
      { current := current.next; k := k + 1; }
      current.next := new Node(x);
      nodes := nodes + current.next;
    }

  predicate bool valid() {
  { return acc(head)  $\wedge$  head  $\neq$  null *
    acc(nodes)  $\wedge$   $0 < \text{nodes.length}$  *
    ( $\forall^* i \in (0 : \text{nodes.length}) \bullet \text{nodes}[i] \neq \text{null} *
      \text{acc}(\text{nodes}[i].\text{next}) * \text{acc}(\text{nodes}[i].\text{value}) *
      (i \neq \text{nodes.length} - 1 \Rightarrow \text{nodes}[i].\text{next} = \text{nodes}[i + 1])$ ) *
    nodes[0] = head  $\wedge$  nodes[nodes.length - 1].next = null; }

  pure int size()
    requires valid();
    { return nodes.length - 1; }

  pure int get(int index)
    requires valid();
    requires  $0 \leq \text{index} < \text{size}()$ ;
    { return nodes[index + 1].value; }
  }

```

Figure 8: A singly-linked list.

The specification uses the ghost field *nodes* to store the sequence of *Node* objects used by the linked list. In particular, the invariant states that the next

pointer of a node in the sequence at index i points to the node at index $i + 1$. To keep the ghost field in sync with the real program state, it must be updated explicitly. For example, the method `add` adds the newly created `Node` object at the end of the sequence.

As shown in Figure 9, it is possible to specify linked data structures using recursive pure methods instead of relying on ghost fields and quantification. For example, `Node.lseg` is a recursive predicate which states that the sequence of nodes starting at n_1 and ending at n_2 forms a list segment. Similarly, `Node.length` is a recursive pure method that returns the length of a list segment. A disadvantage of this approach is that the axioms generated for recursive pure methods (see Section 4) are prone to causing matching loops [12, Section 5] in the theorem prover. Matching loops dramatically increase verification time. Moreover, proving the correctness of a program specified with recursive pure methods often requires an inductive argument. For example, the correctness of the method `add` relies on the property that a state with two list segments, `Node.lseg(a, b)` and `Node.lseg(b, null)`, is equivalent to a state where one has a single list segment, `Node.lseg(a, null)`, whose length is the sum of the smaller list segments. Proving this property requires induction. However, SMT solvers do not automatically construct inductive proofs, and hence one must encode such a proof as a lemma method [20] in the program itself. For example, the method `appendLemma` shown below (which is used in the method `add` of Figure 9) proves the property described above.

```

lemma static void appendLemma(Node a, Node b)
  requires lseg(a, b) * lseg(b, null);
  ensures lseg(a, null);
  ensures length(a, null) =
    old(length(a, b) + length(b, null));
  {
    if(a == b) {
    } else {
      appendLemma(a.next, b);
    }
  }
}

```

More specifically, `appendLemma` encodes an inductive proof. The then branch of the if statement can be considered to be the base case, while the else branch corresponds to the inductive step. An invocation of `appendLemma` then corresponds to applying the lemma. A lemma method and an invocation of such a method should be considered as annotations. Therefore, a lemma should never modify the program state. This can be checked syntactically: a lemma method may not assign to non-ghost fields and can only call other lemma methods and pure methods. To guarantee that it is sound to erase lemma methods, we must enforce that such methods terminate. We can do so by checking that for each invocation of a lemma $e_0.m_1(e_1, \dots, e_n)$; appearing in body of a lemma m_2 one of the following holds: either m_1 appears before m_2 in the program text or the set

of locations required by to be accessible by $e_0.m_1(e_1, \dots, e_n)$'s precondition is smaller than the set of locations required to be accessible by m_2 's precondition.

```

class Node {
  Node next; int value;

  predicate static bool lseg(Node n1, Node n2)
  { return n1 = n2 ? true :
    n1 ≠ null * acc(n1.next) * acc(n1.value) * lseg(n1, n2); }

  pure static int length(Node n1, Node n2)
  requires lseg(n1, n2);
  { return n1 = n2 ? 0 : 1 + length(n1.next, n2); }
}

class LinkedList {
  Node head;

  void add(int x)
  requires valid();
  ensures valid() ∧ size() = old(size()) + 1;
  {
    Node current := head; int l := head.length();
    while(current.next ≠ null)
      invariant current ≠ null;
      invariant acc(current.next) * lseg(head, current) * lseg(current, null);
      invariant length(head, current) + length(current, null) = l;
      {
        Node oldcurrent := current; current := current.next;
        addLemma(oldcurrent);
      }
    current.next := new Node(x); appendLemma(head, current);
  }

  predicate bool valid() {
  { return acc(head) * head ≠ null ∧ lseg(head, null); }

  pure int size()
  requires valid();
  { return length(head, null) - 1; }
}

```

Figure 9: A singly-linked list specified via recursive pure methods.

4 Verification

In this section, we show how one can automatically verify whether programs, such as the ones shown above, satisfy their specification. We start by defining a small Java-like language with method contracts (4.1). We then show how one can generate verification conditions for expressions, assertions and statements (4.2 – 4.5). Finally, we define the notion of *valid program*. A program is valid if the verification conditions for each method are valid first-order formulas. In Section 5, we will prove the soundness of this verification technique: in executions of valid programs assert statements never fail and no *NullPointerExceptions* are thrown.

4.1 Language

We define the following sets.

| set | typical element | meaning |
|---------------|-----------------|--------------------------|
| \mathcal{C} | C, D | class names |
| \mathcal{F} | f | field names |
| \mathcal{M} | m | mutator names |
| \mathcal{P} | p | normal pure method names |
| \mathcal{Q} | q | predicate names |
| \mathcal{X} | x, y | variable names |

All the aforementioned sets are mutually disjoint. \mathcal{X} contains the variable **this**. We describe the details of our verification approach and prove its soundness with respect to the Java-like language of Figure 10. Overlining indicates repetition.

$$\begin{array}{ll}
 \textit{program} & ::= \overline{\textit{class } \bar{s}} \\
 \textit{class} & ::= \mathbf{class } C \{ \overline{\textit{field } \textit{method}} \} \\
 \textit{field} & ::= C f; \\
 \textit{method} & ::= \textit{mutator} \mid \overline{\textit{pure}} \mid \overline{\textit{predicate}} \\
 \textit{mutator} & ::= \mathbf{void } m(\overline{C x}) \mathbf{requires } \phi; \mathbf{ensures } \phi; \{ \bar{s} \} \\
 \textit{pure} & ::= \mathbf{pure } C p(\overline{C x}) \mathbf{requires } \phi; \{ \mathbf{return } e; \} \\
 \textit{predicate} & ::= \mathbf{predicate } \mathbf{bool } q(\overline{C x}) \{ \mathbf{return } \phi; \} \\
 s & ::= C x; \mid x := e; \mid e.f := e; \mid e.m(\bar{e}); \mid x := \mathbf{new } C; \mid \\
 & \quad \mathbf{assert } e = e; \\
 \phi & ::= \mathbf{true} \mid \mathbf{acc}(e.f) \mid \phi \wedge \phi \mid \phi * \phi \mid e = e \mid e = e ? \phi : \phi \mid e.q(\bar{e}) \\
 e & ::= \mathbf{null} \mid x \mid e.f \mid e.p(\bar{e})
 \end{array}$$

Figure 10: Syntax of a Java-like language with contracts.

A program consists of a number of classes and a main routine \bar{s} . Each class defines a number of fields and methods. We distinguish three kinds of methods: mutators, normal pure methods and predicates. Mutators and normal pure methods have a corresponding method contract. The contract of a normal pure method consists of a single assertion, the precondition. A mutator’s

contract additionally declares a postcondition. The body of a mutator consists of a sequence of statements. A statement is either a variable declaration, a variable update, a field update, a mutator invocation, an object construction or an assert statement. A predicate returns an assertion, while a normal pure method returns an expression. An assertion is either **true**, an access assertion, a conjunction, a separating conjunction, an equality, a conditional assertion or an invocation of a predicate. A separating conjunction holds only if both conjuncts hold and the left and right-hand side demand access to disjoint parts of the heap. An expression is either the constant *null*, a variable, a field read or an invocation of a pure method.

In this paper, we syntactically distinguish between normal expressions and assertions. We do so because our approach relies on the property that methods cannot branch over accessibility of a memory location. The distinction between expressions and assertions allows us to enforce this restriction in a very simple, syntactic manner.

Our specification language includes the separating conjunction for specifying disjointness restrictions. However, an alternative design choice would have been to only support normal conjunction and to allow expression disjointness in another way. For example, we could have added another kind of assertion, $\phi_1 \diamond \phi_2$, that expresses that ϕ_1 and ϕ_2 depend on disjoint parts of the heap. However, we did not adopt this choice as it leads to larger, less elegant specifications.

In this paper, we consider only well-formed programs.

Definition 1. *A program π is well-formed (denoted $\text{wf}(\pi)$) if all of the following hold:*

- *Method parameters are not assigned to.*
- *The program is well-typed. We do not formalize a type system here, but refer the reader to [14, 8, 18].*
- *The body and precondition of a pure method only call pure methods defined earlier in the program text.*

Well-formedness includes well-typedness. Moreover, we impose the restriction that a pure method can only call pure methods defined earlier in the program text. Our soundness proof (see Section 5) relies on this property to guarantee the consistency of the axioms we generate for pure methods.

The language described above does not support constructors. However, we can encode constructors using the following syntactic sugar. A constructor

$$C(C_1 x_1, \dots, C_k x_k) \text{ \textbf{requires} } \phi_1; \text{ \textbf{ensures} } \phi_2; \{ \bar{s} \}$$

is a shorthand for the mutator method

$$\begin{aligned} &\text{void } \textit{init}_C(C_1 x_1, \dots, C_k x_k) \\ &\quad \text{\textbf{requires} } \text{acc}(f_1) * \dots * \text{acc}(f_n) * \phi_1; \text{ \textbf{ensures} } \phi_2; \\ &\quad \{ \bar{s} \} \end{aligned}$$

where f_1, \dots, f_n are the fields of C . Accordingly, a constructor invocation $x := \text{new } C(e_1, \dots, e_k)$; abbreviates $x := \text{new } C; x.\textit{init}_C(e_1, \dots, e_k)$;

4.2 Logic

We target a multi-sorted, first-order logic with equality. That is, a formula ψ is either *true*, *false*, an equality between terms, a conjunction, a disjunction, a negation or a quantification. A term τ is a variable or a function application.

$$\begin{aligned}\psi &::= \text{true} \mid \text{false} \mid \tau = \tau \mid \psi \wedge \psi \mid \psi \vee \psi \mid \neg\psi \mid \forall \bar{y} \bullet \psi \\ \tau &::= y \mid g(\bar{\tau})\end{aligned}$$

Throughout this paper, we will use the following shorthands. First of all, $\psi_1 \Rightarrow \psi_2$ is a shorthand for $\neg\psi_1 \vee \psi_2$, $\psi_1 \Leftrightarrow \psi_2$ is a shorthand for $\psi_1 \Rightarrow \psi_2 \wedge \psi_2 \Rightarrow \psi_1$, and $\tau_1 \neq \tau_2$ is a shorthand for $\neg(\tau_1 = \tau_2)$. Secondly, an application of a function g with no parameters will be written g instead of $g()$. In this paper, functions with arity 0 are called constants. The formula $\text{ite}(\tau_1 = \tau_2, \psi_1, \psi_2)$ is a shorthand for $(\tau_1 = \tau_2 \Rightarrow \psi_1) \wedge (\tau_1 \neq \tau_2 \Rightarrow \psi_2)$. Finally, a term τ used in a formula position is a shorthand for $\tau = \text{ttrue}$ (*ttrue* is a constant).

4.3 Signature

Each term in the verification logic has a corresponding sort. The sorts are the following: *ref*, the sort of object references, *bool*, the sort of booleans, *set*, the sort of sets of memory locations, *fname*, the sort of field names, *cname*, the sort of class names and finally *heap*, the sort of heaps.

The signature of the logic consists of *built-in functions* and a number of *program-specific functions*. The built-in functions are the following:

| function | sort |
|------------------|---|
| <i>null</i> | <i>ref</i> |
| <i>emptyset</i> | <i>set</i> |
| <i>singleton</i> | $\text{ref} \times \text{fname} \rightarrow \text{set}$ |
| <i>union</i> | $\text{set} \times \text{set} \rightarrow \text{set}$ |
| <i>contains</i> | $\text{ref} \times \text{fname} \times \text{set} \rightarrow \text{bool}$ |
| <i>setminus</i> | $\text{set} \times \text{set} \rightarrow \text{set}$ |
| <i>select</i> | $\text{heap} \times \text{ref} \times \text{fname} \rightarrow \text{val}$ |
| <i>store</i> | $\text{heap} \times \text{ref} \times \text{fname} \times \text{val} \rightarrow \text{heap}$ |
| <i>allocated</i> | $\text{heap} \times \text{ref} \rightarrow \text{bool}$ |
| <i>allocate</i> | $\text{ref} \times \text{heap} \times \text{cname} \rightarrow \text{heap}$ |
| <i>ok</i> | $\text{heap} \times \text{set} \rightarrow \text{bool}$ |
| <i>succ</i> | $\text{heap} \times \text{set} \times \text{heap} \times \text{set} \rightarrow \text{bool}$ |
| <i>ttrue</i> | <i>bool</i> |

null represents the null reference. The functions *emptyset*, *singleton*, *union*, *contains* and *setminus* represent operations on sets of locations. To avoid cluttering the verification conditions, we employ the standard mathematical notation instead of explicitly writing down the operations as function invocations. For example, we write $(o, f) \in s$ instead of *contains*(*o*, *f*, *s*). The heap is modeled in the verification logic as a mapping from locations to values.

$select(h, o, f)$ returns the value of (o, f) in heap h and $store(h, o, f, v)$ returns a new heap that is equal to h at each location except for (o, f) . More specifically, the value of (o, f) equals v in the heap $store(h, o, f, v)$. We abbreviate $select(h, o, f)$ as $h(o, f)$ and $store(h, o, f, v)$ as $h[(o, f) \mapsto v]$. $allocated(h, o)$ returns whether o is allocated in heap h . We write $allocated_h(x_1, \dots, x_n)$ as a shorthand for $(x_1 = null \vee allocated(h, x_1)) \wedge \dots \wedge (x_n = null \vee allocated(h, x_n))$. $allocate(o, h, C)$ returns a new heap where o is allocated. $ok(h, a)$ denotes that the state with heap h and access set a is potentially reachable by the program. An important property of reachable heaps, is that fields of allocated objects only point to other allocated objects. Finally, $succ(h_1, a_1, h_2, a_2)$ denotes that state (h_2, a_2) is a successor of state (h_1, a_1) . h_2 is a successor of h_1 if each object allocated in h_1 is also allocated in h_2 . Moreover, a successor of a well-formed state is well-formed as well.

The program-specific functions are the following. For each class C in the program text, the logic includes a constant C with sort $cname$. Similarly, for each field f in the program text, the logic contains a constant f with sort $fname$. A standard technique in verification is to encode pure methods as functions in the verification logic. That is, for each normal pure method p with parameters $C_1 x_1, \dots, C_n x_n$ declared in a class C , the logic includes a function $C.p$ with sort $heap \times set \times ref \times ref_1 \times \dots \times ref_n \rightarrow ref$. Similarly, for each predicate $C.q$ with parameters $C_1 x_1, \dots, C_n x_n$, the logic includes a function $C.q$ with sort $heap \times set \times ref \times ref_1 \times \dots \times ref_n \rightarrow bool$ and a function $C.q_{FP}$ with sort $heap \times set \times ref \times ref_1 \times \dots \times ref_n \rightarrow set$. The latter function, $C.q_{FP}$, is called q 's footprint function. During verification, we not only track the value of the heap, but also the set of accessible locations. The second parameter of each function represents this set of locations.

4.4 Theory

We assume that the theory $\Sigma_{prelude}$ (incompletely) axiomatizes the built-in functions. $\Sigma_{prelude}$ may for instance contain a subtheory which axiomatizes the set functions. For example, in our verifier prototype the prelude includes an axiom that encodes that the empty set contains no locations:

$$\forall o, f \bullet (o, f) \notin emptyset$$

The behavior of a pure method is encoded via several axioms. Each normal pure method p has a corresponding axiomatization Σ_p , consisting of an implementation and a frame axiom. More specifically, the axioms corresponding to the normal pure method

$$\mathbf{pure} \ C \ p(C_1 \ x_1, \dots, C_k \ x_k) \ \mathbf{requires} \ \phi_1; \ \{ \ \mathbf{return} \ e; \ }$$

are the following:

- **Implementation axiom.** The implementation axiom relates the function symbol $C.p$ to the pure method's implementation: applying the func-

tion equals evaluating the method body, provided the precondition holds.

$$\begin{aligned} & \forall h, a, \text{this}, x_1, \dots, x_n \bullet \\ & ok(h, a) \wedge \text{this} \neq \text{null} \wedge \text{allocated}_h(\text{this}, x_1, \dots, x_n) \wedge \text{Tr}(\phi_1) \\ & \quad \Downarrow \\ & C.p(h, a, \text{this}, x_1, \dots, x_n) = \text{Tr}(e) \end{aligned}$$

The variable h denotes a heap, while a represents a set of accessible locations. Tr denotes the translation of an expression to a first-order term (see Section 4.5).

- **Frame axiom.** The frame axiom encodes the property that a pure method only depends on locations in the required access set of its precondition. That is, the return value of p is the same in two states, if locations in the required access set of the precondition have the same value in both heaps.

$$\begin{aligned} & \forall h_1, a_1, h_2, a_2, \text{this}, x_1, \dots, x_n \bullet \\ & ok(h_1, a_1) \wedge ok(h_2, a_2) \wedge \text{this} \neq \text{null} \wedge \text{allocated}_h(\text{this}, x_1, \dots, x_n) \wedge \\ & \quad \text{Tr}(\phi_1)[h_1/h, a_1/a] \wedge \text{Tr}(\phi_1)[h_2/h, a_2/a] \wedge \\ & (\forall o, f \bullet (o, f) \in \mathbb{R}(\phi_1)[h_1/h, a_1/a] \Rightarrow (o, f) \in a_2 \wedge h_1(o, f) = h_2(o, f)) \\ & \quad \Downarrow \\ & C.p(h_1, a_1, \text{this}, x_1, \dots, x_n) = C.p(h_2, a_2, \text{this}, x_1, \dots, x_n) \end{aligned}$$

Each predicate q has a corresponding axiomatization Σ_q , consisting of an implementation axiom, frame axiom, footprint implementation axiom, footprint frame axiom and a footprint allocated axiom. More specifically, the axioms corresponding to the predicate

predicate bool $q(C_1 x_1, \dots, C_n x_n) \{ \text{return } \phi; \}$

are the following:

- **Implementation axiom.** The implementation axiom relates q 's function symbol to its implementation.

$$\begin{aligned} & \forall h, a, \text{this}, x_1, \dots, x_n \bullet \\ & ok(h, a) \wedge \text{this} \neq \text{null} \wedge \text{allocated}_h(\text{this}, x_1, \dots, x_n) \\ & \quad \Downarrow \\ & C.q(h, a, \text{this}, x_1, \dots, x_n) = \text{Tr}(\phi) \end{aligned}$$

- **Frame axiom.** The frame axiom encodes the property that a predicate is self-framing.

$$\begin{aligned} & \forall h_1, a_1, h_2, a_2, \text{this}, x_1, \dots, x_n \bullet \\ & ok(h_1, a_1) \wedge ok(h_2, a_2) \wedge \text{this} \neq \text{null} \wedge \text{allocated}_h(\text{this}, x_1, \dots, x_n) \wedge \\ & \quad C.q(h_1, a_1, \text{this}, x_1, \dots, x_n) \wedge \\ & (\forall o, f \bullet (o, f) \in C.q_{\text{FP}}(h_1, a_1, \text{this}, x_1, \dots, x_n) \Rightarrow (o, f) \in a_2 \wedge h_1(o, f) = h_2(o, f)) \\ & \quad \Downarrow \\ & C.q(h_2, a_2, \text{this}, x_1, \dots, x_n) \end{aligned}$$

- **Footprint implementation axiom.** The footprint implementation axiom relates the function symbol $C.q_{\text{FP}}$ to the required access set of the body of the predicate.

$$\begin{aligned} & \forall h, a, \text{this}, x_1, \dots, x_n \bullet \\ & ok(h, a) \wedge \text{this} \neq \text{null} \wedge \text{allocated}_h(\text{this}, x_1, \dots, x_n) \wedge C.q(h, a, \text{this}, x_1, \dots, x_n) \\ & \quad \Downarrow \\ & C.q_{\text{FP}}(h, a, \text{this}, x_1, \dots, x_n) = R(\phi) \end{aligned}$$

- **Footprint frame axiom.** The footprint frame axiom encodes the property that a footprint function frames itself, provided the corresponding predicate holds.

$$\begin{aligned} & \forall h_1, a_1, h_2, a_2, \text{this}, x_1, \dots, x_n \bullet \\ & ok(h_1, a_1) \wedge ok(h_2, a_2) \wedge \text{this} \neq \text{null} \wedge \text{allocated}_h(\text{this}, x_1, \dots, x_n) \wedge \\ & \quad C.q(h_1, a_1, \text{this}, x_1, \dots, x_n) \wedge C.q(h_2, a_2, \text{this}, x_1, \dots, x_n) \wedge \\ & (\forall o, f \bullet (o, f) \in C.q_{\text{FP}}(h_1, a_1, \text{this}, x_1, \dots, x_n) \Rightarrow (o, f) \in a_2 \wedge h_1(o, f) = h_2(o, f)) \\ & \quad \Downarrow \\ & C.q_{\text{FP}}(h_1, a_1, \text{this}, x_1, \dots, x_n) = C.q_{\text{FP}}(h_2, a_2, \text{this}, x_1, \dots, x_n) \end{aligned}$$

- **Footprint accessible axiom.** The footprint accessible axiom states that a predicate footprint only contains accessible locations, provided the predicate itself holds.

$$\begin{aligned} & \forall h, a, \text{this}, x_1, \dots, x_n \bullet \\ & ok(h, a) \wedge \text{this} \neq \text{null} \wedge \text{allocated}_h(\text{this}, x_1, \dots, x_n) \wedge C.q(h, a, \text{this}, x_1, \dots, x_n) \\ & \quad \Downarrow \\ & C.q_{\text{FP}}(h, a, \text{this}, x_1, \dots, x_n) \subseteq a \end{aligned}$$

We define Σ_π as $\Sigma_{\text{prelude}} \cup \bigcup_{p \in \pi} \Sigma_p$. That is, Σ_π is the union of the axioms for the built-in functions and the axioms of each pure method in π . Moreover, we define Σ_{p^*} as the axiomatization of all pure methods defined before p in the program text. Note that Σ_{p^*} does not include Σ_p .

4.5 Verification Conditions

We check the correctness of a program by generating verification conditions. The verification conditions for each statement are shown in Figure 11. The free variables of $\text{vc}(\bar{s}, \psi)$ are h, a and the free variables of ψ and \bar{s} . In the verification conditions, h denotes the heap and a denotes the set of currently accessible locations. Tr and Df denote the translation and respectively the definedness of expressions and assertions (shown in Figures 12 and 13). $\text{mpre}(C, m)$ and $\text{mpost}(C, m)$ respectively denote the pre- and postcondition of the method $C.m$.

The first core ingredient of our approach is that a method can only access a memory location if it has permission to do so. To enforce this restriction, we track a set a of accessible locations. The verification condition for field update

$$\begin{array}{ll}
\text{vc}(C \ x; \bar{s}, \psi) & ::= \forall x \bullet \text{vc}(\bar{s}, \psi) \\
\text{vc}(x := e; \bar{s}, \psi) & ::= \text{Df}(e) \wedge \text{vc}(\bar{s}, \psi)[\text{Tr}(e)/x] \\
\text{vc}(e_1.f := e_2; \bar{s}, \psi) & ::= \text{Df}(e_1) \wedge \text{Df}(e_2) \wedge (\text{Tr}(e_1), f) \in a \wedge \\
& \quad (\text{succ}(h, a, h[(\text{Tr}(e_1), f) \mapsto \text{Tr}(e_2)], a) \Rightarrow \\
& \quad \quad \text{vc}(\bar{s}, \psi)[h[(\text{Tr}(e_1), f) \mapsto \text{Tr}(e_2)]/h]) \\
\text{vc}(e_0.m(e_1, \dots, e_n); \bar{s}, \psi) & ::= \text{Df}(e_0) \wedge \dots \wedge \text{Df}(e_n) \wedge \text{Tr}(e_0) \neq \text{null} \wedge \\
& \quad \text{Tr}(P) \wedge \\
& \quad (\forall h', a' \bullet \\
& \quad \quad \text{succ}(h, a, h', a') \wedge \\
& \quad \quad \text{Tr}(Q)[h'/h, a'/a] \wedge \\
& \quad \quad (\forall o, f \bullet (o, f) \in a \wedge (o, f) \notin \text{R}(P) \Rightarrow \\
& \quad \quad \quad (o, f) \in a' \wedge h(o, f) = h'(o, f)) \wedge \\
& \quad \quad (\forall o, f \bullet (o, f) \in \text{R}(Q)[h'/h, a'/a] \Rightarrow \\
& \quad \quad \quad (o, f) \in \text{R}(P) \vee \neg(o, f) \in a) \\
& \quad \quad \Rightarrow \\
& \quad \quad \text{vc}(\bar{s}, \psi)[h'/h, a'/a]) \\
& \quad \text{where } C \text{ is the type of } e_0, x_1, \dots, x_n \\
& \quad \text{are } C.m \text{'s parameters,} \\
& \quad P \text{ is } \text{mpre}(C, m)[e_0/\mathbf{this}, e_1/x_1, \dots, e_n/x_n], \\
& \quad Q \text{ is } \text{mpost}(C, m)[e_0/\mathbf{this}, e_1/x_1, \dots, e_n/x_n] \\
\text{vc}(x := \mathbf{new} \ C; \bar{s}, \psi) & ::= \forall y \bullet y \neq \text{null} \wedge \neg \text{allocated}(y, h) \Rightarrow \\
& \quad \text{vc}(\bar{s}, \psi)[y/x, (a \cup \{(y, f_1), \dots, (y, f_n)\})/a, \\
& \quad \quad \text{allocate}(y, h)/h] \\
& \quad \text{where } f_1, \dots, f_n \text{ are the fields of } C \\
\text{vc}(\mathbf{assert} \ e_1 = e_2; \bar{s}, \psi) & ::= \text{Df}(e_1 = e_2) \wedge \text{Tr}(e_1 = e_2) \wedge \text{vc}(\bar{s}, \psi)
\end{array}$$

Figure 11: Verification conditions (vc) of statements with respect to postcondition ψ .

checks that the assignee is in the access set a . Similarly, a field read $o.f$ is only well-defined if $o.f$ is an element of a .

The second core ingredient of our approach is that we deduce frame information from a callee's precondition. More specifically, a callee can only read or modify an existing location $o.f$ if its precondition demands access to $o.f$. A naive, literal encoding of this property does not lead to good performance with automatic theorem provers. In particular, the combination of the literal encoding and our approach for data abstraction of Section 2.2 yields verification conditions that are too hard for those provers. Therefore, we propose a slightly different encoding. More specifically, we syntactically infer from the callee's precondition a *required access set*, i.e. a term denoting the set of memory locations required to be accessible by the precondition. The definition of required access set (R) of an assertion is shown in Figure 14. The subformula

$$\forall o, f \bullet (o, f) \in a \wedge (o, f) \notin \text{R}(P) \Rightarrow (o, f) \in a' \wedge h(o, f) = h'(o, f)$$

| | |
|--|---|
| $\text{Tr}(\text{null})$ | $:= \text{null}$ |
| $\text{Tr}(x)$ | $:= x$ |
| $\text{Tr}(e.f)$ | $:= h(\text{Tr}(e), f)$ |
| $\text{Tr}(e0.p(e_1, \dots, e_n))$ | $:= C.p(h, a, \text{Tr}(e_0), \dots, \text{Tr}(e_n))$ |
| $\text{Tr}(\text{true})$ | $:= \text{true}$ |
| $\text{Tr}(\text{acc}(e.f))$ | $:= (\text{Tr}(e), f) \in a$ |
| $\text{Tr}(\phi_1 \wedge \phi_2)$ | $:= \text{Tr}(\phi_1) \wedge \text{Tr}(\phi_2)$ |
| $\text{Tr}(\phi_1 * \phi_2)$ | $:= \text{Tr}(\phi_1) \wedge \text{Tr}(\phi_2) \wedge (\text{R}(\phi_1) \cap \text{R}(\phi_2) = \emptyset)$ |
| $\text{Tr}(e_1 = e_2)$ | $:= \text{Tr}(e_1) = \text{Tr}(e_2)$ |
| $\text{Tr}(e_1 = e_2 ? \phi_1 : \phi_2)$ | $:= \text{ite}(\text{Tr}(e_1 = e_2), \text{Tr}(\phi_1), \text{Tr}(\phi_2))$ |
| $\text{Tr}(e0.q(e_1, \dots, e_n))$ | $:= C.q(h, a, \text{Tr}(e_0), \dots, \text{Tr}(e_n))$ |

Figure 12: Translation (Tr) of expressions and assertions.

| | |
|--|--|
| $\text{Df}(\text{null})$ | $:= \text{true}$ |
| $\text{Df}(x)$ | $:= \text{true}$ |
| $\text{Df}(e.f)$ | $:= \text{Df}(e) \wedge (\text{Tr}(e), f) \in a$ |
| $\text{Df}(e0.p(e_1, \dots, e_n))$ | $:= \text{Df}(e_0) \wedge \dots \wedge \text{Df}(e_n) \wedge \text{Tr}(e_0) \neq \text{null} \wedge$ $\text{Tr}(\text{mpre}(C, p)[e_0/\text{this}, e_1/x_1, \dots, e_n/x_n])$ |
| $\text{Df}(\text{true})$ | $:= \text{true}$ |
| $\text{Df}(\text{acc}(e.f))$ | $:= \text{Df}(e) \wedge \text{Tr}(e) \neq \text{null}$ |
| $\text{Df}(\phi_1 \wedge \phi_2)$ | $:= \text{Df}(\phi_1) \wedge (\text{Tr}(\phi_1) \Rightarrow \text{Df}(\phi_2))$ |
| $\text{Df}(\phi_1 * \phi_2)$ | $:= \text{Df}(\phi_1 \wedge \phi_2)$ |
| $\text{Df}(e_1 = e_2)$ | $:= \text{Df}(e_1) \wedge \text{Df}(e_2)$ |
| $\text{Df}(e_1 = e_2 ? \phi_1 : \phi_2)$ | $:= \text{Df}(e_1) \wedge \text{Df}(e_2) \wedge \text{ite}(\text{Tr}(e_1 = e_2), \text{Df}(\phi_1), \text{Df}(\phi_2))$ |
| $\text{Df}(e0.q(e_1, \dots, e_n))$ | $:= \text{Df}(e_0) \wedge \dots \wedge \text{Df}(e_n) \wedge \text{Tr}(e_0) \neq \text{null}$ |

Figure 13: Definedness (Df) of expressions and assertions.

| | |
|---|---|
| $\text{R}(\text{true})$ | $:= \emptyset$ |
| $\text{R}(\text{acc}(e.f))$ | $:= \{(\text{Tr}(e), f)\}$ |
| $\text{R}(\phi_1 \wedge \phi_2)$ | $:= \text{R}(\phi_1) \cup \text{R}(\phi_2)$ |
| $\text{R}(\phi_1 * \phi_2)$ | $:= \text{R}(\phi_1) \cup \text{R}(\phi_2)$ |
| $\text{R}(e_1 = e_2)$ | $:= \emptyset$ |
| $\text{R}(e_1 = e_2 ? \phi_1 : \phi_2)$ | $:= \text{ite}(\text{Tr}(e_1 = e_2), \text{R}(\phi_1), \text{R}(\phi_2))$ |
| $\text{R}(e0.q(e_1, \dots, e_n))$ | $:= C.q_{\text{FP}}(h, a, \text{Tr}(e_0), \dots, \text{Tr}(e_n))$ |

Figure 14: Required access set (R) of assertions.

in the verification condition of method invocation encodes the property that all locations $o.f$ that are accessible to the callee and that were not in the required access set of the precondition remain accessible and retain their value. Note that this is a *free* postcondition: callers can assume the postcondition holds, but it is not necessary to explicitly prove the postcondition when verifying a mutator’s implementation (see Definition 2). In addition to the “free modifies” clause, callers may assume a second free postcondition, the swinging pivot property:

$$\forall o, f \bullet (o, f) \in R(Q)[h'/h, a'/a] \Rightarrow (o, f) \in R(P) \vee \neg(o, f) \in a$$

The swinging pivot property states that all locations required to be accessible by the postcondition are either required to be accessible by the precondition or are not accessible to the callee. This property is crucial for preserving disjointness.

4.6 Validity

We can now define the notion of valid program. A program is *valid* if all methods and the main routine are valid. A mutator is valid if the assertions in the method contract are well-defined and the body satisfies the method contract. The main routine is valid if it satisfies the contract **requires true; ensures true**; $\{\bar{s}\}$. A normal pure method is valid if the precondition is well-defined and the body is a well-defined assertion, provided the precondition holds. Finally, a predicate is valid if its body is a well-defined assertion.

Definition 2. *A mutator method*

$$\text{void } m(C_1 \ x_1, \dots, C_n \ x_n) \text{ requires } \phi_1; \text{ ensures } \phi_2; \{ \bar{s} \}$$

is valid if all of the following hold:

- The precondition is well-defined.

$$\begin{array}{c} \Sigma_\pi \vdash \forall h, a, \text{this}, x_1, \dots, x_n \bullet \\ ok(h, a) \wedge \text{this} \neq \text{null} \wedge \text{allocated}_h(\text{this}, x_1, \dots, x_n) \\ \Downarrow \\ \text{Df}(\phi_1) \end{array}$$

- The postcondition is well-defined, provided the precondition holds.

$$\begin{array}{c} \Sigma_\pi \vdash \forall h, a, h', a', \text{this}, x_1, \dots, x_n \bullet \\ ok(h, a) \wedge \text{succ}(h, a, h', a') \wedge \text{this} \neq \text{null} \wedge \text{allocated}_h(\text{this}, x_1, \dots, x_n) \wedge \text{Tr}(\phi_1) \\ \Downarrow \\ \text{Df}(\phi_2)[h'/h, a'/a] \end{array}$$

- The method body satisfies the method contract.

$$\begin{array}{c} \Sigma_\pi \vdash \forall h, a, \text{this}, x_1, \dots, x_n \bullet \\ ok(h, a) \wedge \text{this} \neq \text{null} \wedge \text{allocated}_h(\text{this}, x_1, \dots, x_n) \wedge \text{Tr}(\phi_1) \\ \Downarrow \\ \text{vc}(\bar{s}, \text{Tr}(\phi_2)) \end{array}$$

Definition 3. The main routine \bar{s} is valid if the following holds:

$$\Sigma_\pi \vdash \forall h, a \bullet ok(h, a) \Rightarrow vc(\bar{s}, true)$$

Definition 4. A predicate

$$\text{predicate bool } q(C_1 x_1, \dots, C_n x_n) \{ \text{return } \phi; \}$$

is valid if its body is a well-defined assertion:

$$\begin{array}{c} \Sigma_{prelude} \cup \Sigma_{q*} \vdash \forall h, a, this, x_1, \dots, x_n \bullet \\ ok(h, a) \wedge this \neq null \wedge \text{allocated}_h(this, x_1, \dots, x_n) \\ \Downarrow \\ \text{Df}(\phi) \end{array}$$

Definition 5. A pure method

$$\text{pure } C p(C_1 x_1, \dots, C_n x_n) \text{ requires } \phi_1; \{ \text{return } e; \}$$

is valid if its precondition is well-defined and its body is well-defined, provided the precondition holds:

$$\begin{array}{c} \Sigma_{prelude} \cup \Sigma_{p*} \vdash \forall h, a, this, x_1, \dots, x_n \bullet \\ ok(h, a) \wedge this \neq null \wedge \text{allocated}_h(this, x_1, \dots, x_n) \\ \Downarrow \\ \text{Df}(\phi_1) \wedge (\text{Tr}(\phi_1) \Rightarrow \text{Df}(e)) \end{array}$$

Definition 6. A program π is valid (denoted $\text{valid}(\pi)$) if all methods (both pure and mutator) and the main routine are valid.

5 Soundness

In this section, we prove that the verification technique described in the previous sections is sound, i.e. in executions of valid programs assert statements never fail and no *NullPointerExceptions* are thrown. We start by defining an execution semantics for programs written in the language of Figure 10. Execution of a program is in a stuck configuration (or goes wrong) if that configuration is non-final, but cannot make progress. We then introduce the notion of valid configuration. We prove that executions of valid programs do not get stuck by proving that (1) that the initial state is valid, (2) that the small-step relation \rightarrow preserves validity and (3) that valid configuration are never stuck.

We define \mathcal{O} as the set of object references, \mathbb{B} as the set of booleans and \mathcal{V} as the set of values (the union of \mathcal{O} and \mathbb{B}). Each object reference o has a corresponding type, denoted $\text{type}(o)$. That is, type is a function from object references to class names. \mathcal{O} includes the *null* reference.

5.1 Execution Semantics

We start by defining configurations.

Definition 7. *A configuration consists of:*

- a heap H , a partial function from object references to object states. An object state is a partial function from field names to object references.
- a stack of activation records S . Each activation record consists of
 - an environment Γ , a partial function from variable names to object references.
 - an access set $A \subseteq \mathcal{O} \times \mathcal{F}$, a set of memory locations. A thread can only access memory locations in A .
 - an old heap G . G is the value of the heap at the time the activation record was put onto the stack.
 - an old access set $B \subseteq \mathcal{O} \times \mathcal{F}$, a set of memory locations. B stores the access set of the callee at the time the activation record was put onto the stack.
 - a statement sequence \bar{s} .

A configuration is a pair consisting of a global heap H and a stack S (Definition 7). The heap maps object references to object states and each object state maps field names to object references. An object reference o is *allocated* in a heap H if $o \in \text{dom}(H)$. A memory location is an (object reference, field name) pair. A memory location $o.f$ is *allocated* if o is allocated and $f \in \text{dom}(H(o))$. We will sometimes implicitly uncurry the heap. That is, we write $H(o, f)$ instead of $H(o)(f)$. The stack consists of a list of activation records. The first element of the list is the top of the stack. nil denotes the empty stack and the concatenation of an activation record a and a stack s is denoted $a \cdot s$. Note that each activation record contains ghost variables (A , G and B) that are only needed for the soundness proof. Those variables can be omitted in executions of programs that successfully verify.

In the remainder of this section, we describe 4 judgments that together define execution of programs. More specifically, the judgment $H, \Gamma, A \vdash e \Downarrow v$ states that the expression e evaluates to a value v in the heap H , environment Γ and access set A . Similarly, the judgment $H, \Gamma, A \vdash \phi \Downarrow b$ expresses that the assertion ϕ evaluates to the boolean b . $H, \Gamma, A \vdash_R \phi \Downarrow s$ states that the assertion ϕ requires the locations in the set s to be accessible. Finally, $\sigma \rightarrow \sigma'$ states that the configuration σ can reach configuration σ' by executing a single statement.

5.1.1 Statements

Execution of programs is defined by the small-step relation \rightarrow (Figure 15). \rightarrow defines a *run-time checking* execution semantics, i.e. it gets stuck at run-time errors such as null dereferences, failing assert statements and pre- and

postcondition violations. The goal of the verification technique described in Section 4 is to rule out programs that might get stuck.

$$\begin{array}{c}
\frac{\Gamma' = \Gamma[x \mapsto \text{null}]}{(H, (\Gamma, A, G, B, C \ x; \bar{s}) \cdot S) \rightarrow (H, (\Gamma', A, G, B, \bar{s}) \cdot S)} \\
\\
\frac{H, \Gamma, A \vdash e \Downarrow v \quad \Gamma' = \Gamma[x \mapsto v]}{(H, (\Gamma, A, G, B, x := e; \bar{s}) \cdot S) \rightarrow (H, (\Gamma', A, G, B, \bar{s}) \cdot S)} \\
\\
\frac{H, \Gamma, A \vdash e_1 \Downarrow v_1 \quad H, \Gamma, A \vdash e_2 \Downarrow v_2 \quad (v_1, f) \in A \quad H' = H[(v_1, f) \mapsto v_2]}{(H, (\Gamma, A, G, B, e_1.f := e_2; \bar{s}) \cdot S) \rightarrow (H', (\Gamma, A, G, B, \bar{s}) \cdot S)} \\
\\
\frac{\text{type}(v_0) = C \quad \text{mbody}(C, m) = \bar{r} \quad \text{mparams}(C, m) = C_1 \ x_1, \dots, C_n \ x_n \quad \Gamma' = [\mathbf{this} \mapsto v_0, x_1 \mapsto v_1, \dots, x_n \mapsto v_n] \quad H, \Gamma', A \vdash \text{mpre}(C, m) \Downarrow \mathbf{true} \quad H, \Gamma', A \vdash_R \text{mpre}(C, m) \Downarrow A'}{(H, (\Gamma, A, G, B, e_0.m(e_1, \dots, e_n); \bar{s}) \cdot S) \rightarrow (H, (\Gamma', A', H, A, \bar{r}) \cdot (\Gamma, A \setminus A', G, B, e_0.m(e_1, \dots, e_n); \bar{s}) \cdot S)} \\
\\
\frac{\text{type}(\Gamma'(\mathbf{this})) = C \quad H, \Gamma', A' \vdash \text{mpost}(C, m) \Downarrow \mathbf{true} \quad H, \Gamma', A' \vdash_R \text{mpost}(C, m) \Downarrow A''}{(H, (\Gamma', A', G', B', \text{nil}) \cdot (\Gamma, A, G, B, e_0.m(e_1, \dots, e_n); \bar{s}) \cdot S) \rightarrow (H, (\Gamma, A \cup A'', G, B, \bar{s}) \cdot S)} \\
\\
\frac{o \notin \text{dom}(H) \quad o \neq \text{null} \quad \text{type}(o) = C \quad \text{fields}(C) = C_1 \ f_1, \dots, C_n \ f_n \quad H' = H[o \mapsto [f_1 \mapsto \text{null}, \dots, f_n \mapsto \text{null}]] \quad A' = A \cup \{(o, f_1), \dots, (o, f_n)\}}{(H, (\Gamma, A, G, B, x := \mathbf{new} \ C; \bar{s}) \cdot S) \rightarrow (H', (\Gamma[x \mapsto o], A', G, B, \bar{s}) \cdot S)} \\
\\
\frac{H, \Gamma, A \vdash e_1 \Downarrow v \quad H, \Gamma, A \vdash e_2 \Downarrow v}{(H, (\Gamma, A, G, B, \mathbf{assert} \ e_1 = e_2; \bar{s}) \cdot S) \rightarrow (H, (\Gamma, A, G, B, \bar{s}) \cdot S)}
\end{array}$$

Figure 15: Execution of statements.

$H, \Gamma, A \vdash e \Downarrow v$ denotes that the expression e evaluates to value v . Similarly, $H, \Gamma, A \vdash \phi \Downarrow b$ denotes that the assertion ϕ evaluates to the boolean b . Finally, $H, \Gamma, A \vdash_R \phi \Downarrow s$ denotes that the assertion ϕ demands access to the set of locations s . The set s is called ϕ 's required access set. The latter three judgments will be defined in the next section.

A variable declaration $C \ x;$ initializes x to null . A variable declaration cannot be stuck. A variable update $x := e;$ changes the value of x in the environment to v , where v is the value of e . A variable update is stuck if e is

stuck. A field update $e_1.f := e_2$; sets the value of $v_1.f$ to v_2 , where v_1 and v_2 are the values of e_1 , respectively e_2 . A field update is stuck if either e_1 or e_2 get stuck or if the thread does not have permission to access $v_1.f$ (i.e. $(v_1, f) \notin A$). A mutator invocation $e_0.m(e_1, \dots, e_n)$; puts a new activation record onto the call stack. The environment of the new activation record is initialized to the values of the parameters. The new activation record's access set is initialized to the set of locations for which the precondition demands accessibility. Moreover, that set of locations is subtracted from the required access set of the caller. G and B store a copy of the heap and access set. Note that G and B are never modified. The statements of the new activation record are the statements in the body of the callee. A mutator invocation is stuck if either one of the parameters gets stuck, evaluation of the precondition gets stuck, the precondition does not hold, or if computing the required access set gets stuck. A method return pops the top activation record from the stack. A return is stuck if the postcondition gets stuck or if the postcondition does not hold. An object creation $x := \mathbf{new} C$; selects an arbitrary, non-allocated object reference o and adds a new location $o.f$ to the heap for each field f declared in C . In addition, the access set is extended such that the activation record gains access to all of o 's fields. Finally, the value of x in Γ is changed to o . Object creation cannot be stuck. Finally, an assert statement **assert** $e_1 = e_2$; has no effect, but it is stuck if evaluation of e_1 or e_2 gets stuck or if their values are different.

Execution starts in the *initial state* $(\emptyset, (\emptyset, \emptyset, \emptyset, \emptyset, \bar{s}) \cdot \text{nil})$. That is, execution starts in a state with an empty heap and a stack with a single activation record. This activation record has an empty access set, empty environment and executes the main routine.

Definition 8. A configuration σ is final (denoted $\text{final}(\sigma)$) if the stack contains a single activation record with an empty statement sequence.

Definition 9. A configuration σ is stuck (denoted $\text{stuck}(\sigma)$) if σ is not final and there exists no σ' such that $\sigma \rightarrow \sigma'$.

5.1.2 Expressions

The judgment $H, \Gamma, A \vdash e \Downarrow v$ states that e evaluates to the object reference v (Figure 16). The constant *null* evaluates to itself. A variable x evaluates to the value for x in environment. A field read $e.f$ evaluates to the value of the location $v.f$ in the heap, where v is the value of e . A field read $e.f$ gets stuck if evaluation of e gets stuck or if the location being read is not accessible (i.e. $(v, f) \notin A$). A normal pure method invocation $e_0.p(e_1, \dots, e_n)$ evaluates to the value of the body of the callee. Such an invocation gets stuck if either one of the parameters gets stuck, the precondition gets stuck, the precondition does not hold, the receiver is *null*, computing the required access set gets stuck or if evaluation of the body gets stuck.

Definition 10. Evaluation of an expression e is stuck in a heap H , environment Γ and access set A if there does not exist a value v such that $H, \Gamma, A \vdash e \Downarrow v$.

$$\begin{array}{c}
H, A, \Gamma \vdash \text{null} \Downarrow \text{null} \quad H, \Gamma, A \vdash x \Downarrow \Gamma(x) \quad \frac{H, \Gamma, A \vdash e \Downarrow v \quad (v, f) \in A}{H, \Gamma, A \vdash e.f \Downarrow H(v, f)} \\
\\
\frac{
\begin{array}{c}
v_0 \neq \text{null} \quad H, \Gamma, A \vdash e_0 \Downarrow v_0 \dots H, \Gamma, A \vdash e_n \Downarrow v_n \\
\text{type}(v_0) = C \quad \text{mparams}(C, p) = C_1 \ x_1, \dots, C_n \ x_n \\
H, [\mathbf{this} \mapsto v_0, \dots, x_n \mapsto v_n], A \vdash \text{mpre}(C, p) \Downarrow \mathbf{true} \\
H, [\mathbf{this} \mapsto v_0, \dots, x_n \mapsto v_n], A \vdash_R \text{mpre}(C, p) \Downarrow A' \\
H, [\mathbf{this} \mapsto v_0, \dots, x_n \mapsto v_n], A' \vdash \text{mbody}(C, p) \Downarrow v
\end{array}
}{H, \Gamma, A \vdash e_0.p(e_1, \dots, e_n) \Downarrow v}
\end{array}$$

Figure 16: Evaluation of expressions.

5.1.3 Assertions

The judgment $H, \Gamma, A \vdash \phi \Downarrow b$ states that the assertion ϕ evaluates to the boolean b (Figure 17). A boolean constant b evaluates to itself. An access assertion $\mathbf{acc}(e.f)$ evaluates to **true** only if the location $v.f$ is an element of A , where v is the value of e . An access assertion gets stuck if evaluation of e gets stuck. A conjunction $\phi_1 \wedge \phi_2$ evaluates to **true** only if both ϕ_1 and ϕ_2 evaluate to **true**. A conjunction gets stuck if either ϕ_1 or ϕ_2 gets stuck. A separating conjunction $\phi_1 * \phi_2$ holds only if $\phi_1 \wedge \phi_2$ evaluates to **true** and the set of locations demanded to be accessible by ϕ_1 is disjoint from the set demanded to be accessible by ϕ_2 . Note that both normal conjunction and separating conjunction shortcut, i.e. if the left-hand side evaluates to **false**, then the right-hand side is not evaluated. An equality $e_1 = e_2$ evaluates to **true** if v_1 equals v_2 and gets stuck if e_1 or e_2 get stuck. A conditional assertion $e_1 = e_2 ? \phi_1 : \phi_2$ evaluates to the value of ϕ_1 if v_1 equals v_2 ; otherwise it evaluates to the value of ϕ_2 . A conditional assertion gets stuck if one of its subexpressions or subassertions gets stuck. Note that ϕ_2 is not evaluated (and hence cannot get stuck) if the condition holds and vice versa. Finally, a predicate instance evaluates to the value of its body. A predicate gets stuck if its body gets stuck.

Definition 11. *Evaluation of an assertion ϕ is stuck in a heap H , environment Γ and access set A if there does not exist a boolean b such that $H, \Gamma, A \vdash \phi \Downarrow b$.*

The judgment $H, \Gamma, A \vdash_R \phi \Downarrow s$ states that the assertion ϕ demands (or requires) access to the set of locations s (Figure 18). That is, if one of the locations in s is not accessible, then ϕ will evaluate to **false**. A boolean constant b requires access to the empty set. An access assertion $\mathbf{acc}(e.f)$ demands access to the singleton set $\{v.f\}$, where v is the value of e . The required access set of a conjunction $\phi_1 \wedge \phi_2$ is the union of the required access sets of ϕ_1 and ϕ_2 . Similarly, the required access set of the separating conjunction $\phi_1 * \phi_2$ is the union of the required access sets of ϕ_1 and ϕ_2 . The equality $e_1 = e_2$ demands access to no location (although the expressions might get stuck if A is too small).

$$\begin{array}{c}
H, \Gamma, A \vdash b \Downarrow b \\
\frac{H, \Gamma, A \vdash e \Downarrow v}{H, \Gamma, A \vdash \mathbf{acc}(e.f) \Downarrow ((v, f) \in A)} \\
\frac{H, \Gamma, A \vdash \phi_1 \Downarrow \mathbf{false}}{H, \Gamma, A \vdash \phi_1 \wedge \phi_2 \Downarrow \mathbf{false}} \quad \frac{H, \Gamma, A \vdash \phi_1 \Downarrow \mathbf{true} \quad H, \Gamma, A \vdash \phi_2 \Downarrow b_2}{H, \Gamma, A \vdash \phi_1 \wedge \phi_2 \Downarrow b_2} \\
\frac{H, \Gamma, A \vdash \phi_1 \wedge \phi_2 \Downarrow \mathbf{false}}{H, \Gamma, A \vdash \phi_1 * \phi_2 \Downarrow \mathbf{false}} \\
\frac{H, \Gamma, A \vdash \phi_1 \wedge \phi_2 \Downarrow \mathbf{true} \quad H, \Gamma, A \vdash_R \phi_1 \Downarrow s_1 \quad H, \Gamma, A \vdash_R \phi_2 \Downarrow s_2}{H, \Gamma, A \vdash \phi_1 * \phi_2 \Downarrow (s_1 \cap s_2 = \emptyset)} \\
\frac{H, \Gamma, A \vdash e_1 \Downarrow v_1 \quad H, \Gamma, A \vdash e_2 \Downarrow v_2}{H, \Gamma, A \vdash e_1 = e_2 \Downarrow v_1 = v_2} \\
\frac{H, \Gamma, A \vdash e_1 \Downarrow v \quad H, \Gamma, A \vdash e_2 \Downarrow v \quad H, \Gamma, A \vdash \phi_1 \Downarrow b}{H, \Gamma, A \vdash e_1 = e_2 ? \phi_1 : \phi_2 \Downarrow b} \\
\frac{H, \Gamma, A \vdash e_1 \Downarrow v_1 \quad H, \Gamma, A \vdash e_2 \Downarrow v_2 \quad v_1 \neq v_2 \quad H, \Gamma, A \vdash \phi_2 \Downarrow b}{H, \Gamma, A \vdash e_1 = e_2 ? \phi_1 : \phi_2 \Downarrow b} \\
\frac{v_0 \neq \mathit{null} \quad \text{type}(v_0) = C \quad \text{mparams}(C, q) = C_1 x_1, \dots, C_n x_n \quad H, [\mathbf{this} \mapsto v_0, \dots, x_n \mapsto v_n], A \vdash \mathbf{mbody}(C, q) \Downarrow \mathbf{true}}{H, \Gamma, A \vdash e_0.q(e_1, \dots, e_n) \Downarrow b}
\end{array}$$

Figure 17: Evaluation of assertions.

A conditional assertion demands access to either the required access set of ϕ_1 or ϕ_2 , depending on whether the condition holds. The required access set of a predicate is the required access set of its body. Computation of the required access set gets stuck if one of the subexpressions or subassertions gets stuck.

Definition 12. *Computing the required access set of an assertion ϕ is stuck in a heap H , environment Γ and access set A if there does not exist a set s such that $H, \Gamma, A \vdash_R \phi \Downarrow s$.*

5.1.4 Well-formed Configurations

We define the notion of well-formed configuration (Definition 14) and prove that \rightarrow preserves well-formedness of configurations (Theorem 3). We start by defining what it means for a heap H , environment Γ , and access set A to be well-formed. Informally, (H, Γ, A) is well-formed if the range of H , the range of

$$\begin{array}{c}
H, \Gamma, A \vdash_R b \Downarrow \emptyset \qquad \frac{H, \Gamma, A \vdash e \Downarrow v}{H, \Gamma, A \vdash_R \mathbf{acc}(e.f) \Downarrow \{(v, f)\}} \\
\\
\frac{H, \Gamma, A \vdash_R \phi_1 \Downarrow s_1 \quad H, \Gamma, A \vdash_R \phi_2 \Downarrow s_2}{H, \Gamma, A \vdash_R \phi_1 \wedge \phi_2 \Downarrow s_1 \cup s_2} \\
\\
\frac{H, \Gamma, A \vdash_R \phi_1 \Downarrow s_1 \quad H, \Gamma, A \vdash_R \phi_2 \Downarrow s_2}{H, \Gamma, A \vdash_R \phi_1 * \phi_2 \Downarrow s_1 \cup s_2} \qquad \frac{}{H, \Gamma, A \vdash_R e_1 = e_2 \Downarrow \emptyset} \\
\\
\frac{H, \Gamma, A \vdash e_1 \Downarrow v \quad H, \Gamma, A \vdash e_2 \Downarrow v \quad H, \Gamma, A \vdash_R \phi_1 \Downarrow s}{H, \Gamma, A \vdash_R e_1 = e_2 ? \phi_1 : \phi_2 \Downarrow s} \\
\\
\frac{H, \Gamma, A \vdash e_1 \Downarrow v_1 \quad H, \Gamma, A \vdash e_2 \Downarrow v_2 \quad v_1 \neq v_2 \quad H, \Gamma, A \vdash_R \phi_2 \Downarrow s}{H, \Gamma, A \vdash_R e_1 = e_2 ? \phi_1 : \phi_2 \Downarrow s} \\
\\
\frac{v_0 \neq \mathit{null} \quad \text{type}(v_0) = C \quad \text{mparams}(C, q) = C_1 x_1, \dots, C_n x_n \quad H, [\mathbf{this} \mapsto v_0, \dots, x_n \mapsto v_n], A \vdash_R \mathbf{mbody}(C, q) \Downarrow s}{H, \Gamma, A \vdash_R e_0.q(e_1, \dots, e_n) \Downarrow s}
\end{array}$$

Figure 18: Required access set.

Γ and the set A do not contain non-allocated, non-null object references. We will use the name *context* for a triple (H, Γ, A) .

Definition 13. A heap H is well-formed (denoted $\text{wf}(H)$) if each location in its domain holds either null or an allocated object reference. In particular, an allocated location never holds a non-allocated object.

$$\text{wf}(H) \Leftrightarrow \forall o \in \text{dom}(H), f \in \text{dom}(H(o)) \bullet o \neq \mathit{null} \Rightarrow H(o, f) \in \text{dom}(H)$$

An environment Γ is well-formed with respect to a heap H (denoted $\text{wf}(\Gamma, H)$) if Γ does map variables to non-allocated object references.

$$\text{wf}(\Gamma, H) \Leftrightarrow \forall x \in \text{dom}(\Gamma) \bullet \Gamma(x) \neq \mathit{null} \Rightarrow \Gamma(x) \in \text{dom}(H)$$

A set of memory locations A is well-formed with respect to a heap H (denoted $\text{wf}(A, H)$) if each location in A is allocated in H .

$$\text{wf}(A, H) \Leftrightarrow \forall (o, f) \in A \bullet o \in \text{dom}(H) \wedge f \in \text{dom}(H(o))$$

A triple H, Γ, A is well-formed if H is well-formed and both Γ and A are well-formed with respect to H .

Definition 14 defines what it means for a configuration to be *well-formed*. The most important subcomponents of this definition are properties 2(c) and 3. More specifically, property 3 holds only if the access sets of different activation records are disjoint. Thus, property 3 implies that in a well-formed configuration no two activation records have permission to access the same location at the same time. Property 2(c) is the frame property that lies at the heart of our approach. This property states that in each non-top activation record $(\Gamma_i, A_i, G_i, B_i, \overline{s_i})$, the access set A_i frames the global heap H with respect to the old heap G_{i-1} : the value of each location in A_i has the same value in H and in G_{i-1} . In other words, property 2(c) guarantees that locations which are accessible to a caller but which are not required to be accessible by a callee's precondition cannot be modified by the callee. Note that f_D denotes the subset of f with domain D .

Definition 14. *A configuration*

$$\sigma = (H, (\Gamma_1, A_1, G_1, B_1, \overline{s_1}) \cdot \dots \cdot (\Gamma_k, A_k, G_k, B_k, \overline{s_k}))$$

is well-formed (denoted $\text{wf}(\sigma)$) if all of the following hold:

1. For the top activation record $(\Gamma_1, A_1, G_1, B_1, \overline{s_1})$, the following hold:
 - (a) The triple (H, Γ_1, A_1) is well-formed.
 - (b) The free variables of $\overline{s_1}$ are in the domain of Γ_1 .
 - (c) The heap H is a successor of G_1 .
2. For each non-top activation record $(\Gamma_i, A_i, G_i, B_i, \overline{s_i})$ with $i \in \{2, \dots, k\}$ the following hold:
 - (a) The triple $(G_{i-1}, \Gamma_i, B_{i-1})$ is well-formed.
 - (b) The free variables of $\overline{s_i}$ are in the domain of Γ_i .
 - (c) The access set A_i frames the heap H wrt G_{i-1} : $H|_{A_i} = G_{i-1}|_{A_i}$.
 - (d) The heap G_{i-1} is a successor of G_i .
 - (e) The first statement of $\overline{s_i}$ is a method invocation $e_0.m(e_1, \dots, e_n)$; For each e_j where $j \in \{0, \dots, n\}$, evaluation does not get stuck and yields a value v_j . If the formal parameters of m are x_1, \dots, x_n , then Γ_{i-1} maps **this** to v_0 and each x_j to v_j . Finally, computing the required access set of the precondition of m does not get stuck and yields some set s . B_{i-1} is the union of A_i and s .
3. The access sets of different activation records are disjoint.

$$\forall i, j \in \{1, \dots, k\} \bullet i \neq j \Rightarrow A_i \cap A_j = \emptyset$$

The proof of preservation of well-formedness relies on two additional properties. Theorem 1 states that if an assertion ϕ holds in a context H, Γ, A , then

its required access set is a subset of A . Note that this implies that \vdash_R computes a lower bound on A for ϕ to hold: if the required access set is not a subset of A , then ϕ does not hold. Theorem 2 states that expressions never evaluate to non-allocated, non-null object references in well-formed contexts.

Theorem 1. *Suppose H, Γ, A is a well-formed triple, the free variables of ϕ are in Γ , ϕ evaluates to **true** and the required access set of ϕ is some set s . Then, s is a subset of A .*

$$\begin{aligned} \forall H, \Gamma, A, \phi, s \bullet \text{wf}(H, \Gamma, A) \wedge \text{free}(\phi) \subseteq \text{dom}(\Gamma) \wedge \\ H, \Gamma, A \vdash \phi \Downarrow \mathbf{true} \wedge H, \Gamma, A \vdash_R \phi \Downarrow s \\ \Downarrow \\ s \subseteq A \end{aligned}$$

Proof. By induction on the size of the derivation $H, \Gamma, A \vdash_R \phi \Downarrow s$ and case analysis on the kind of assertion.

- **True.** Suppose the assertion is **true**. Trivial, since the required access set of **true** is the empty set.
- **Access assertion.** Suppose the assertion is $\text{acc}(e.f)$. Moreover, assume the value of e is v . Since $\text{acc}(e.f)$ evaluates to **true**, it follows that (1) $(v, f) \in A$. Furthermore, (2) the required access set of $\text{acc}(e.f)$ is the singleton $\{(v, f)\}$. From (1) and (2), we can deduce that the required access set is a subset of A .
- **Conjunction.** Suppose the assertion is $\phi_1 \wedge \phi_2$. The derivations for ϕ_1 and ϕ_2 are both smaller than the derivation for $\phi_1 \wedge \phi_2$. Therefore, the required access sets of ϕ_1 and ϕ_2 are subsets of A . It follows that the union of both required access sets is a subset of A .
- **Separating Conjunction.** Suppose the assertion is $\phi_1 * \phi_2$. The argument for proving regular conjunction also applies here.
- **Equality.** Suppose the assertion is $e_1 = e_2$. Trivial, since the required access set of $e_1 = e_2$ is the empty set.
- **Predicate instance.** Suppose the assertion is $e_0.q(e_1, \dots, e_n)$ and that q 's body is ϕ . Since the derivation $H, [\mathbf{this} \mapsto v_0, \dots, x_n \mapsto v_n], A \vdash_R \phi \Downarrow s$ is smaller than the one for $e_0.q(e_1, \dots, e_n)$, the required access set is a subset of A .

□

Theorem 2. *Suppose H, Γ, A is a well-formed triple, the free variables of e are in Γ and e evaluates to an object reference o . Then, o is either null or o is allocated in H .*

$$\begin{aligned} \forall H, \Gamma, A, e, o \bullet \text{wf}(H, \Gamma, A) \wedge \text{free}(e) \subseteq \text{dom}(\Gamma) \wedge H, \Gamma, A \vdash e \Downarrow o \\ \Downarrow \\ o = \text{null} \vee o \in \text{dom}(H) \end{aligned}$$

Proof. By induction on the size of the derivation of $H, \Gamma, A \vdash e \Downarrow o$ and case analysis on the kind of expression.

- **Null.** Suppose the expression is *null*. Trivial.
- **Variable.** Suppose the expression is x . The environment is well-formed, hence it follows that $\Gamma(x)$ is either *null* or allocated in H .
- **Field read.** Suppose the expression is $e.f$ and that e evaluates to v . $(v, f) \in \text{dom}(H)$ and H is well-formed. Therefore, $H(v, f)$ is either *null* or allocated in H .
- **Normal pure method invocation.** Suppose the expression is a pure method invocation $e_0.p(e_1, \dots, e_n)$. For each $i \in \{0, \dots, n\}$, the derivation for e_i is smaller than the derivation of $e_0.p(e_1, \dots, e_n)$. Therefore, the value of e_i is either *null* or allocated in H . As a consequence, the environment $[\mathbf{this} \mapsto v_0, x_1 \mapsto v_1, \dots, x_n \mapsto v_n]$ is well-formed wrt H . Suppose the body of p is e . The derivation for e is smaller than the derivation of $e_0.p(e_1, \dots, e_n)$. It follows that e 's value (which is also the value of $e_0.p(e_1, \dots, e_n)$) is either *null* or allocated in H .

□

We can now prove that well-formed is preserved by \rightarrow .

Theorem 3 (Preservation of well-formedness). *In a well-formed program, \rightarrow preserves well-formedness.*

$$\forall \sigma, \sigma' \bullet \text{wf}(\sigma) \wedge \sigma \rightarrow \sigma' \Rightarrow \text{wf}(\sigma')$$

Proof. By case analysis on the step taken.

- **Field update.** Suppose the statement sequence of the top activation records in σ is $e_1.f := e_2; \bar{s}$.
 1. (a) By Theorem 2, if the value of e_2 is a non-null object reference, then it is allocated in H . Therefore, $(H[(v_1, f) \mapsto v_2], \Gamma_1, A_1)$ is a valid triple.
 - (b) Trivial
 - (c) Trivial.
 2. (a) Trivial.
 - (b) Trivial.
 - (c) Since (v_1, f) is the access set of the top activation record, it is not in the access set of any other activation record. Moreover, no old heap or access set was changed. Therefore, $H|_{A_i} = G_{i-1}|_{A_i}$, for each $i \in \{2, \dots, k\}$ (where k is the number of activation records).
 - (d) Trivial.
 - (e) Trivial.

3. Trivial, no access set was modified.
- **Mutator invocation.** Suppose the statement sequence of the top activation record in σ is $e_0.m(e_1, \dots, e_l); \bar{s}$.
 1. (a) The heap is not changed, so it remains well-formed. The environment of the new top activation record is well-formed wrt H , because if the value of some actual parameter e_i is a non-null object reference, then it is allocated (Theorem 2). Finally, the access set of the new top activation record is well-formed because it is a subset of the old access set (Theorem 1).
 - (b) Follows from the well-formedness of programs, i.e. a free variable of a method body is either **this** or a method parameter.
 - (c) Trivial.
 2. We only need to consider the old top activation record.
 - (a) Trivial.
 - (b) Trivial.
 - (c) Trivial, since the old heap of the new top activation record equals the heap.
 - (d) Trivial, since the old heap of the new top activation record equals the heap.
 - (e) Follows immediately from the step rule for method invocation.
 3. The access set of the new top activation record is the required access set of the precondition of the method. By Theorem 1, this access set is a subset of the the access set of the old top activation record. Therefore, the access set of the old top activation record is divided into two access set, and Property 3 is preserved.
 - **Return.** Suppose the statement sequence of the top activation record is empty and that the size of the stack is larger than 1.
 1. (a) The current heap is well-formed. Because the environment of the activation record second from the top is well-formed wrt to the old heap and the current heap is a successor of the old heap, the environment is well-formed wrt the current heap. The access set of the new top activation record is the union of the required access set of the postcondition A_Q and the access set of the old activation record second from the top A_2 . A_Q is well-formed wrt H , because it is a subset of the access set of the old top activation record (Theorem 1). A_2 is well-formed wrt H , since A_2 is well-formed wrt the old heap of the old top activation record and the current heap is a successor of that heap. Therefore, the union of A_Q and A_2 is well-formed wrt H .
 - (b) Trivial.
 - (c) Trivial, because the successor relation is transitive.

2. Trivial, because we just pop an activation record from the stack and do not modify any old heaps or old access sets.
 3. Suppose A_Q is the required access set of the postcondition of the old top activation record. It follows from Theorem 1 that A_Q is a subset of the access of the old top activation record. Since the old configuration was well-formed, the union of A_Q and A_2 is disjoint from the access set of any other activation record.
- **Object creation.** Suppose the statement sequence of the top activation records in σ is $x := \mathbf{new} C; \bar{s}$.
 1. (a) H remains well-formed, because the heap is extended with new locations that either hold *null* or an integer. Γ remains well-formed because o is allocated in the new heap. A remains well-formed because all locations added to A are fields of the newly allocated object o .
 - (b) Trivial.
 - (c) Trivial, since no location is removed from the heap.
 2. Trivial, since only the access set and statement list of the top activation record and the heap are modified.
 3. In σ_1 , all access sets are disjoint (property 3). Moreover, each access set only contains locations allocated in H (properties 1(a), 2(a) and 2(d)). Therefore, locations of the new object o are not in any access set of σ_1 . Hence, adding those locations to the access set of the top activation record preserves disjointness.

□

Theorem 4. *Executions of well-formed programs reach only well-formed states.*

Proof. Follows immediately from the well-formedness of the initial state and Theorem 3. □

5.2 Interpretation

We interpret the verification logic using the interpretation \mathcal{J} . First of all, \mathcal{J} maps each sort to a set.

| sort | set |
|--------------|--|
| <i>val</i> | $\mathcal{V} = \mathcal{O} \cup \mathbb{B}$ |
| <i>ref</i> | \mathcal{O} |
| <i>bool</i> | \mathbb{B} |
| <i>set</i> | $\mathcal{P}(\mathcal{O} \times \mathcal{F})$ |
| <i>cname</i> | \mathcal{C} |
| <i>fname</i> | \mathcal{F} |
| <i>heap</i> | $\mathcal{O} \times \mathcal{F} \rightarrow \mathcal{O}$ |

Secondly, \mathcal{J} maps each function symbol in the signature to a function. The interpretation for the built-in functions is as expected. We assume \mathcal{J} is a model for the prelude axioms. That is, $\mathcal{J} \models \Sigma_{\text{prelude}}$.

| function | interpretation |
|---------------------------------|---|
| $null$ | $null$ |
| $emptyset$ | \emptyset |
| $singleton(o, f)$ | $\{ (o, f) \}$ |
| $union(s_1, s_2)$ | $s_1 \cup s_2$ |
| $contains(s, o, f)$ | $(o, f) \in s$ |
| $setminus(s_1, s_2)$ | $s_1 \setminus s_2$ |
| $select(h, o, f)$ | $h(o, f)$ |
| $store(h, o, f, v)$ | $h[(o, f) \mapsto v]$ |
| $allocated(h, o)$ | $o \in \text{dom}(h)$ |
| $allocate(o, h, C)$ | $h[(o, f_1) \mapsto null, \dots, (o, f_n) \mapsto null]$ where $\text{fields}(\mathcal{J}(C)) = C_1 f_1, \dots, C_n f_n$ |
| $ok(h, a)$ | $\text{wf}(h) \wedge \text{wf}(a, h)$ |
| $\text{succ}(h_1, a_1, h_2, a)$ | $\text{succ}(h_1, a_1, h_2, a_2)$ |
| $ttrue$ | true |

We interpret the program-specific functions as follows. The constant C is interpreted as the class name C . Similarly, the constant f is interpreted as the field name f . The interpretation of the other function symbols is as follows:

- $\mathcal{J}(C.p)(H, A, v_0, \dots, v_n)$ equals v if there exists a v such that the expression $v_0.p(v_1, \dots, v_n)$ evaluates to v in heap H and access set A (i.e. $H, [\mathbf{this} \mapsto v_0, x_1 \mapsto v_1, \dots, x_n \mapsto v_n], A \vdash \mathbf{this}.p(x_1, \dots, x_n) \Downarrow v$); otherwise, the result equals $null$.
- $\mathcal{J}(C.q)(H, A, v_0, \dots, v_n)$ equals the boolean b if there exists a b such that $v_0.q(v_1, \dots, v_n)$ evaluates to b in heap H and access set A (i.e. $H, [\mathbf{this} \mapsto v_0, x_1 \mapsto v_1, \dots, x_n \mapsto v_n], A \vdash \mathbf{this}.q(x_1, \dots, x_n) \Downarrow b$).
- $\mathcal{J}(C.q_{\text{FP}})(H, A, v_0, \dots, v_n)$ equals the set s if there exists a s such that the required access set of $v_0.q(v_1, \dots, v_n)$ evaluates to s in heap H and access set A (i.e. $H, [\mathbf{this} \mapsto v_0, x_1 \mapsto v_1, \dots, x_n \mapsto v_n], A \vdash_R \mathbf{this}.q(x_1, \dots, x_n) \Downarrow s$).

We write $\mathcal{J}, H, \Gamma, A \models \psi$ to indicate the formula ψ holds under a certain interpretation. More specifically, $\mathcal{J}, H, \Gamma, A \models \psi$ holds if ψ is true under an interpretation that maps each function to its interpretation in \mathcal{J} , that maps h to H , a to A and each other variable to its value in Γ .

5.3 Truth of Axioms

Before we start proving soundness, we have to prove that $\mathcal{J} \models \Sigma_\pi$, if π is valid.

Theorem 5. *Suppose π is a valid program and H, Γ, A is a well-formed triple. Then all of the following hold.*

- Suppose e is an expression whose free variables are in the domain of Γ such that $\mathfrak{J}, H, \Gamma, A \models \text{Df}(e)$ and $\mathfrak{J} \models \Sigma_{p^*}$, where p is the largest pure method appearing in e . Then evaluation of e does not get stuck. Furthermore, if $H, \Gamma, A \vdash e \Downarrow v$, then $\mathfrak{J}, H, \Gamma, A \models \text{Tr}(e) = v$.
- Suppose ϕ is an assertion whose free variables are in the domain of Γ such that $\mathfrak{J}, H, \Gamma, A \models \text{Df}(\phi)$, and $\mathfrak{J} \models \Sigma_{p^*}$, where p is the largest pure method appearing in ϕ . Then evaluation of ϕ does not get stuck. Furthermore, if $H, \Gamma, A \vdash \phi \Downarrow b$, then $\mathfrak{J}, H, \Gamma, A \models \text{Tr}(\phi) = b$. In addition, if $\mathfrak{J}, H, \Gamma, A \models \text{Tr}(\phi)$, then computing the required access set of ϕ does not get stuck. Moreover, $\mathfrak{J}, H, \Gamma, A \models \mathbf{R}(\phi) = s$, where $H, \Gamma, A \vdash_R \phi \Downarrow s$.

Proof. By induction on the size of the expression (respectively assertion). We perform case analysis on the kind of expression (respectively assertion).

- **Null.** Suppose the expression is *null*. Evaluation of *null* is never stuck. It immediately follows from the definitions of \mathfrak{J} and Tr that the interpretation of the translation equals *null*.
- **Variable.** Suppose the expression is a variable x . Evaluation of a variable never gets stuck. Moreover, x evaluates to $\Gamma(x)$. Since $\text{Tr}(x) = x$, it follows that $\mathfrak{J}, H, \Gamma, A \models \text{Tr}(x) = \Gamma(x)$.
- **Field access.** Suppose the expression is a field read $e.f$. Since $e.f$ is well-defined, it follows that

$$\mathfrak{J}, H, \Gamma, A \models \text{Df}(e) \wedge (\text{Tr}(e), f) \in a$$

Since e is syntactically smaller than $e.f$ and $\mathfrak{J}, H, \Gamma, A \models \text{Df}(e)$, it follows that execution of e does not get stuck and yields some value v . Moreover, $\mathfrak{J}, H, \Gamma, A \models \text{Tr}(e) = v$. Because e does not get stuck and $\mathfrak{J}, H, \Gamma, A \models (v, f) \in a$, it follows that $e.f$ does not get stuck. Finally, the value of $e.f$ is $H(v, f)$. This is also the interpretation of the translation.

- **Normal pure method invocation.** Suppose the expression is a normal pure method invocation $e_0.p(e_1, \dots, e_n)$. Since $e_0.p(e_1, \dots, e_n)$ is well-defined, it follows that

$$\begin{aligned} \mathfrak{J}, H, \Gamma, A \models & \text{Df}(e_0) \wedge \dots \wedge \text{Df}(e_n) \wedge \text{Tr}(e_0) \neq \text{null} \wedge \\ & \text{Tr}(\text{mpre}(C, p)[e_0/\mathbf{this}, e_1/x_1, \dots, e_n/x_n]) \end{aligned}$$

Since each e_i (with $i \in \{0, \dots, n\}$) is syntactically smaller than the expression $e_0.p(e_1, \dots, e_n)$ and $\mathfrak{J}, H, \Gamma, A \models \text{Df}(e_i)$, it follows that execution of e_i does not get stuck and yields some value v_i such that $\mathfrak{J}, H, \Gamma, A \models \text{Tr}(e_i) = v_i$. Moreover, $v_0 \neq \text{null}$ so $e_0.p(e_1, \dots, e_n)$ does not get stuck because the receiver is *null*. Since π is a valid program, p is a valid normal pure method:

$$\begin{aligned} & \Sigma_{\text{prelude}} \cup \Sigma_{p^*} \vdash \forall h, a, \mathbf{this}, x_1, \dots, x_n \bullet \\ & \text{ok}(h, a) \wedge \mathbf{this} \neq \text{null} \wedge \text{allocated}_h(\mathbf{this}, x_1, \dots, x_n) \\ & \quad \Downarrow \\ & \text{Df}(\phi_1) \end{aligned}$$

Since \mathfrak{J} is a model for $\Sigma_{prelude} \cup \Sigma_{p^*}$, H is well-formed, v_0 is not equal to $null$ and each v_i is either null or allocated, it follows that $\mathfrak{J}, H, \Gamma, A \models \text{Df}(\phi_1)$. Since the precondition is smaller than the call $e_0.p(e_1, \dots, e_k)$ (well-formedness implies that the precondition does not call p itself), the precondition is well-defined, it follows that evaluation of the precondition does not get stuck. Because

$$\mathfrak{J}, H, \Gamma, A \models \text{Tr}(\text{mpre}(C, p)[e_0/\mathbf{this}, e_1/x_1, \dots, e_k/x_k])$$

, the precondition evaluates to **true**. As a consequence, computing the required access set of the precondition does not get stuck. We may conclude that $e_0.p(e_1, \dots, e_k)$ does not get stuck. It follows immediately from the definition of normal pure method evaluation (Figure 16) and the definition of \mathfrak{J} , that the interpretation of $\text{Tr}(e_0.p(e_1, \dots, e_k))$ equals the value of the evaluation.

- **True.** Suppose the assertion is **true**. Evaluation of **true** is never stuck. It immediately follows from the definitions of \mathfrak{J} and Tr that the interpretation of the translation equals *true*.
- **Access assertion.** Suppose the assertion is an access assertion $\mathbf{acc}(e.f)$. Since $\mathbf{acc}(e.f)$ is well-defined, it follows that

$$\mathfrak{J}, H, \Gamma, A \models \text{Df}(e) \wedge \text{Tr}(e) \neq null$$

Since e is syntactically smaller than $e.f$ and $\mathfrak{J}, H, \Gamma, A \models \text{Df}(e)$, it follows that execution of e does not get stuck and yields some value v . Moreover, $\mathfrak{J}, H, \Gamma, A \models \text{Tr}(e) = v$. Because e does not get stuck, it follows that $\mathbf{acc}(e.f)$ does not get stuck. The assertion $\mathbf{acc}(e.f)$ holds only if $(v, f) \in A$ which is also the interpretation of the translation. The required access set of $\mathbf{acc}(e.f)$ does not get stuck and yields the singleton $\{(v, f)\}$ which is also the interpretation of $\mathbf{R}(\mathbf{acc}(e.f))$.

- **Predicate invocation.** Suppose the assertion is a predicate invocation $e_0.q(e_1, \dots, e_n)$. The proof is similar to the one for normal pure method invocation.

□

Theorem 6 (Consistency). *If π is a valid program, then $\mathfrak{J} \models \Sigma_\pi$.*

Proof. Assume π is a valid program. Σ_π is the union of $\Sigma_{prelude}$ and the axioms of each pure method p in π . Since \mathfrak{J} models $\Sigma_{prelude}$, it suffices to show that \mathfrak{J} is a model for each Σ_p . We prove that latter property by constructing a set of pure methods S , such that if a pure method p is in S , then all pure methods defined before p in the program text are also in S . We define Σ_S as the union of the axioms of each pure method p in S . We proceed by induction on the size of S . If S is empty, then trivially, $\mathfrak{J} \models \Sigma_S$. If S is not empty, select the pure method p from S appearing last in the program text. It follows from the

induction hypothesis that \mathfrak{J} is a model for each pure method appearing before p in the program text. That is, $\mathfrak{J} \models \Sigma_{p^*}$. We have to show that \mathfrak{J} is a model for each of p 's axioms. p is either a normal pure method or a predicate. If p is a normal pure method, then p 's implementation and frame axioms must hold under \mathfrak{J} .

- **Implementation axiom.** We must show that

$$\begin{aligned} & \forall h, a, \text{this}, x_1, \dots, x_n \bullet \\ & ok(h, a) \wedge \text{this} \neq \text{null} \wedge \text{allocated}_h(\text{this}, x_1, \dots, x_n) \wedge \text{Tr}(\phi_1) \\ & \quad \Downarrow \\ & C.p(h, a, \text{this}, x_1, \dots, x_n) = \text{Tr}(e) \end{aligned}$$

Take an arbitrary heap H , access set A , object reference o and values v_1, \dots, v_n such that the following holds

$$\mathfrak{J}, H, \Gamma, A \models ok(h, a) \wedge \text{this} \neq \text{null} \wedge \text{allocated}_h(\text{this}, x_1, \dots, x_n) \wedge \text{Tr}(\phi_1)$$

Γ is a shorthand for $[\text{this} \mapsto o, x_1 \mapsto v_1, \dots, x_n \mapsto v_n]$. π is valid, hence p is a valid normal pure method. Furthermore, $\mathfrak{J} \models \Sigma_{\text{prelude}} \cup \Sigma_{p^*}$. Therefore, it follows that

$$\begin{aligned} & \mathfrak{J} \models \forall h, a, \text{this}, x_1, \dots, x_n \bullet \\ & ok(h, a) \wedge \text{this} \neq \text{null} \wedge \text{allocated}_h(\text{this}, x_1, \dots, x_n) \\ & \quad \Downarrow \\ & \text{Df}(\phi_1) \wedge (\text{Tr}(\phi_1) \Rightarrow \text{Df}(e)) \end{aligned}$$

By instantiating the quantifier above with H, Γ and A and since $\mathfrak{J}, H, \Gamma, A \models ok(h, a) \wedge \text{this} \neq \text{null} \wedge \text{allocated}_h(\text{this}, x_1, \dots, x_n) \wedge \text{Tr}(\phi_1)$, we get

$$\mathfrak{J}, H, \Gamma, A \models \text{Df}(e)$$

Because of Theorem 5, evaluation of e does not get stuck and yields some value v . It immediately follows from the definition of \mathfrak{J} for normal pure methods that

$$\mathfrak{J}, H, \Gamma, A \models C.p(h, a, \text{this}, x_1, \dots, x_n) = \text{Tr}(e)$$

- **Frame axiom.** We must show that

$$\begin{aligned} & \forall h_1, a_1, h_2, a_2, \text{this}, x_1, \dots, x_n \bullet \\ & ok(h_1, a_1) \wedge ok(h_2, a_2) \wedge \text{this} \neq \text{null} \wedge \text{allocated}_h(\text{this}, x_1, \dots, x_n) \wedge \\ & \quad \text{Tr}(\phi_1)[h_1/h, a_1/a] \wedge \text{Tr}(\phi_1)[h_2/h, a_2/a] \wedge \\ & (\forall o, f \bullet (o, f) \in \mathbf{R}(\phi_1)[h_1/h, a_1/a] \Rightarrow (o, f) \in a_2 \wedge h_1(o, f) = h_2(o, f)) \\ & \quad \Downarrow \\ & C.p(h_1, a_1, \text{this}, x_1, \dots, x_n) = C.p(h_2, a_2, \text{this}, x_1, \dots, x_n) \end{aligned}$$

Take two arbitrary heaps H_1 and H_2 , two arbitrary access sets A_1 and A_2 , object reference o and values v_1, \dots, v_n such that

$$\begin{aligned} \mathfrak{J}, \Gamma \models & \\ & \forall h_1, a_1, h_2, a_2, \text{this}, x_1, \dots, x_n \bullet \\ & ok(h_1, a_1) \wedge ok(h_2, a_2) \wedge \text{this} \neq \text{null} \wedge \text{allocated}_h(\text{this}, x_1, \dots, x_n) \wedge \\ & \text{Tr}(\phi_1)[h_1/h, a_1/a] \wedge \text{Tr}(\phi_1)[h_2/h, a_2/a] \wedge \\ & (\forall o, f \bullet (o, f) \in \mathbb{R}(\phi_1)[h_1/h, a_1/a] \Rightarrow (o, f) \in a_2 \wedge h_1(o, f) = h_2(o, f)) \end{aligned}$$

Γ is a shorthand for the environment $[\text{this} \mapsto o, x_1 \mapsto v_1, \dots, x_n \mapsto v_n, h_1 \mapsto H_1, \dots, a_2 \mapsto A_2]$. Suppose that (*)

$$\mathfrak{J}, \Gamma \models C.p(h_1, a_1, \text{this}, x_1, \dots, x_n) \neq C.p(h_2, a_2, \text{this}, x_1, \dots, x_n)$$

Then, there exists a location $o.f$ that is read during evaluation of p 's body that holds a different value in H_1 and H_2 . A location is only readable in the body if it is in the access set. The access set during evaluation of the body is a subset of the required access set of the precondition. However, all of those locations must have the same values in H_1 and H_2 , hence (*) is false and

$$\mathfrak{J}, \Gamma \models C.p(h_1, a_1, \text{this}, x_1, \dots, x_k) = C.p(h_2, a_2, \text{this}, x_1, \dots, x_k)$$

If p is a predicate, then p 's implementation, frame, footprint implementation, footprint frame and footprint accessible axioms must hold under \mathfrak{J} . In the remainder of the proof, we rename p to q .

- **Implementation axiom.** Similar to the proof for the implementation axiom of normal pure methods.
- **Frame axiom.** Similar to the proof for the frame axiom of normal pure methods.
- **Footprint implementation axiom.** Similar to the proof for the implementation axiom of normal pure methods.
- **Footprint frame axiom.** Similar to the proof for the frame axiom of normal pure methods.
- **Footprint accessible axiom.** We have to show that

$$\begin{aligned} \mathfrak{J} \models \forall h, a, \text{this}, x_1, \dots, x_n \bullet & \\ ok(h, a) \wedge \text{this} \neq \text{null} \wedge \text{allocated}_h(\text{this}, x_1, \dots, x_n) \wedge C.q(h, a, \text{this}, x_1, \dots, x_n) & \\ \Downarrow & \\ C.q_{\text{FP}}(h, a, \text{this}, x_1, \dots, x_n) \subseteq a & \end{aligned}$$

Take an arbitrary heap H , access set A , object reference o and values v_1, \dots, v_n such that the following holds

$$\mathfrak{J}, H, \Gamma, A \models ok(h, a) \wedge \text{this} \neq \text{null} \wedge C.q(h, a, \text{this}, x_1, \dots, x_n)$$

Γ is a shorthand for $[this \mapsto o, x_1 \mapsto v_1, \dots, x_n \mapsto v_n]$. Since $\mathfrak{J}, H, \Gamma, A \models \text{Df}(C.q(h, a, this, x_1, \dots, x_n))$ and $\mathfrak{J}, H, \Gamma, A \models C.q(h, a, this, x_1, \dots, x_n)$, it follows from Theorem 5 that $C.q(this, x_1, \dots, x_n)$ does not get stuck and yields **true**. It now follows immediately from Theorem 1 that the footprint contains only accessible locations. \square

5.4 Soundness

We define the notion of valid configuration (Definition 15). A configuration is *valid* if the verification conditions of the statements in each activation record hold under a certain interpretation. That is, the verification condition of the top activation record must hold when interpreting the h as the global H and a as the access set of the top activation record. Similarly, the verification condition of the i th (with i between 2 and the size of the stack) activation record must hold when interpreting h as G_{i-1} and the access set as B_{i-1} . G_{i-1} and B_{i-1} are respectively the old heap and old access set of the activation record directly above the i th activation record.

Definition 15. *A configuration*

$$\sigma = (H, (\Gamma_1, A_1, G_1, B_1, \overline{s_1}) \cdot \dots \cdot (\Gamma_k, A_k, G_k, B_k, \overline{s_k}))$$

is valid (denoted $\text{valid}(\sigma)$) if all of the following hold:

- σ is well-formed.
- The verification condition of the top activation record holds in the heap H , the environment Γ_1 and the access set A_1 .

$$\mathfrak{J}, H, \Gamma_1, A_1 \models \text{vc}(\overline{s_1}, \psi_1)$$

- For each non-top activation record $(\Gamma_i, A_i, G_i, B_i, \overline{s_i})$ (with $i \in \{2, \dots, k\}$), the verification condition holds in the heap H_{i-1} , the environment Γ_i and the access set B_{i-1} .

$$\forall i \in \{2, \dots, k\} \bullet \mathfrak{J}, H_{i-1}, \Gamma_i, B_{i-1} \models \text{vc}(\overline{s_i}, \psi_i)$$

ψ_i is the postcondition of the method being executed in the i th activation record or true for main routine's activation record.

Finally, we can show that validity is preserved by \rightarrow (Theorem 7) and that valid configurations are never stuck (Theorem 8). Since the initial configuration in a valid program is valid, it follows that executions of valid programs never get stuck.

Theorem 7 (Preservation). *In an execution of a valid program, the small-step relation \rightarrow preserves validity.*

$$\forall \sigma_1, \sigma_2 \bullet \text{valid}(\sigma_1) \wedge \sigma_1 \rightarrow \sigma_2 \Rightarrow \text{valid}(\sigma_2)$$

Proof. By case analysis on the step taken.

- **Field update.** Suppose the statements of the top activation record in σ_1 are $e_1.f := e_2; \bar{s}$. It follows from the validity of σ_1 that (1) $\mathfrak{J}, H, \Gamma_1, A_1 \models \text{Df}(e_1)$, (2) $\mathfrak{J}, H, \Gamma_1, A_1 \models \text{Df}(e_2)$ and (3) $\mathfrak{J}, H, \Gamma_1, A_1 \models \text{vc}(\bar{s}, \psi_1)[h[(\text{Tr}(e_1), f) \mapsto \text{Tr}(e_2)]]/h]$. Because of (1), (2) and Theorem 5, evaluation of e_1 and e_2 does not get stuck and yields some values v_1 and v_2 , such that (4) $\mathfrak{J}, H, \Gamma_1, A_1 \models \text{Tr}(e_1) = v_1 \wedge \text{Tr}(e_2) = v_2$. It follows from Theorem 5, (3) and (4) that $\mathfrak{J}, H[(v_1, f) \mapsto v_2], \Gamma_1, A_1 \models \text{vc}(\bar{s}, \psi)$. Therefore, σ_2 is valid.
- **Mutator invocation.** Suppose the statements of the top activation record in σ_1 are $e_0.m(e_1, \dots, e_k); \bar{s}$. It follows immediately from the rule for method invocation (Figure 15) and Definition 15 that the old top activation record remains valid. Since π is a valid program, m is a valid mutator (Definition 2). In particular, m 's body satisfies its method contract:

$$\Sigma_\pi \vdash \forall h, a, \text{this}, x_1, \dots, x_k \bullet \text{ok}(h, a) \wedge \text{this} \neq \text{null} \wedge \text{Tr}(\phi_1) \Rightarrow \text{vc}(\bar{r}, \text{Tr}(\phi_2))$$

where \bar{r} is the body of m . Suppose A' is the required access set of m 's precondition. Since $\mathfrak{J} \models \Sigma_\pi$ and by instantiating h with H and a with A' , we get

$$\mathfrak{J}, H, \Gamma, A' \models \text{ok}(h, a) \wedge \text{this} \neq \text{null} \wedge \text{Tr}(\phi_1) \Rightarrow \text{vc}(\bar{r}, \text{Tr}(\phi_2))$$

Since $\text{ok}(H, A')$, the receiver is not null and the precondition holds, it follows that

$$\mathfrak{J}, H, \Gamma, A' \models \text{vc}(\bar{r}, \text{Tr}(\phi_2))$$

That is, the new top activation record is valid. We may conclude that σ_2 is valid.

- **Return.** Suppose the statement list of the top activation record in σ_1 is empty. Since σ_1 's activation record second from the top is valid, it follows that

$$\begin{aligned} & \mathfrak{J}, G_1, \Gamma_2, B_1 \models \text{Df}(e_0) \wedge \dots \wedge \text{Df}(e_n) \wedge \text{Tr}(e_0) \neq \text{null} \wedge \text{Tr}(P) \wedge \\ & (\forall h', a' \bullet \\ & \quad \text{succ}(h, a, h', a') \wedge \\ & \quad \text{Tr}(Q)[h'/h, a'/a] \wedge \\ & \quad (\forall o, f \bullet (o, f) \in a \wedge (o, f) \notin \text{R}(P) \Rightarrow (o, f) \in a' \wedge h(o, f) = h'(o, f)) \wedge \\ & \quad (\forall o, f \bullet (o, f) \in \text{R}(Q)[h'/h, a'/a] \Rightarrow (o, f) \in \text{R}(P) \vee \neg(o, f) \in a) \\ & \Rightarrow \\ & \quad \text{vc}(\bar{s}, \psi)[h'/h, a'/a] \end{aligned}$$

where P is a shorthand for $\text{mpre}(C, m)[e_0/\text{this}, e_1/x_1, \dots, e_n/x_n]$ and Q is $\text{mpost}(C, m)[e_0/\text{this}, e_1/x_1, \dots, e_n/x_n]$. Suppose A'' is the new access

set of σ_2 's top activation record. Let's instantiate h with H and a' with A'' .

$$\begin{aligned}
& \mathfrak{J}, G_1, \Gamma_2[h' \mapsto H, a' \mapsto A''], B_1 \models \\
& \quad succ(h, a, h', a') \wedge \\
& \quad Tr(Q)[h'/h, a'/a] \wedge \\
& \quad (\forall o, f \bullet (o, f) \in a \wedge (o, f) \notin R(P) \Rightarrow (o, f) \in a' \wedge h(o, f) = h'(o, f)) \wedge \\
& \quad (\forall o, f \bullet (o, f) \in R(Q)[h'/h, a'/a] \Rightarrow (o, f) \in R(P) \vee \neg(o, f) \in a) \\
& \quad \Rightarrow \\
& \quad vc(\bar{s}, \psi)[h'/h, a'/a]
\end{aligned}$$

Since σ_1 is well-formed, $\mathfrak{J}, G_1, \Gamma_2[h' \mapsto H, a' \mapsto A''], B_1 \models succ(h, a, h', a')$. Since a step was taken, the postcondition evaluated to true. In other words, $\mathfrak{J}, G_1, \Gamma_2[h' \mapsto H, a' \mapsto A''], B_1 \models Tr(Q)[h'/h, a'/a]$. The fact that locations in the access set of the caller and not required to be accessible by the callee remain accessible and retain their value follows from well-formedness properties 2(c) and 2(e). More specifically, the set of locations $a \setminus R(P)$ is exactly A_2 (property 2(e)). In addition, we can deduce from property 2(c) that those locations are not modified. Hence, $\mathfrak{J}, G_1, \Gamma_2[h' \mapsto H, a' \mapsto A''], B_1 \models (\forall o, f \bullet (o, f) \in a \wedge (o, f) \notin R(P) \Rightarrow (o, f) \in a' \wedge h(o, f) = h'(o, f))$. Finally, we know that $R(Q)[h'/h, a'/a]$ is a subset of A_1 (Theorem 1). Moreover, $B_1 = A_2 \cup R(P)$ (property 2(e)) and $A_1 \cap A_2 = \emptyset$ (property 2(c)). Therefore, $(B_1 \setminus R(P)) \cap R(Q)[h'/h, a'/a] = \emptyset$. It immediately follows that $\mathfrak{J}, G_1, \Gamma_2[h' \mapsto H, a' \mapsto A''], B_1 \models (\forall o, f \bullet (o, f) \in R(Q)[h'/h, a'/a] \Rightarrow (o, f) \in R(P) \vee \neg(o, f) \in a)$. We may conclude that the top activation record of σ_2 is valid, and hence that σ_2 itself is valid. \square

Theorem 8 (Progress). *Suppose σ_1 is a valid, non-final configuration of a valid program π . There exists a configuration σ_2 , such that $\sigma_1 \rightarrow \sigma_2$.*

$$\forall \sigma_1 \bullet \text{valid}(\sigma_1) \wedge \neg \text{final}(\sigma_1) \Rightarrow (\exists \sigma_2 \bullet \sigma_1 \rightarrow \sigma_2)$$

Proof. By case analysis on the first statement of the top activation record of σ .

- **Field update.** Suppose the statements of the top activation record are $e_1.f := e_2; \bar{s}$. Since σ_1 's top activation record is valid, it follows that

$$\begin{aligned}
& \mathfrak{J}, H, \Gamma_1, A_1 \models Df(e_1) \wedge Df(e_2) \wedge (Tr(e_1), f) \in a \wedge \\
& \quad vc(\bar{s}, \psi_1)[h[(Tr(e_1), f) \mapsto Tr(e_2)]/h]
\end{aligned}$$

e_1 and e_2 are well-defined, thus it follows from Theorem 5 that their evaluation does not get stuck. Assume e_1 evaluates to v_1 and e_2 to v_2 . Theorem 5 also implies that $\mathfrak{J}, H, \Gamma_1, A_1 \models Tr(e_1) = v_1$. Hence, $(v_1, f) \in A_1$ and σ_1 is not stuck.

- **Mutator invocation.** Suppose the statements of the top activation record are $e_0.m(e_1, \dots, e_k); \bar{s}$. Since σ_1 's top activation record is valid, it

follows that

$$\mathfrak{J}, H, \Gamma_1, A_1 \models \text{Df}(e_0) \wedge \dots \wedge \text{Df}(e_n) \wedge \text{Tr}(e_0) \neq \text{null} \wedge \text{Tr}(P)$$

where P is $\text{mpre}(C, m)[e_0/\mathbf{this}, e_1/x_1, \dots, e_n/x_n]$. Since each e_i (with $i \in \{0, \dots, k\}$) is well-defined and Theorem 5, evaluation of e_i does not get stuck and the interpretation of $\text{Tr}(e_i)$ equals v_i . It then also follows that the receiver is not null. Since π is a valid program, it follows that the m 's precondition is well-defined.

$$\Sigma_{\text{prelude}} \cup \Sigma_{p^*} \vdash \forall h, a, \text{this}, x_1, \dots, x_k \bullet \text{ok}(h, a) \wedge \text{this} \neq \text{null} \Rightarrow \text{Df}(\text{mpre}(C, m))$$

Because $\mathfrak{J} \models \Sigma_\pi$ and by instantiating the quantifier we get

$$\mathfrak{J}, H, \Gamma'_1, A_1 \models \text{ok}(h, a) \wedge \text{this} \neq \text{null} \Rightarrow \text{Df}(\text{mpre}(C, m))$$

where Γ'_1 is $[\text{this} \mapsto v_0, x_1 \mapsto v_1, \dots, x_n \mapsto v_k]$. Since the precondition is well-defined and Theorem 5, its evaluation and computing its required access set does not get stuck. Moreover, because

$$\mathfrak{J}, H, \Gamma_1, A_1 \models \text{Tr}(P)$$

the precondition returns **true**. We may conclude that σ_1 is not stuck.

- **Return.** Suppose the statement list of the top activation record is empty and that the stack contains at least two activation records. Since σ_1 's top activation record is valid, it follows that (1)

$$\mathfrak{J}, H, \Gamma_1, A_1 \models \text{mpost}(C, m)$$

where $C.m$ is the method being executed in the top activation record. Since π is a valid program, it follows that the postcondition of $C.m$ is well-defined, provided the precondition held in the pre-state (Definition 2).

$$\begin{aligned} & \Sigma_\pi \vdash \forall h, a, h', a', \text{this}, x_1, \dots, x_k \bullet \\ & \text{ok}(h, a) \wedge \text{succ}(h, a, h', a') \wedge \text{this} \neq \text{null} \wedge \text{Tr}(\text{mpre}(C, m)) \\ & \quad \downarrow \\ & \text{Df}(\text{mpost}(C, m))[h'/h, a'/a] \end{aligned}$$

Since $\mathfrak{J} \models \Sigma_\pi$ (Theorem 6) and by instantiating h with G_1 , a with B_1 , h' with H and a' with A_1 , we get

$$\begin{aligned} & \mathfrak{J}, G_1, \Gamma_1[h' \mapsto H, a' \mapsto A_1], B_1 \models \\ & \text{ok}(h, a) \wedge \text{succ}(h, a, h', a') \wedge \text{this} \neq \text{null} \wedge \text{Tr}(\text{mpre}(C, m)) \\ & \quad \downarrow \\ & \text{Df}(\text{mpost}(C, m))[h'/h, a'/a] \end{aligned}$$

Since σ_1 is well-formed and the activation record second from the top is valid, it follows that

$$\begin{aligned} & \mathfrak{J}, G_1, \Gamma_1[h' \mapsto H, a' \mapsto A_1], B_1 \models \\ & \text{ok}(h, a) \wedge \text{succ}(h, a, h', a') \wedge \text{this} \neq \text{null} \wedge \text{Tr}(\text{mpre}(C, m)) \end{aligned}$$

Therefore, the postcondition of $C.m$ is well-defined:

$$\mathfrak{J}, H, \Gamma_1, A_1 \models \text{Df}(\text{mpost}(C, m))$$

Theorem 5, the well-formedness of the postcondition and the fact that the postcondition holds imply that the postcondition returns **true** and computing its required access set does not get stuck. Hence, σ_1 is not stuck.

□

Corollary 1 (Soundness). *Valid programs do not get stuck.*

Proof. Follows immediately from the fact that the initial state is valid, and Theorems 7 and 8. □

6 Inheritance

Inheritance is a key component of the object-oriented paradigm that allows a class to be defined as an extension of one or more existing classes. For example, consider the class *BackupCell* from Figure 19. *BackupCell* is an extension of its superclass *Cell*. More specifically, a *BackupCell* satisfies all properties of *Cell*, but can additionally undo the last call to *setX*. Reasoning about inheritance in verification is challenging, since the code to be executed for a call is not determined at compile-time but at run-time (depending on the type of the receiver). In particular, one must ensure that the contract used when verifying the call and the contract used when verifying the callee match. In this section, we extend the approach described in the previous sections to inheritance. The encoding of inheritance described here is similar to earlier proposals by Jacobs *et al.* [19], Leavens *et al.* [23] and Parkinson *et al.* [40].

Method calls can both be statically and dynamically bound, depending on the method itself and the calling context. For example, the call to *getX* in the client code of Figure 3(b) is dynamically bound, while the call to *getX* in the body of *BackupCell.setX* is statically bound. In Java, constructors, private methods and super calls are statically bound, while all other calls are dynamically bound. To distinguish statically bound calls of pure methods from dynamically bound ones, we introduce an additional function symbol in the verification logic. More specifically, for a pure method p defined in a class C , the signature not only includes a symbol $C.p$ but additionally contains a function symbol $C.p_D$. The former symbol is used to encode statically bound calls, while the latter is used for dynamically bound calls.

The relationship between $C.p$ and $C.p_D$ is encoded via a number of axioms. More specifically, $C.p$ equals $C.p_D$ whenever the dynamic type of the receiver

```

class BackupCell extends Cell {
  int backup;

  BackupCell()
    ensures valid() ∧ getX() = 0;
  { super(); }

  void setX(int v)
    requires valid();
    ensures valid();
    ensures getX() = v ∧ getBackup() = old(getX());
  { backup := super.getX(); super.setX(v); }

  void undo()
    requires valid();
    ensures valid() ∧ getX() = old(getBackup());
  { super.setX(backup); }

  predicate bool valid()
  { return acc(backup) * super.valid(); }

  pure int getBackup()
    requires valid();
  { return backup; }
}

```

Figure 19: The class *BackupCell* annotated with implicit dynamic frames annotations.

equals C .

$$\begin{array}{c}
\forall h, a, this, x_1, \dots, x_n \bullet \\
\text{type}(this) = C \\
\Downarrow \\
C.p(h, a, this, x_1, \dots, x_n) = C.p_D(h, a, this, x_1, \dots, x_n)
\end{array}$$

Note that *type* is a function that returns the dynamic type of an object. Furthermore, whenever a method $D.p$ overrides $C.p$, we add the following axiom to our theory:

$$\begin{array}{c}
\forall h, a, this, x_1, \dots, x_n \bullet \\
\text{type}(this) <: D \\
\Downarrow \\
C.p_D(h, a, this, x_1, \dots, x_n) = D.p_D(h, a, this, x_1, \dots, x_n)
\end{array}$$

The above axiom encodes the property that dynamically bound calls to $C.p$ and $D.p$ are equal if the receiver is a subtype (denoted $<:$) of D .

Invocations of pure methods with receiver **this** are treated differently in contracts and code. In normal code, such calls follow the standard rules that guide binding. However, if the **this**-call appears in the method contract of a statically bound call of a method m , then it is treated as statically bound; otherwise the **this**-call is dynamically bound. Method implementations are verified under the assumption that the corresponding method is called statically. This assumption is sound provided each subclass overrides each instance method of its superclass. Indeed, if an actual call is statically bound, then the contract seen by the caller and the contract for verifying the implementation are equal. If the actual call is dynamically bound, then the dynamic type of the receiver equals the static type of **this** for the callee and therefore the static contract equals the dynamic one (follows from the first inheritance axiom). In Figure 19, *BackupCell* overrides each method except *getX*. If a class does not override a method m , we generate a default method override which simply calls the superclass and inherits the superclass contract as is. This method is subject to verification like other methods.

Note that *BackupCell* does not depend on internal details of its superclass. As a consequence, changing *Cell*'s internal representation (within the boundaries set by its method contracts) can never break *BackupCell*.

To ensure that the implementation of a subclass D does not break the method contracts defined in a superclass C , we check that the contract of each method in C is satisfied by a method body that calls the method defined in D . More specifically, for each method m in C , we check that a method body that calls $D.m$ satisfies the contract of $C.m$, assuming that the dynamic type of the receiver is D . The latter proof obligation ensures that no existing code can be broken by the addition of a new subclass.

| | |
|--|---|
| <pre> class C { C() ensures valid(); { } void m(int x) requires valid() ∧ 0 ≤ x; { assert 0 ≤ x; } pure bool valid() reads ∅; { return true; } } </pre> <p style="text-align: center;">(a)</p> | <pre> class D extends C { D() ensures true; { super(); } void m(int x) requires x = 0; { assert x = 0; } pure bool valid() reads ∅; { return false; } } </pre> <p style="text-align: center;">(b)</p> |
|--|---|

Figure 20: A class C and its subclass D .

The rules for subclassing outlined above generalize Liskov's substitution

| program | time taken | source |
|------------------------|------------|--------------|
| Cell | 0.1 | [39, 13, 19] |
| Interval | 0.4 | |
| ArrayList and Iterator | 0.8 | [21, 37] |
| LinkedList | 43 | [47, 21] |
| Resource Pool | 2.1 | [13] |
| Marriage | 0.2 | [29] |
| MasterClock | 0.2 | [4] |
| Subject-Observer | 11 | [2, 38] |
| Recell, TCell, DCell | 0.5 | [40] |
| Visitor (framing only) | 127 | [13] |

Table 1: Table showing the time taken (in seconds) to verify each program.

principle [34]. For example, consider the classes C and D from Figure 20. D is a valid subclass of C , even though $D.m$'s precondition is stronger than the precondition of its overridden method, $C.m$. This kind of subclassing is safe, since it is not possible to establish the precondition of a dynamically bound call $o.m()$ where o 's static type is C and its dynamic type is D . Indeed, $o.valid()$ never holds for an object with dynamic type D . That is, one can never pass an object with dynamic type D to a method that expects a valid instance of C .

The extension for dealing with inheritance described above solves the extended state problem [25, 35] by allowing predicates and pure methods to be redefined in subclasses. For example, the predicate $BackupCell.valid$ overrides and redefines $Cell.valid$ to account for the additional field, $backup$. The meaning of $o.valid()$ is determined by the dynamic type of o . For instance, if o 's dynamic type is $Cell$, then one should interpret $o.valid()$ as defined in the class $Cell$. However, if the dynamic type of o is unknown, then precise meaning of $o.valid()$ cannot be determined. As consequence, $o.valid()$ does not in general imply that $o.x$ can be updated, as redefinitions of $valid$ do not have to include the permission to access $o.x$. For example, the following code snippet does not verify.

```

void assignToX(Cell o)
  requires o ≠ null ∧ o.valid();
  { o.x := 1; }

```

Note that the assignment in the method $setX$ of Figure 19 does verify, because we assume **this**-calls in method contracts are statically bound when verifying method implementations. As argued above, this assumption is sound only if each virtual method is overridden and reverified in each subclass.

7 Experience

To demonstrate the approach described in this paper is amenable to automatic, static verification, we implemented it in a verifier prototype. The prototype

was used to verify several (variations of) programs used in related work. The time taken to verify each program and a reference to the paper(s) containing the program is shown in Table 1. The experiments were executed on a desktop machine with a Pentium Core Duo 2.66 GHz processor and 4 GB of memory running Windows Vista. To discharge the verification conditions, we used the Z3 [11] theorem prover. The verifier itself and the programs shown in Table 1 can be downloaded from <http://www.cs.kuleuven.be/~jans/vericool3>.

Iterated Star

The verification technique described in Section 4 did not include iterated star assertions. In our implementation, we handle this type of assertion as follows. An iterated star is translated to two universal quantifiers (i and j are fresh variables).

$$\begin{aligned} & (\forall x \bullet \text{Tr}(\text{min}) \leq x < \text{Tr}(\text{max}) \Rightarrow \text{Tr}(\phi)) \wedge \\ & (\forall i, j \bullet \text{Tr}(\text{min}) \leq i < \text{Tr}(\text{max}) \wedge \text{Tr}(\text{min}) \leq j < \text{Tr}(\text{max}) \wedge i \neq j \Rightarrow \\ & \quad \text{R}(\phi[i/x]) \cap \text{R}(\phi[j/x]) = \emptyset) \end{aligned}$$

The first quantification states that ϕ holds for all integers between min and max , while the second one states that the required access set is disjoint at different indices. An iterated star is well-defined only if the bounds are well-defined and the assertion is well-defined for all integers within those bounds. That is, the definedness of an iterated star is as follows.

$$\text{Df}(\text{min}) \wedge \text{Df}(\text{max}) \wedge (\forall x \bullet \text{Tr}(\text{min}) \leq x < \text{Tr}(\text{max}) \Rightarrow \text{Df}(\phi))$$

What is the required access set of an iterated star? Informally, the required access is the union of the required access sets of ϕ for all indices in the range: $\bigcup_{\text{Tr}(\text{min}) \leq x < \text{Tr}(\text{max})} \text{R}(\phi)$. However, \bigcup is not a first-order concept. Inspired by the encoding of comprehensions described by Monehan *et al.* [28], we encode the required access set of an iterated star as follows. For each iterated star in the program text, we generate a function in the verification logic union_i (where i is unique for each iterated star) with sort $\text{heap} \times \text{set} \times \text{int} \times \text{int} \times \text{ref}_1 \times \dots \times \text{ref}_k \rightarrow \text{set}$. This function represents the required access set of the corresponding iterated star. The first and second parameter are the heap and access set. The third and fourth parameter are the lower and upper bounds. The parameters $\text{ref}_1 \times \dots \times \text{ref}_k$ represent the free variables appearing in the body of the iterated star. Several axioms describe the behavior of union_i . For example, we add an axiom that states a set is disjoint from a union only if it is disjoint from all the elements.

$$\begin{aligned} & \forall h, a, \text{min}, \text{max}, x_1, \dots, x_k, s \bullet s \cap \text{union}_i(h, a, \text{min}, \text{max}, x_1, \dots, x_n) = \emptyset \Leftrightarrow \\ & (\forall x \bullet \text{Tr}(\text{min}) \leq x < \text{Tr}(\text{max}) \Rightarrow s \cap \text{R}(\phi) = \emptyset) \end{aligned}$$

Whenever two different iterated stars are sufficiently similar, we generate only one union function instead of two. Two iterated stars are sufficiently similar if

they differ only in the name of the quantified variable or in their range. Such similar iterated stars typically occur in loop invariants and postconditions.

Currently, our verifier prototype only supports quantifying via the iterated star described above, where the quantified variable is an integer that lies within a certain interval. However, an interesting research question is whether we can generalize iterated star to arbitrary quantification over all objects of a certain type. For example, one might write the following assertion to indicate all *Clock* objects with master *m* are up to date.

$$\forall \text{Clock } c \bullet c \neq \text{null} \wedge c.\text{getMaster}() = m \Rightarrow c.\text{inSync}()$$

At the time of writing however, it is not clear how one should reason about and frame such assertions.

Modularity

It is sound to use the implementation axioms of a library during verification of client code. For example, it is ok to rely on *getX*'s implementation axiom when verifying Figure 3(b). However, when the correctness proof of code relies on implementation axioms of methods implemented in other modules, that code's correctness becomes dependent on another module's internal details. For that reason, our verifier prototype makes a pure method's implementation axiom available only to other methods implemented in the same module. Indeed, to prove the correctness of Figure 3(b), it suffices to rely on *Cell*'s frame and footprint allocated axioms.

Triggers

An important consideration in the design of a verifier that relies on an SMT solver to discharge verification conditions is what triggers to use. A trigger consists of one or more terms. The SMT solver uses triggers to determine when and how it should instantiate universal quantifiers. Z3 requires a trigger to mention all quantified variables.

The implementation, frame and allocated axioms all contain a universal quantifier. The trigger for the implementation axiom is $C.p(h, a, \text{this}, x_1, \dots, x_n)$. The trigger for the allocated axiom is the term $(o, f) \in C.p(h, a, \text{this}, x_1, \dots, x_n)$. Finally, the trigger for the frame axiom consists of the terms $\text{succ}(h_{old}, a_{old}, h, a)$ and $C.p(h, a, \text{this}, x_1, \dots, x_n)$. In an earlier version of the tool, we instead used $ok(h_{old}, a_{old})$, $ok(h, a)$ and $C.p(h, a, \text{this}, x_1, \dots, x_n)$ as a trigger for the frame axiom. However, this choice was considerably slower as the theorem prover had to consider $n \times (n - 1)$ possible instantiations (any two well-formed states) instead of $n - 1$ (any two successor states), where n is the number of states appearing in the program text.

Flexible Pure Methods

It is crucial for soundness that the axioms generated for pure methods are consistent if the program is valid. For example, consider the pure method *bad* shown below.

```
pure int bad() { return bad() + 1; }
```

bad's implementation axiom is inconsistent: there is no interpretation for the function *bad* such that $bad() = bad() + 1$. To ensure verification is sound, we must somehow detect such “dangerous” pure methods. In this paper, we ensure consistency by enforcing a simple, but very restrictive rule: a method can only call pure methods defined earlier in the program text. This rule guarantees that pure methods terminate, which in turn implies that their implementation axioms are consistent. However, this rule is overly restrictive. For example, the program of Figure 9 would not be well-formed as the pure method *length* calls itself.

In our implementation, we instead use more flexible rules. More specifically, the body of a normal pure method p_2 can call any other pure method p_1 if either p_1 is defined before p_2 or if the size of the required access set of the callee's precondition is strictly smaller than the required access set of p_2 's precondition. In addition, a pure method can be annotated with a measure. The pure method can call itself if its measure decreases. A predicate can call any other predicate in positive positions (similar to [37]). Ensuring consistency of the encoding of pure methods is an active area of research [27, 42].

8 Related Work

The dynamic frames approach [2, 21, 1, 26, 43, 46] solves the frame problem by explicitly annotating methods with effect annotations. More specifically, the contract of a mutator consists of a modifies clause and a “swinging pivot postcondition”, while a pure method's contract includes a reads clause. The expressiveness of the dynamic frames approach stems from the fact that these effect annotations can mention arbitrary sets of memory locations. To support data abstraction, these location sets may be specified in terms of dynamic frames, i.e. pure methods or ghost fields that denote sets of locations. As an example, consider the dynamic frames version of the class *Cell* from Figure 3(a) (method *swap* not included) shown in Figure 21. *setX*'s contract includes a modifies clause indicating that all locations in the dynamic frame *footprint* can potentially be changed by the method. In addition, *setX*'s last postcondition encodes the swinging pivot property. The contract of each pure method includes a reads clause indicating that its return value only depends on locations in *footprint*(). All the latter effect annotations (indicated with the grey background) need to be provided by the developer, and must be checked explicitly by the verifier. In our approach on the other hand, none of the annotations in grey need to be provided or checked explicitly (they are free postconditions!). Instead, we only check at each field access that the corresponding location is accessible, which

```

class Cell {
  int x;

  Cell()
    modifies  $\emptyset$ ;
    ensures  $valid() \wedge getX() = 0$ ;
    ensures fresh( $footprint()$ );
  {  $x := 0$ ; }

  void setX(int v)
    requires  $valid()$ ;
    modifies  $footprint()$ ;
    ensures  $valid() \wedge getX() = v$ ;
    ensures fresh( $footprint()$ 
      \old( $footprint()$ ));
  {  $x := v$ ; }

  pure int getX()
    requires  $valid()$ ;
    reads  $footprint()$ ;
  { return  $x$ ; }

  pure bool valid()
    reads  $footprint()$ ;
  { return  $true$ ; }

  pure set  $footprint()$ 
    reads  $footprint()$ ;
  { return { (this,  $x$ ) }; }
}

```

Figure 21: The class *Cell* with traditional dynamic frames annotations.

allows us to deduce an upper bound on the set of readable and writable locations. Since access assertions can typically be piggy-backed onto invariants, as shown in the predicate *valid* of class *ArrayList* of Figure 6, contracts do not need to include additional effect annotations. Moreover, as callers typically already have to establish a callee’s invariant and the invariant is opaque to the caller, checking the access assertions inside the callee’s precondition incurs no additional cost.

Our approach was heavily inspired by separation logic [39, 40, 41]. In particular, the access assertion $\mathbf{acc}(e.f)$ is similar to separation logic’s points-to predicate $e.f \mapsto _$ and Parkinson and Bierman’s abstract predicates inspired our predicate methods. As an example, consider the class *Cell* annotated with separation logic specifications shown below.

The behavior of each of *Cell*’s methods is described in terms of the predicate *cell*. For example, the precondition of *setX* requires that **this** refers to a valid *Cell* object with an arbitrary value. The postcondition ensures that **this** still points to a valid *Cell* object and that it holds the value *v*. Separation logic specifications typically encode the validity, footprint and state of a data structure in a single predicate. Implicit dynamic frame specifications on the other hand specify the validity and footprint via predicates, but use pure methods to describe the state. Another difference between separation logic and implicit dynamic frames is that we allow using heap-dependent expressions, in particular field reads and pure method invocations, inside assertions. Separation logic extends classical logic with primitives for describing the structure of the heap

```

class Cell {
  int x;

  Cell()
    ensures cell(this, 0);
  { x := 0; }

  void setX(int v)
    requires cell(this, -);
    ensures cell(this, v);
  { x := v; }

  int getX()
    requires cell(this, ?v);
    ensures cell(this, v);
  { return x; }

  predicate cell(Cell c, int v)  $\equiv c.x \mapsto v$ 
}

```

Figure 22: The class *Cell* with separation logic annotations.

and with rules for reasoning about these new primitives. The basic primitive of separation logic is the separating conjunction. Semantically, a separating conjunction $P * Q$ holds if the heap can be divided into two disjoint parts such that P holds for one part and Q holds for the other. The implicit dynamic frames approach supports separating conjunction, but avoids the need to come up with a proper subdivision of the heap by computing required access sets and by checking that these access sets are disjoint. The most important rule in separation logic is the frame rule.

$$\frac{\{P\} \bar{s} \{Q\}}{\{P * R\} \bar{s} \{Q * R\}}$$

This rule states that a local specification, $\{P\} \bar{s} \{Q\}$, can be extended to a global one by adding additional, disjoint state R . This paper shows that separation logic's underlying idea, inferring frame information from preconditions, can be encoded in classical, first-order logic. An advantage of this approach is that standard theorem provers for first-order logic [11, 12] can be used to discharge proof obligations.

Several authors have proposed ways of using separation logic specifications in tools. SLICK [36] is a tool that transforms separation logic annotations for Java programs into runtime checks. The access set used in our verification conditions resembles the coloring of objects used in this tool. Berdine *et al.* [7] demonstrate that separation logic is amenable to automatic static verification by building a verifier, called smallfoot, for a small, procedural language. Smallfoot uses sym-

bolic execution instead of verification condition generation for checking that a program satisfies its specification. A symbolic heap in smallfoot consists of a number of pure formulae and a list of spatial formulae. The spatial formulae both describe which locations are accessible and what values they hold. The footprint of the spatial formulae corresponds to the set of accessible locations in implicit dynamic frames. To frame methods invocations, smallfoot performs frame inference. That is, when verifying a call, the chunks corresponding to the callee’s precondition are removed from the heap. The chunks that remain are called the frame. Afterwards, the chunks corresponding to the callee’s postcondition are added to this frame. In implicit dynamic frames, fresh variables for the heap and access set are introduced after each method invocation. Framing then relies on the free postconditions and on the frame axioms for pure methods. Distefano and Parkinson [13] extend the ideas used in smallfoot to a substantial subset of Java. In particular, they construct an automatic verifier for Java, called jStar. jStar is able to infer certain loop invariants provided developers input the necessary abstraction rules.

In the basic Boogie methodology [3], data abstraction is limited to object invariants. More specifically, each object has a ghost field *inv*, and the methodology ensures that the invariant of an object *o* holds whenever *o.inv* is true. To ensure the soundness of the approach, the Boogie methodology imposes several restrictions: *inv* can only be updated using special operations called **pack** and **unpack**, updating a field *o.f* requires *o.inv* to be false, and finally the invariant itself can only mention fields within the object’s ownership cone. The implicit dynamic frames approach can be considered to be conceptually simpler than the Boogie methodology (and its extensions), since it does not impose any methodological restrictions.

Leino and Müller [29] and Barnett and Naumann [4] extend the basic Boogie methodology to deal with non-hierarchical object structures. In particular, they allow invariants to mention fields outside of the object’s ownership cone provided certain visibility requirements are met. More specifically, if the invariant of class *C* mentions a field *f* of a non-owned object, then *C* must be visible in the the class declaring *f*. No such restriction is present in our approach.

Jacobs and Piessens [19] and Darvas and Leino [9] both extend the basic Boogie methodology with support for data abstraction using pure methods. Similarly to our approach, they model pure methods as functions in the verification logic. Both approaches essentially solve the framing problem by encoding in the verification logic that these functions depend only on a number of ownership cones instead of on the entire heap.

Leino and Müller [30] extend the basic Boogie methodology with model fields to achieve data abstraction. A model field declaration consists of a type, a name, and a constraint. A model field cannot be assigned to within the program text; instead the model field is assigned a random value satisfying the constraint whenever the object is being packed. To prevent unsound reasoning arising from unsatisfiable constraints, Leino and Müller require the theorem prover to come up with a witness before assuming the constraint holds. However, experience shows that theorem provers (in particular Simplify) are unable to find witnesses

even in simple cases, and as such it is unlikely that their approach is suitable for use within an automatic program verifier.

In [33], the authors propose using data groups to specify side-effects. To ensure soundness, their approach imposes two methodological restrictions: the pivot uniqueness and owner exclusion restriction. Our approach imposes no such restrictions, and as a consequence it can handle programs that [33] cannot. For example, the former restriction rules out sharing of representation objects, as is the case in the iterator pattern.

Müller’s thesis [35] combines model fields with an ownership type system called Universes. Model fields are similar to pure methods that have no parameters. Model fields may depend on the fields of owned objects and the fields of peer objects, i.e. objects with the same owner as the receiver. However, model fields can only depend on peers if a model field is visible within the peer. For example, if the pure method *hasNext* from Figure 6 were a model field, then *hasNext* would have to be visible to the class *ArrayList*. Our approach has no such restriction.

The use of pure methods in specifications has been discussed extensively in the literature [42, 10, 19, 5, 9]. In particular, encoding pure methods as functions in the logic is a standard technique in verification. To the best of our knowledge, this is the first approach that derives an upper bound on the set of readable locations from preconditions of pure methods. Some authors propose broadening the range of admissible pure methods by allowing certain side-effects. We believe our approach can be extended to support such weakly pure methods.

Verification of Java-programs with JML-like [24] annotations has received considerable attention in the research community [24, 6, 15]. To the best of our knowledge, all the JML tools rely on explicit effect annotations for framing. We believe those tools might benefit from our approach to reduce the number of effect annotations.

Zee *et al.* [47] focus on verification of linked data structures. Their technique for dealing with such data structures inspired our specification of linked list. In particular, they use a ghost field to represent the set of all nodes in a list and rely on quantification over that set in the invariant to appropriately constrain the values and next pointers of the list.

A preliminary version of this work was presented at the 2008 FTFJP workshop [44]. This preliminary version already sparked the interest of other authors [31, 32]. In particular, Leino and Müller combine implicit dynamic frames with fractional permissions and concurrency. However, they encode accessibility differently and do not show how to deal with data abstraction or inheritance in their encoding. Moreover, they provide no formal soundness proof.

9 Conclusion

In this paper, we improve upon the classical dynamic frames approach in two ways: (1) method contracts are more concise and (2) fewer proof obligations

must be discharged by the verifier. We have proven soundness, implemented the approach in a verifier prototype and demonstrated its expressiveness by verifying several challenging examples from related work.

Acknowledgements

Jan Smans is a research assistant of the Fund for Scientific Research – Flanders (FWO). Bart Jacobs is a postdoctoral fellow of the Fund for Scientific Research - Flanders (FWO). This research is partially funded by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy. We would like to thank Sophia Drossopoulou and Peter Müller for their useful comments and remarks.

References

- [1] Anindya Banerjee, Mike Barnett, and David A. Naumann. Boogie meets regions: a verification experience report. In *VSTTE*, 2008.
- [2] Anindya Banerjee, David A. Naumann, and Stan Rosenberg. Regional logic for local reasoning about global invariants. In *ECOOP*, 2008.
- [3] Mike Barnett, Robert DeLine, Manuel Fahndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6), 2004.
- [4] Mike Barnett and David A. Naumann. Friends need a bit more: Maintaining invariants over shared state. In *MPC*, 2004.
- [5] Mike Barnett, David A. Naumann, Wolfram Schulte, and Qi Sun. 99.44% pure: Useful abstractions in specifications. In *FTFJP*, 2004.
- [6] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer-Verlag, 2007.
- [7] Josh Berdine, Cristiano Calcagno, and Peter W. O’Hearn. Symbolic execution with separation logic. In *APLAS*, 2005.
- [8] G.M. Bierman, M.J. Parkinson, and A.M. Pitts. An imperative core calculus for Java and Java with effects. Technical Report 563, University of Cambridge Computer Laboratory, 2003.
- [9] m Darvas and K. Rustan M. Leino. Practical reasoning about invocations and implementations of pure methods. In *Fundamental Approaches to Software Engineering (FASE)*, 2007.
- [10] m Darvas and Peter Muller. Reasoning about method calls in interface specifications. *Journal of Object Technology*, 5(5), 2006.

- [11] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.
- [12] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: A theorem prover for program checking. *Journal of the ACM*, 52(3), 2005.
- [13] Dino Distefano and Matthew Parkinson. jStar: Towards practical verification for Java. In *OOPSLA*, 2008.
- [14] Sophia Drossopoulou and Susan Eisenbach. The Java type system is sound – probably. In *European Conference on Object-oriented Programming (ECOOP)*, 1997.
- [15] Cormac Flanagan, K. Rustan, M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *PLDI*, 2002.
- [16] Alexey Gotsman, Josh Berdine, Byron Cook, Noam Rinetzky, and Mooly Sagiv. Local reasoning for storable locks and threads. In *APLAS*, 2007.
- [17] Christian Haack and Clement Hurlin. Separation logic contracts for a Java-like language with fork/join. In *AMAST*, 2008.
- [18] Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *ACM SIGPLAN Conference on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, 1999.
- [19] Bart Jacobs and Frank Piessens. Inspector methods for state abstraction. *Journal of Object Technology*, 6(5), 2007.
- [20] Bart Jacobs and Frank Piessens. The VeriFast program verifier. Technical Report 520, Department of Computer Science, Katholieke Universiteit Leuven, 2008.
- [21] Yannis T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In *Formal Methods*, 2006.
- [22] Yannis T. Kassios. *A Theory of Object Oriented Refinement*. PhD thesis, University of Toronto, 2006.
- [23] Gary T. Leavens. JML’s rich, inherited specifications for behavioral subtypes. In *FMOODS*, 2006.
- [24] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. 1999.
- [25] K. Rustan M. Leino. Data groups: Specifying the modification of extended state. In *OOPSLA*, 1998.

- [26] K. Rustan M. Leino. Specification and verification of object-oriented software. *Marktoberdorf International Summer School – Lecture Notes*, 2008.
- [27] K. Rustan M. Leino and Ronald Middelkoop. Proving consistency of pure methods and model fields. In *FASE*, 2009.
- [28] K. Rustan M. Leino and Rosemary Monahan. Automatic verification of textbook programs that use comprehensions. In *FTFJP*, 2007.
- [29] K. Rustan M. Leino and Peter Müller. Object invariants in dynamic contexts. In *ECOOP*, 2004.
- [30] K. Rustan M. Leino and Peter Müller. A verification methodology for model fields. In *ESOP*, 2006.
- [31] K. Rustan M. Leino and Peter Müller. A basis for verifying multi-threaded programs. In *ESOP*, 2009.
- [32] K. Rustan M. Leino, Peter Müller, and Jan Smans. Verification of concurrent programs with Chalice. In A. Aldini and R. Gorrieri, editors, *Foundations of Security Analysis and Design V*, Lecture Notes in Computer Science. Springer-Verlag, 2009.
- [33] K. Rustan M. Leino, Arnd Poetzsch-Heffter, and Yunhong Zhou. Using data groups to specify and check side effects. In *PLDI*, 2002.
- [34] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. 16(6), 1994.
- [35] Peter Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.
- [36] Huu Hai Nguyen, Viktor Kuncak, and Wei-Ngan Chin. Runtime checking for separation logic. In *VMCAI*, 2008.
- [37] Matthew Parkinson. *Local Reasoning for Java*. PhD thesis, University of Cambridge, 2005.
- [38] Matthew Parkinson. Class invariants: The end of the road? In *IWACO*, 2007.
- [39] Matthew Parkinson and Gavin Bierman. Separation logic and abstraction. In *POPL*, 2005.
- [40] Matthew Parkinson and Gavin Bierman. Separation logic, abstraction and inheritance. In *POPL*, 2008.
- [41] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, 2002.

- [42] Arsenii Rudich, m Darvas, and Peter Muller. Checking well-formedness of pure-method specifications. In *Formal Methods (FM)*, 2008.
- [43] Bernd Schoeller. *Making Classes Provable Through Contracts, Models and Frames*. PhD thesis, ETH Zurich, 2007.
- [44] Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames. In *FTFJP*, 2008.
- [45] Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames. In *European Conference on Object-oriented Programming (ECOOP)*, 2009.
- [46] Jan Smans, Bart Jacobs, Frank Piessens, and Wolfram Schulte. An automatic verifier for Java-like programs based on dynamic frames. In *FASE*, 2008.
- [47] Karen Zee, Viktor Kuncak, and Martin Rinard. Full functional verification of linked data structures. In *ACM Conf. Programming Language Design and Implementation (PLDI)*, 2008.