Implicit Dynamic Frames

JAN SMANS, BART JACOBS, and FRANK PIESSENS, K.U. Leuven

2

An important, challenging problem in the verification of imperative programs with shared, mutable state is the frame problem in the presence of data abstraction. That is, one must be able to specify and verify upper bounds on the set of memory locations a method can read and write without exposing that method's implementation.

Separation logic is now widely considered the most promising solution to this problem. However, unlike conventional verification approaches, separation logic assertions cannot mention heap-dependent expressions from the host programming language, such as method calls familiar to many developers. Moreover, separation logic-based verifiers are often based on symbolic execution. These symbolic execution-based verifiers typically do not support non-separating conjunction, and some of them rely on the developer to explicitly fold and unfold predicate definitions. Furthermore, several researchers have wondered whether it is possible to use verification condition generation and standard first-order provers instead of symbolic execution to automatically verify conformance with a separation logic specification.

In this article, we propose a variant of separation logic called *implicit dynamic frames* that supports heap-dependent expressions inside assertions. Conformance with an implicit dynamic frames specification can be checked by proving the validity of a number of first-order verification conditions. To show that these verification conditions can be discharged automatically by standard first-order provers, we have implemented our approach in a verifier prototype and have used this prototype to verify several challenging examples from related work. Our prototype automatically folds and unfolds predicate definitions, as required, during the proof and can reason about non-separating conjunction which is used in the specifications of some of these examples. Finally, we prove the soundness of the approach.

Categories and Subject Descriptors: F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs

General Terms: Verification

Additional Key Words and Phrases: Program verification, frame problem, separation logic

ACM Reference Format:
Smans, J., Jacobs, B., and Piessens, F. 2012. Implicit dynamic frams. ACM Trans. Program. Lang. Syst. 34, 1, Article 2 (April 2012), 58 pages.
DOI = 10.1145/2160910.2160911 http://doi.acm.org/10.1145/2160910.2160911

1. INTRODUCTION

A module in an object-oriented program consists of a number of types. Each type provides a number of methods to create, modify, and query instances of that type. The behavior of a module can therefore be specified by describing the behavior of each of its methods.

A standard technique in verification is to define a method's behavior in terms of a method contract consisting of a pre- and postcondition. The precondition describes under which circumstances callers are allowed to call the corresponding method. The

Authors' addresses: J. Smans, B. Jacobs, and F. Piessens, Department of Computer Science, K.U. Leuven; corresponding author's; email; jan.smans@cs.kuleuven.be.

DOI 10.1145/2160910.2160911 http://doi.acm.org/10.1145/2160910.2160911

```
class Cell {

  Cell()
    requires true;
    ensures getX() = 0;

  int getX()
    requires true;
    ensures true;

  void setX(int v)
    requires true;
    ensures getX() = v;
}
```

Fig. 1. The class *Cell* annotated with method contracts (implementation omitted).

postcondition describes the effect of the method on the program state. As an example, consider the method *setX* of class *Cell* shown in Figure 1. *setX*'s precondition imposes no restrictions on callers, while its postcondition ensures that $getX()$ equals v when the method returns. Note that the specifications are highlighted by a grey background.

There are two ways to reason about the invocation of a method. Either a developer reasons about a call in terms of the callee's method contract or in terms of its implementation. However, the latter approach is problematic for several reasons. First of all, reasoning in terms of the implementation is cumbersome, as the caller has to take into account internal details of the callee. Indeed, the implementation of a class is often complex compared to its external contract. For example, Java's *HashTable* simply defines an updatable mapping from keys to values, but its implementation uses an array of buckets where each bucket is a linked list. Clearly, it is not efficient to reason about hashcodes, array indices, and buckets simply to look up the value for a given key. Second, reasoning in terms of the callee's implementation is not modular. That is, any change to the callee's implementation would force developers to revisit and recheck each caller. Finally, the callee's implementation may not even be available at the call site. For example, if the callee is an interface or the source code is not available for some other reason, then it is simply impossible to reason in terms of the implementation. For those reasons, verification tools typically reason about a call based on the callee's method contract instead of its implementation. As an example, consider the following code snippet.

```
Cell c := new Cell();
c.setX(5);
assert c.getX() = 5.
```

The method contracts of Figure 1 suffice to prove that the assert statement never fails. In particular, the truth of the assertion immediately follows from the postcondition of *setX*. Now, consider the following code snippet.

```
Cell c1 := new Cell();
c1.setX(5);
Cell c2 := new Cell();
c2.setX(10);
assert c1.getX() = 5;
```

Can we still prove that the assertion on the last line never fails? The answer is no. The contracts of the constructor and $setX$ are too weak to prove that this is the case. Indeed, $setX$'s contract allows the implementation to change the program state arbitrarily, as long as it ensures that $getX()$ returns the given value v. In particular, the contract does not prevent $c_2.setX(10)$ from modifying $c_1.getX()$. For example, the implementation shown in Figure 28 of Appendix A satisfies the contracts of Figure 1, but it would cause the assert statement just shown to fail.

To ensure that the assertion never fails, we must somehow strengthen $Cell$'s method contracts. More specifically, the constructor and $setX$ should additionally specify an upper bound on the set of memory locations they can modify. Similarly, $getX$ should declare an upper bound on the set of memory locations it can read. However, these additional specifications should not expose $Cell$'s implementation. This problem is called the *frame problem in the presence of data abstraction*.

In the last decade, several solutions to the frame problem in the presence of data abstraction have been proposed [O'Hearn et al. 2001; Kassios 2006; Banerjee et al. 2008; Leino et al. 2002] (see Section 7 for a detailed comparison). In particular, separation logic [O'Hearn et al. 2001] is now widely considered to be the most promising solution to this problem. However, unlike conventional verification approaches such as JML [Burdy et al. 2005], Eiffel [EIFFEL 2006], Code Contracts [Barnett et al. 2010] and Spec# [Barnett et al. 2004], where an assertion is more or less a boolean expression from the host programming language, separation logic assertions cannot mention heap-dependent expressions such as method calls. Moreover, separation logic-based verifiers are often based on symbolic execution [Berdine et al. 2005; Distefano and Parkinson 2008; Jacobs et al. 2010]. These symbolic execution-based verifiers typically do not support non-separating conjunction. In addition, several researchers have wondered whether it is possible to use verification condition generation and standard first-order provers instead of symbolic execution to automatically verify conformance with a separation logic specification. In summary, the contributions of this article are as follows.

—We propose a marriage between separation logic and conventional specification approaches called *implicit dynamic frames*. In particular, we propose a variant of separation logic that supports heap-dependent expressions inside assertions.
—We show how conformance with an implicit dynamic frames specification can be mechanically checked via verification condition generation and first-order theorem proving. Unlike several symbolic execution-based separation logic verifiers [Berdine et al. 2005; Distefano and Parkinson 2008; Jacobs et al. 2010], we can reason about non-separating conjunction. Moreover, contrary to VeriFast [Jacobs et al. 2010], predicates need not be folded and unfolded explicitly by the developer, but instead, this is done automatically by the theorem prover as required during the proof.
—We have implemented our approach in a verifier prototype and used this prototype to verify several challenging examples used in related work, such as the composite pattern. Our implementation shows that the generated verification conditions can be discharged automatically by standard, first-order provers.
—We prove the soundness of implicit dynamic frames.

This article extends our earlier work presented at the European Conference on Object-Oriented Programming (ECOOP) [Smans et al. 2009] with several additional examples, a soundness proof, and an updated comparison with related work.

The remainder of this article is structured as follows. In Section 2, we informally introduce implicit dynamic frames via a number of examples. We then formalize the verification approach (Section 3) and prove its soundness (Section 4). Section 5 shows a number of example specifications, including challenging programs from related work.

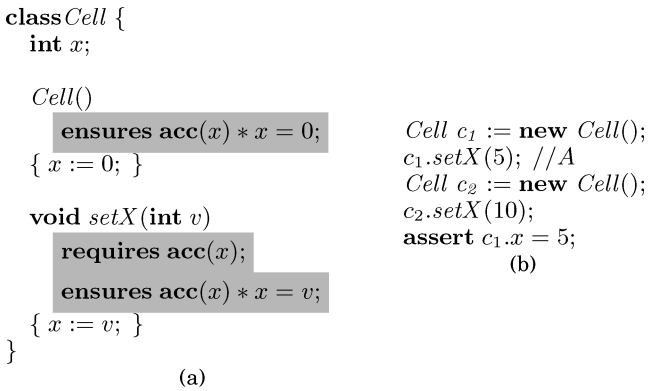

Fig. 2. The class *Cell* and some client code.

Finally, we discuss the experience with our verifier prototype, compare with related work, and conclude in Sections 6, 7, and 8.

2. IMPLICIT DYNAMIC FRAMES

This section informally explains how the implicit dynamic frames approach solves the frame problem (Section 2.1) and how it deals with data abstraction (Section 2.2) via a number of examples.

2.1. Framing

Just like separation logic, we rely on permissions to solve the frame problem. That is, an activation record can only access a memory location 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 $\mathbf{acc}(o.f)$. For example, the body of the method *setX* of Figure 2 can access the field x as the precondition requires x to be accessible.

The set of permissions held by an activation record can change over time. In particular, when a method a calls a method b, the permissions required to be accessible by b's precondition conceptually transfer from a to the callee b. Similarly, when a method call returns, the permissions described by that method's postcondition transfer to the caller. When a new object is created, the permissions to access the new object's fields transfer from the memory manager to the constructor (at the beginning of the constructor). For example, *Cell*'s constructor shown in Figure 2 has the permission to initialize x to zero, because at the beginning of the constructor's execution, this permission implicitly transfers from the system to that constructor.

Our verification approach enforces the program-wide invariant that for each allocated memory location $o.f$, at most one activation record holds the permission to access $o.f$. This invariant is the key to framing, as it allows us to infer an upper bound on the set of memory locations modifiable by a callee by looking at the permissions retained by the caller (after transferring the permissions required by the precondition to the callee). More specifically, the set of access permissions held by an activation record cannot change while that activation record is not on top of the call stack. Therefore, if an activation record retains the permission to access $o.f$ when performing a call, then that memory location cannot be modified by the callee, as it does not hold permission to do so.

Given the new method contracts for *Cell* of Figure 2(a) and the rules for framing just outlined, we can now prove that the assert statement in Figure 2(b) never fails. 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 demand

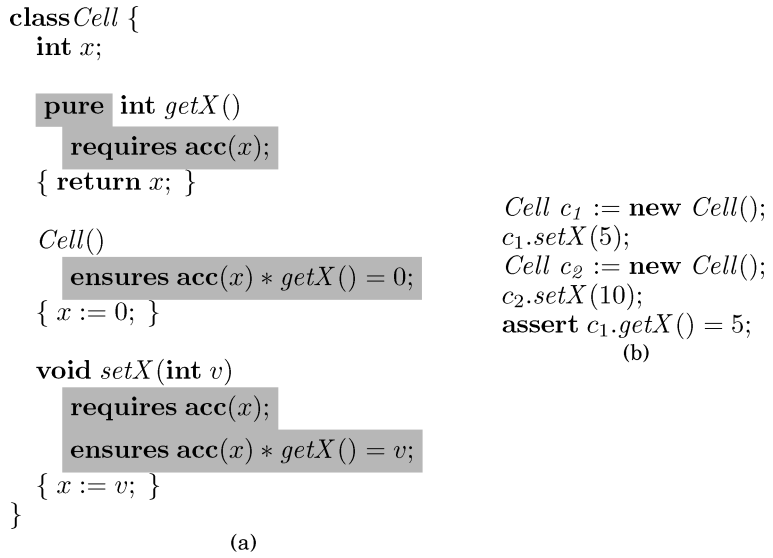

Fig. 3. The class *Cell* and some client code.

access to any existing memory location, the main procedure retains the permission to $c_1.x$ during execution of the constructor. It follows that c_2's constructor cannot modify $c_1.x$. Moreover, c_1 and c_2 cannot be equal, as constructors always return a fresh object reference.[1] Similarly, $c_2.setX(10)$ only requires the permission to access $c_2.x$, and therefore the main procedure again retains the permission to access c_1 during execution of that method. As neither c_2's constructor nor the subsequent call to $setX$ modify $c_1.x$, we can conclude that $c_1.x$ still holds the value 5, and hence that the assertion will succeed.

Note that the "bad" implementation of *Cell* shown in Figure 28 of Appendix A does not satisfy the contracts of Figure 2. In particular, $setX$ modifies $global.x$, but the method does not have permission to do so.

2.2. Data Abstraction

Data abstraction is crucial in the construction of modular programs, as it ensures that internal changes in one module do not propagate to other modules. However, the class *Cell* of Figure 2(a) was not written with data abstraction in mind. In particular, (1) client code must directly access the field x to query a *Cell* object's 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.

In this article, we tackle data abstraction by allowing certain methods to be mentioned inside specifications and by using predicates.

2.2.1. Pure Methods. Developers typically solve the first issue just mentioned by adding query methods or *getters* to their classes. For example, the class *Cell* of Figure 3(a) defines a method $getX$ to query a *Cell* object's state. The method is marked with the annotation **pure** to indicate it does not have side effects. As shown in Figure 3(b), the assert statement no longer has to mention internal fields but can, instead, be rephrased in terms of the query method.

[1]The inequality of c_1 and c_2 also follows from the fact that the set of permissions returned by a callee is disjoint from the set of permissions retained by the caller during that call.

To complete the decoupling between $Cell$'s implementation and client, we should also solve issue (2) and make $Cell$'s method contracts implementation-independent. In implicit dynamic frames, we (partially) solve issue (2) by allowing pure methods to be used inside specifications. That is, the effect of one method can 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 $getX$.

To prove the assertion at the end of Figure 3(b), we 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 prove that the set of memory locations modified by those statements is disjoint from the set of locations that $c_1.getX()$ depends on. But how can we determine which memory locations influence the return value of $getX$? The answer is simple.

We can deduce from the precondition of a pure method an upper bound on the set of memory locations readable by that method: a pure method p can only read a memory location $o.f$ if p's precondition requires $o.f$ to be accessible. In other words, the return value of a pure method only depends on memory locations required to be accessible by its precondition. For example, we can deduce from the precondition of $getX$ that the return value of $c_1.getX()$ depends on only $c_1.x$. As $c_1.x$ is neither modified by c_2's constructor nor by $c_2.setX(10)$, it follows that $c_1.getX()$ equals 5 at the end of the code snippet.

2.2.2. Predicates. The contracts of $Cell$ shown in Figure 3(a) are not implementation-independent yet, as they still expose the exact set of permissions required by each method. In particular, they mention $\mathbf{acc}(x)$. As a consequence, internal changes to $Cell$ (e.g., adding an additional field) cannot be made without affecting the method contracts. To make the method contracts implementation-independent, we must somehow abstract over the set of permissions required and ensured by each method.

In implicit dynamic frames, we abstract over permissions via predicates (similar to Parkinson and Bierman's abstract predicates [2005]). As an example, consider the predicate $valid$ of Figure 4(a). $c.valid()$ can be thought of as a shorthand for $\mathbf{acc}(c.x)$, just as $c.getX()$ is in some sense a different notation for $c.x$. As shown in Figure 4(a), the contracts of $Cell$ can be made implementation-independent by using the predicate instead of directly using the access permissions. Note that the body of $valid$ is only visible inside the module declaring the class $Cell$. Outside of that module, $valid$ is simply an opaque container of permissions.

We can deduce an upper bound on the set of memory locations readable by a pure method from its precondition. However, a predicate does not have a precondition, so what memory locations does the validity of a predicate depend on? That answer is that a predicate is *self-framing*. That is, the truth of a predicate depends only on memory locations that the predicate itself requires to be accessible.

Given the properties of pure methods and predicates outlined, we can now prove that the assert statement at the end of Figure 4(b) never fails. Informally, the reasoning is as follows. At program location *A*, the postcondition of $c_1.setX(5)$ holds: $c_1.valid()$ is true, and $c_1.getX()$ equals 5. As c_2's constructor does not demand access to any existing memory location, the main procedure retains the permissions required to be accessible by $c_1.valid()$ during execution of the constructor. Therefore, $c_1.valid()$ still holds after c_2's constructor, as none of the memory locations required to be accessible by $c_1.valid()$ can be modified and $valid$ is self-framing. Moreover, $c_1.getX()$ still equals 5, because all memory locations required to be accessible by its precondition have the same value before and after the constructor call. Finally, the set of permissions returned by the constructor's postcondition is disjoint from the set of locations retained by the main procedure. This property is sometimes called the swinging pivot restriction [Leino and Nelson 2002; Kassios 2006]. It immediately follows from this property that the set of memory locations required to be accessible by $c_1.valid()$ is disjoint from the set of

```
class Cell {
  int x;

  predicate valid() = acc(x);

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

  Cell()
    ensures valid() * getX() = 0;
  { x := 0; }

  void setX(int v)
    requires valid();
    ensures valid() * getX() = v;
  { x := v; }
}
```

(a)

```
Cell c1 := new Cell();
c1.setX(5); //A
Cell c2 := new Cell();
c2.setX(10);
assert c1.getX() = 5;
```

(b)

Fig. 4. The class *Cell* and some client code.

memory locations required to be accessible by $c_2.valid()$. $c2.setX(10)$ can only modify locations covered by $c_2.valid()$. This set of locations is disjoint from $c_1.valid()$, and hence both $c_1.valid()$ and $c_1.getX()$ are not affected by $c_2.setX(10)$. We may conclude that the assertion, $c_1.getX() = 5$, holds in every execution.

Note that the correctness proof just sketched does not depend on internal implementation details of the class *Cell* but only on the class' specification. As a consequence, changing *Cell*'s internal representation (within the boundaries set by its method contracts) does not endanger the correctness of the client code in Figure 4(b).

Our verification approach supports a variant of separation logic's separating conjunction (denoted as $P * Q$). To illustrate the use of separating conjunction, consider the method *swap*. The notation **old**(e) refers the value of the expression e in the method pre-state. *swap*'s precondition requires c_1 and c_2 to be separately valid, that is, both $c_1.valid()$ and $c_2.valid()$ hold, and 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 $c_2.valid()$. If we would have used a normal conjunction instead of a separating one, we would not be able to prove that the precondition of $c_2.setX(tmp)$ holds. In particular, the separating conjunction guarantees that the modifications to c_1 do not affect the validity of c_2.

```
void swap(Cell c1, Cell c2)
  requires c1 ≠ null ∧ c2 ≠ null;
  requires c1.valid() * c2.valid();
  ensures c1.valid() * c2.valid();
  ensures c1.getX() = old(c2.getX());
  ensures c2.getX() = old(c1.getX());
{
  int tmp := c1.getX();
  c1.setX(c2.getX());
  c2.setX(tmp); }
}
```

The specification of $Cell$ shown in Figure 4(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 by 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. Section 5 contains additional examples of implicit dynamic frames specifications.

3. VERIFICATION

In this section, we show how one can formally verify whether programs, such as the ones previously shown, satisfy their specification. More specifically, we start by defining a small imperative language (Section 3.1). We then show how to generate verification conditions for expressions, assertions, and statements in this language (Sections 3.2–3.5). Finally, we define the notion of *valid program* (Section 3.5.4). A program is valid if the verification conditions for each method and predicate are logically valid first-order formulas.

As we will explain later in this section, the key idea in the encoding of permissions is the explicit tracking of the set of accessible memory locations in the generated verification conditions. More specifically, the variable a in the verification conditions represents the set of accessible locations. Whenever the set of permissions changes (e.g., malloc or free), a is updated accordingly. Moreover, at each heap access, one must prove that a includes the location being read or written.

We show the soundness of the verification approach in Section 4. More specifically, we formally prove that executions of valid programs never fail.

3.1. Language

We define the following sets.

set	typical element	meaning
$\mathcal{Q}$	q	predicate names
$\mathcal{F}$	f	function names
$\mathcal{P}$	p	pure function names
$\mathcal{X}$	x, y	variable names

All of the aforementioned sets are disjoint. We describe our verification approach with respect to the imperative language of Figure 5. Overlining indicates repetition; annotations are highlighted in gray.

A program consists of zero or more modules. Each module declares functions, pure functions, and predicates. Both normal and pure functions have a corresponding function contract consisting of two assertions: a precondition and a postcondition. An assertion α is either an access assertion, a separating conjunction, a regular conjunction, an equality, a predicate, or a let assertion. The body of a mutator consists of a sequence of statements. A statement s is either an allocation, a heap update, a variable update, a function call, an assertion, or a deallocation. The body of a pure method returns an expression. An expression e is either a variable, a constant, a heap read, a pure function call, or a let expression.

In the remainder of this article, we consider only well-formed programs (see Definition 3.1). Note that well-formedness imposes the restriction that functions and predicates only mention functions and predicates defined earlier in the program text. This restriction is a simple, syntactic check that guarantees the consistency of the

$$
\begin{array}{lll}
\pi & ::= & \overline{module} \\
module & ::= & \textbf{module}\ \{\ \overline{decl}\ \} \\
decl & ::= & func \mid purefunc \mid predicate \\
predicate & ::= & \textbf{predicate}\ q(\overline{x})\ = \alpha; \\
func & ::= & \textbf{func}\ f(\overline{x})\ contract\ \{\ \overline{s}\ \} \\
purefunc & ::= & \textbf{pure}\ \textbf{func}\ p(\overline{x})\ contract\ \{\ \textbf{return}\ e;\ \} \\
contract & ::= & \textbf{requires}\ \alpha;\ \textbf{ensures}\ \alpha; \\
\alpha & ::= & \textbf{acc}(x) \mid \alpha * \alpha \mid \alpha \wedge \alpha \mid x = x \mid q(\overline{x}) \mid \textbf{let}\ x := e\ \textbf{in}\ \alpha \\
s & ::= & x := \textbf{cons}(\overline{x});\ \mid [x] := x;\ \mid x := e;\ \mid f(\overline{x}) \mid \textbf{assert}\ x = x;\ \mid \\
 & & \textbf{free}(x); \\
e & ::= & x \mid c \mid [x] \mid p(\overline{x}) \mid \textbf{let}\ x := e\ \textbf{in}\ e
\end{array}
$$

Fig. 5. A small imperative language. Annotations are highlighted in gray.

encoding of pure methods and predicates. We discuss this restriction together with more flexible solutions for ensuring consistency in Section 5.4.2.

Definition 3.1. A program is *well-formed* if all of the following hold.

—The program mentions only functions and predicates defined by the program.
—Predicate and function names are unique within the program. Parameter names are unique within a function or predicate. Parameters are not assigned within function bodies.
—The free variables of declaration bodies and contracts consist of the declaration parameters.
—The program contains a single function named $main$ with zero parameters and contract **requires** $0 = 0$; **ensures** $0 = 0$.
—The contract and body of functions and predicates mention only functions and predicates defined earlier in the program text.

Figure 29 in Appendix B shows how the program of Figure 4 can be encoded in the small, imperative language of Figure 5. In our examples, separating conjunction ($*$) has lower precedence than non-separating conjunction $\wedge$.

3.2. Logic

We target a multisorted, first-order logic with equality. That is, a term τ is either a variable or a function application. A formula ϕ is either $true$, $false$, a binary connective (and, or, implication, equivalence), a negation, a predicate, an equality, or a quantification.

$$
\begin{array}{lll}
\tau & ::= & x \mid f(\overline{\tau}); \\
\phi & ::= & true \mid false \mid \phi\ \mathsf{bop}\ \phi \mid \neg\phi \mid q(\overline{\tau}) \mid \tau = \tau \mid \forall \overline{x} \bullet \phi
\end{array}
$$

3.3. Signature

Each term has a corresponding sort. The sorts are int, the sort of integers, set, the sort of sets of integers, and $heap$, the sort of heaps. We do not explicitly mention sorts when they are clear from the context.

The signature of the logic consists of a number of built-in and program-specific functions and first-order predicates. The built-in functions are the following.

function	sort
select	$heap \to int \to int$
store	$heap \to int \to int \to heap$
emptyset	set
insert	$set \to int \to set$
union	$set \to set \to set$
intersect	$set \to set \to set$
zero	int
succ	$int \to int$
sum	$int \to int \to int$

The built-in predicates are the following.

predicate	sort
contains	$set \to int \to bool$
subset	$set \to set \to bool$
leq	$int \to int \to bool$

select and *store*, respectively, represent heap read and heap update. The functions *emptyset*, *insert*, *union*, and *intersect* and the predicates *contains* and *subset* correspond to the standard set operations. Finally, *zero*, *succ*, *sum*, and *leq* encode integers and the standard integer operations.

In the remainder of this article (in particular in the verification conditions), we will denote applications of built-in functions using the standard mathematical notation. For example, we denote $contains(s, v)$ as $v \in s$ and $sum(x, y)$ as $x + y$.

In addition to the built-in functions and predicates, the signature also contains a number of program-specific functions. More specifically, for each predicate,

$$\textbf{predicate } q(x_1, \ldots, x_n) = \alpha,$$

the signature contains a predicate q and function q_{FP} (FP is an abbreviation for footprint). The sort of the predicate q is

$$heap \to set \to int_1 \to \cdots \to int_n \to bool,$$

while the sort of the function q_{FP} is

$$heap \to set \to int_1 \to \cdots \to int_n \to set.$$

For example, the predicate *cell* of Figure 29 in Appendix B has a corresponding first-order predicate *cell* with sort $heap \to set \to int \to \mathit{bool}$ and a first-order function $cell_{FP}$ with sort $heap \to set \to int \to \mathit{set}$. For each pure function

$$\textbf{pure func } p(x_1, \ldots, x_n) \textbf{ requires } \alpha_1;\ \textbf{ensures } \alpha_2;\ \{\ \textbf{return } e;\ \},$$

the signature contains a function f with sort

$$heap \to set \to int_1 \to \cdots \to int_n \to int.$$

For example, the pure function *cell_get* of Figure 29 in Appendix B has a corresponding first-order function f with sort $heap \to set \to int \to \mathit{int}$. In Section 3.5.1, we will show that function calls (respectively, predicate instances) are translated into applications of the corresponding first-order function (respectively, predicate).

3.4. Theory

We check that the generated verification conditions are valid with respect to a particular theory. $\Sigma_{\mathsf{prelude}}$ is a theory which axiomatizes the built-in functions. For example,

the prelude contains the following axioms relating *select* and *store*:

$$\forall h, i, v, j \bullet i \neq j \Rightarrow select(store(h, i, v), j) = select(h, j),$$
$$\forall h, i, v \bullet select(store(h, i, v), i) = v.$$

Next to $\Sigma_{\mathsf{prelude}}$, each function and predicate symbol has a number of corresponding axioms. In these axioms, Tr and Df, respectively, denote translation and well-definedness of expressions and assertions, while Ras denotes the set of locations required to be accessible by an assertion. These functions will be defined in Section 3.5. Each pure function,

$$\textbf{pure func } p(x_1, \ldots, x_n) \textbf{ requires } \alpha_1;\ \textbf{ensures } \alpha_2;\ \{\ \textbf{return } e;\ \},$$

has three corresponding axioms.

—The function's *implementation axiom* states that applying the function corresponds to evaluating its body, provided the precondition holds.

$$\forall h, a, x_1, \ldots, x_n \bullet \mathsf{Tr}(\alpha_1) \Rightarrow p(h, a, x_1, \ldots, x_n) = \mathsf{Tr}(e).$$

—The *frame axiom* states that the function's return value depends only on memory locations required to be accessible by the precondition.

$$\forall h, h', a, a', x_1, \ldots, x_n \bullet$$
$$\mathsf{Tr}(\alpha_1) \wedge (\forall l \bullet l \in \mathsf{Ras}(\alpha_1) \Rightarrow l \in a' \wedge select(h, l) = select(h', l))$$
$$\Downarrow$$
$$p(h, a, x_1, \ldots, x_n) = p(h', a', x_1, \ldots, x_n).$$

—The *postcondition axiom* states that the function's result satisfies the postcondition, provided the precondition holds.

$$\forall h, a, x_1, \ldots, x_n \bullet \mathsf{Tr}(\alpha_1) \Rightarrow \mathsf{Tr}(\alpha_2)[p(h, a, x_1, \ldots, x_n)/result].$$

We call the theory consisting only of the implementation axiom the *internal theory*, denoted Σ_p^I. The function's *external theory* Σ_p^E consists of the frame and postcondition axioms. Σ_p^I can only be used by other functions in the module containing p, while Σ_p^E can only be used in client code. The subdivision in internal and external axioms ensures that the correctness of client code does not depend on internal implementation decisions of other modules.

Each predicate,

$$\textbf{predicate } q(x_1, \ldots, x_n) = \alpha,$$

has four axioms describing the predicate symbol q and the function symbol q_{FP}.

—The *implementation axiom* states that q holds in a given state whenever that state satisfies its body.

$$\forall h, a, x_1, \ldots, x_n \bullet q(h, a, x_1, \ldots, x_n) = \mathsf{Tr}(\alpha).$$

—The *footprint axiom* states that q_{FP} consists of the locations required to be accessible by the predicate body.

$$\forall h, a, x_1, \ldots, x_n \bullet q(h, a, x_1, \ldots, x_n) \Rightarrow q_{\mathsf{FP}}(h, a, x_1, \ldots, x_n) = \mathsf{Ras}(\alpha).$$

$$
\begin{array}{rcl}
\mathsf{Tr}(x) & \equiv & x \\
\mathsf{Tr}(c) & \equiv & c \\
\mathsf{Tr}([x]) & \equiv & \mathit{select}(h, x) \\
\mathsf{Tr}(p(y_1, \ldots, y_n)) & \equiv & p(h, a, y_1, \ldots, y_n) \\
\mathsf{Tr}(\mathbf{let}\ x := e_1\ \mathbf{in}\ e_2) & \equiv & \mathsf{Tr}(e_2)[\mathsf{Tr}(e_1)/x]
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{Df}(x) & \equiv & \mathit{true} \\
\mathsf{Df}(c) & \equiv & \mathit{true} \\
\mathsf{Df}([x]) & \equiv & x \in a \\
\mathsf{Df}(p(y_1, \ldots, y_n)) & \equiv & \mathsf{Tr}(P)[y_1/x_1, \ldots, y_n/x_n] \\
\mathsf{Df}(\mathbf{let}\ x := e_1\ \mathbf{in}\ e_2) & \equiv & \mathsf{Df}(e_1) \wedge \mathsf{Df}(e_2)[\mathsf{Tr}(e_1)/x]
\end{array}
$$

Fig. 6. Translation and definedness of expressions. In the definedness of pure function calls, $x_1, \ldots, x_n$ and P, respectively, denote p's parameters and precondition.

—The *frame axiom* states that q is self-framing, provided q holds. That is, if q holds, then changing memory locations outside of q_{FP} affects neither q nor q_{FP}.

$$
\begin{array}{c}
\forall h, h', a, a', x_1, \ldots, x_n \bullet \\
q(h, a, x_1, \ldots, x_n) \wedge \\
(\forall l \bullet l \in q_{\mathsf{FP}}(h, a, x_1, \ldots, x_n) \Rightarrow l \in a' \wedge \mathit{select}(h, l) = \mathit{select}(h', l)) \\
\Downarrow \\
q(h', a', x_1, \ldots, x_n) \wedge \\
q_{\mathsf{FP}}(h, a, x_1, \ldots, x_n) = q_{\mathsf{FP}}(h', a', x_1, \ldots, x_n).
\end{array}
$$

—The *footprint accessible axiom* states that q_{FP} contains only accessible locations, provided q holds.

$$\forall h, a, x_1, \ldots, x_n \bullet q(h, a, x_1, \ldots, x_n) \Rightarrow q_{\mathsf{FP}}(h, a, x_1, \ldots, x_n) \subseteq a.$$

We call the theory consisting only of the implementation and footprint implementation axioms the *internal theory*, denoted Σ_q^I. The predicate's *external theory* Σ_q^E consists of the frame and footprint allocated axioms. Just as for pure functions, the internal theory can only be used by other functions in the module containing q, while the external theory can only be used for verifying client code.

Each declaration d is verified with respect to a theory $\Sigma_{<d}$ (read as the theory consisting of all axioms that can be used during verification of d). $\Sigma_{<d}$ is the union of $\Sigma_{\mathsf{prelude}}$, the internal axioms of predicates and functions defined before d in the same module, and the external axioms of modules defined before d's module in the program text. Σ_π is the union of $\Sigma_{\mathsf{prelude}}$ and all axioms of all predicates and pure functions defined in π.

3.5. Verification Conditions

In this section, we define how expressions and assertions are encoded in the logic (Sections 3.5.1 and 3.5.2) and how verification conditions are generated for statements (Section 3.5.3).

3.5.1. Expressions. As shown in Figure 6, expressions are encoded in the logic as first-order terms via the function Tr. For example, a heap read $[x]$ is encoded as $\mathit{select}(h, x)$, and a pure function call is encoded as an application of the corresponding function symbol. Here (and in the remaining verification conditions) the variable h is of sort heap and represents the current value of the heap. Similarly, the variable a is of sort set and denotes the set of memory locations accessible to the current activation record.

Evaluation of an expression can fail (see Figure 10). For example, dereferencing a dangling pointer results in an error. Therefore, we not only define translation of expressions but also their well-definedness. More specifically, given an expression e,

$$
\begin{array}{rcl}
\mathsf{Tr}(\mathbf{acc}(x)) & \equiv & x \in a \\
\mathsf{Tr}(\alpha_1 * \alpha_2) & \equiv & \mathsf{Tr}(\alpha_1 \wedge \alpha_2) \wedge \mathsf{Ras}(\alpha_1) \cap \mathsf{Ras}(\alpha_2) = \emptyset \\
\mathsf{Tr}(\alpha_1 \wedge \alpha_2) & \equiv & \mathsf{Tr}(\alpha_1) \wedge \mathsf{Tr}(\alpha_2) \\
\mathsf{Tr}(x_1 = x_2) & \equiv & x_1 = x_2 \\
\mathsf{Tr}(q(x_1, \ldots, x_n)) & \equiv & q(h, a, x_1, \ldots, x_n) \\
\mathsf{Tr}(\mathbf{let}\ x := e\ \mathbf{in}\ \alpha) & \equiv & \mathsf{Tr}(\alpha)[\mathsf{Tr}(e)/x]
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{Df}(\mathbf{acc}(x)) & \equiv & \mathit{true} \\
\mathsf{Df}(\alpha_1 * \alpha_2) & \equiv & \mathsf{Df}(\alpha_1 \wedge \alpha_2) \\
\mathsf{Df}(\alpha_1 \wedge \alpha_2) & \equiv & \mathsf{Df}(\alpha_1) \wedge (\mathsf{Tr}(\alpha_1) \Rightarrow \mathsf{Df}(\alpha_2)) \\
\mathsf{Df}(x_1 = x_2) & \equiv & \mathit{true} \\
\mathsf{Df}(q(x_1, \ldots, x_n)) & \equiv & \mathit{true} \\
\mathsf{Df}(\mathbf{let}\ x := e\ \mathbf{in}\ \alpha) & \equiv & \mathsf{Df}(e) \wedge \mathsf{Df}(\alpha)[\mathsf{Tr}(e)/x]
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{Ras}(\mathbf{acc}(x)) & \equiv & \{\ x\ \} \\
\mathsf{Ras}(\alpha_1 * \alpha_2) & \equiv & \mathsf{Ras}(\alpha_1 \wedge \alpha_2) \\
\mathsf{Ras}(\alpha_1 \wedge \alpha_2) & \equiv & \mathsf{Ras}(\alpha_1) \cup \mathsf{Ras}(\alpha_2) \\
\mathsf{Ras}(x_1 = x_2) & \equiv & \emptyset \\
\mathsf{Ras}(q(x_1, \ldots, x_n)) & \equiv & q_{\mathsf{FP}}(h, a, x_1, \ldots, x_n) \\
\mathsf{Ras}(\mathbf{let}\ x := e\ \mathbf{in}\ \alpha) & \equiv & \mathsf{Ras}(\alpha)[\mathsf{Tr}(e)/x]
\end{array}
$$

Fig. 7. Translation, definedness, and required access set of assertions.

the function Df of Figure 6 computes a first-order formula whose validity implies that e does not fail. For example, dereferencing a pointer is well-defined if the address is accessible. Similarly, a pure function call is well-defined if the callee's precondition holds.

3.5.2. Assertions. As shown in Figure 7, assertions are encoded in the logic as first-order formulas via the function Tr. For example, an access assertion $\mathbf{acc}(x)$ holds if x is a member of the set of accessible memory locations a.

As assertions can contain expressions and as evaluation of an expression can fail, we define well-definedness of assertions. More specifically, given an assertion α, the function Df of Figure 7 computes a first-order formula whose validity implies that subexpressions of α do not fail. In particular, a let assertion $\mathbf{let}\ x := e\ \mathbf{in}\ \alpha$ is well-defined if both the subexpression e and the subassertion α, where the encoding of e is substituted for x, are well-defined.

Well-definedness of conjunction uses shortcut semantics. That is, the right-hand side must be well-defined only if the left-hand side holds. As a consequence, conjunction is not commutative with respect to well-definedness. For example, $\mathsf{Df}(\mathbf{acc}(x) \wedge [x] = 0)$ reduces to true, while $\mathsf{Df}([x] = 0 \wedge \mathbf{acc}(x))$ does not. Note however that non-commutativity of well-definedness for conjunction is not inherent to implicit dynamic frames. In particular, one can use the following definition for well-definedness of conjunction instead of the one shown in Figure 6.

$$\mathsf{Df}(\alpha_1 \wedge \alpha_2) \equiv (\mathsf{Df}(\alpha_1) \wedge (\mathsf{Tr}(\alpha_1) \Rightarrow \mathsf{Df}(\alpha_2))) \vee (\mathsf{Df}(\alpha_2) \wedge (\mathsf{Tr}(\alpha_2) \Rightarrow \mathsf{Df}(\alpha_1))).$$

Each assertion requires certain memory locations to be accessible. This set of memory locations is sometimes called the assertion's *footprint* [O'Hearn et al. 2001; Reynolds 2002]. As shown in Figure 7, the function Ras computes the footprint (a first-order term with sort set) for a given assertion. For instance, the required access set of $\mathbf{acc}(x)$ is the singleton set containing only x, while the required access set of a conjunction is the union of the footprint of the left-hand side and the right-hand side. The concept of footprint is crucial for defining the meaning of the separating conjunction. More

$$\mathsf{vc}(\mathsf{nil}, \phi) \equiv$$
$$\quad \phi$$

$$\mathsf{vc}(x := \mathbf{cons}(x_1, \ldots, x_n);\ \overline{s}, \phi) \equiv$$
$$\quad (\forall y \bullet$$
$$\qquad \{y, \ldots, y + n - 1\} \cap a = \emptyset$$
$$\qquad \Rightarrow$$
$$\qquad \mathsf{vc}(\overline{s}, \phi)[store(\ldots store(h, y, x_1) \ldots, y + n - 1, x_n)/h, a \cup \{\ y, \ldots, y + n - 1\ \}/a, y/x])$$

$$\mathsf{vc}([x_1] := x_2;\ \overline{s}, \phi) \equiv$$
$$\quad x_1 \in a \wedge \mathsf{vc}(\overline{s}, \phi)[store(h, x_1, x_2)/h]$$

$$\mathsf{vc}(x := e;\ \overline{s}, \phi) \equiv$$
$$\quad \mathsf{Df}(e) \wedge \mathsf{vc}(\overline{s}, \phi)[\mathsf{Tr}(e)/x]$$

$$\mathsf{vc}(f(y_1, \ldots, y_n); \overline{s}, \phi) \equiv$$
$$\quad P[y_1/x_1, \ldots, y_n/x_n] \wedge$$
$$\quad (\forall h', a' \bullet$$
$$\qquad \mathsf{Tr}(Q)[h'/h, a'/a, y_1/x_1, \ldots, y_n/x_n] \wedge$$
$$\qquad (\forall l \bullet l \in a - \mathsf{Ras}(P[y_1/x_1, \ldots, y_n/x_n]) \Rightarrow l \in a' \wedge select(h, l) = select(h', l)) \wedge\ //A$$
$$\qquad \mathsf{Ras}(Q)[h'/h, a'/a, y_1/x_1, \ldots, y_1/x_n] \cap a \subseteq \mathsf{Ras}(P)[y_1/x_1, \ldots, y_n/x_n]\ //B$$
$$\qquad \Rightarrow$$
$$\qquad \mathsf{vc}(\overline{s}, \phi)[h'/h, a'/a])$$

$$\mathsf{vc}(\mathbf{assert}\ x_1 = x_2;\ \overline{s}, \phi) \equiv$$
$$\quad x_1 = x_2 \wedge \mathsf{vc}(\overline{s}, \phi)$$

$$\mathsf{vc}(\mathbf{free}(x);\ \overline{s}, \phi) \equiv$$
$$\quad x \in a \wedge \mathsf{vc}(\overline{s}, \phi)[a - \{x\}/a]$$

Fig. 8. Verification conditions for statements. Variables bound in a quantifier are fresh. $x_1, \ldots, x_n$, P, and Q, respectively, in the verification condition for function calls, respectively, denote f's parameters, precondition, and postcondition.

specifically, a separating conjunction $\alpha_1 * \alpha_2$ holds if both α_1 and α_2 hold and their footprints are disjoint. In this article, we use the terms footprint and required access set interchangeably. Note that a separating conjunction $\alpha_1 * \alpha_2$ is equivalent to $\alpha_1 \wedge \alpha_2$ if the footprint of either α_1 or α_2 is the empty set. For example, $\mathbf{acc}(x) * [x] = 0$ is equivalent to $\mathbf{acc}(x) \wedge [x] = 0$.

3.5.3. Statements. The function vc defined in Figure 8 is a predicate transformer [Dijkstra 1975]. More specifically, given a sequence of statements $\overline{s}$ and a postcondition ϕ, $\mathsf{vc}(\overline{s}, \phi)$ denotes a precondition (a first-order formula) such that executions of $\overline{s}$ starting from a state satisfying this precondition either diverge or terminate in a state satisfying ϕ.

The definition of vc shown in Figure 8 is largely standard for imperative languages with aliasing. In particular, the heap is modeled as an updatable map from addresses to values [Bornat 2000]. The novel aspect of the verification conditions of Figure 8 is the tracking and use of access set a. For example, the verification condition for heap update checks that the access set a contains the updated memory location. Moreover, the verification condition for function calls relies on the access set for framing. More specifically, the conjunct marked A in the verification condition encodes the fact that when the caller retains access to a memory location (after transferring the permissions in the

precondition's footprint to the caller), then that memory location remains accessible and is not modified during the call. Furthermore, the conjunct marked B guarantees that the footprint of the postcondition is disjoint from the set of memory locations retained by the caller. The latter guarantee is crucial to prove disjointness of access sets and is sometimes called the swinging pivot property [Leino and Nelson 2002; Kassios 2006].

Note that the conjuncts marked A and B are free postconditions. That is, A and B can be assumed when verifying a call, but it is not necessary to prove them when verifying the implementation of the callee. In Section 4, we justify why A and B follow from the permission methodology.

3.5.4. Validity. A program is valid if the verification conditions of each declaration d in the program are provable in the theory $\Sigma_{<d}$ (Definition 3.5). More specifically, a predicate is valid (Definition 3.2) if its body is a well-defined assertion. A pure function is valid (Definition 3.3) if its contract and body are well-defined and its postcondition holds. In Section 4, we will prove that validity of a predicate or pure function implies the truth of its axioms. For example, the frame axiom of a pure method holds because its body is only well-defined if it reads memory locations in the precondition's footprint. Finally, a mutator is valid (Definition 3.4) if its contract is well-defined and the body satisfies the function contract.

Definition 3.2. A predicate,

$$\textbf{predicate}\ q(x_1, \ldots, x_n) = \alpha,$$

is valid if its body is a well-defined assertion such that

$$\Sigma_{<q} \vdash \forall h, a, x_1, \ldots, x_n \bullet \mathsf{Df}(\alpha).$$

Definition 3.3. A pure function,

$$\textbf{pure func}\ p(x_1, \ldots, x_n)\ \textbf{requires}\ \alpha_1;\ \textbf{ensures}\ \alpha_2;\ \{\ \textbf{return}\ e;\ \},$$

is valid if all of the following hold.

—The precondition is well-defined.

$$\Sigma_{<p} \vdash \forall h, a, x_1, \ldots, x_n \bullet \mathsf{Df}(\alpha_1).$$

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

$$\Sigma_{<p} \vdash \forall h, a, x_1, \ldots, x_n \bullet \mathsf{Tr}(\alpha_1) \Rightarrow \mathsf{Df}(\alpha_2).$$

—The function body is well-defined, provided the precondition holds.

$$\Sigma_{<p} \vdash \forall h, a, x_1, \ldots, x_n \bullet \mathsf{Tr}(\alpha_1) \Rightarrow \mathsf{Df}(e).$$

—The postcondition holds, given that the function returns its body and provided the precondition holds.

$$\Sigma_{<p} \vdash \forall h, a, x_1, \ldots, x_n \bullet \mathsf{Tr}(\alpha_1) \Rightarrow \mathsf{Tr}(\alpha_2)[\mathsf{Tr}(e)/\mathit{result}].$$

Definition 3.4. A function,

$$\textbf{func}\ f(x_1, \ldots, x_n)\ \textbf{requires}\ \alpha_1;\ \textbf{ensures}\ \alpha_2;\ \{\ \overline{s}\ \},$$

is valid if all of the following hold.

—The precondition is well-defined.

$$\Sigma_{<f} \vdash \forall h, a, x_1, \ldots, x_n \bullet \mathsf{Df}(\alpha_1).$$

$$H, A, \gamma, x \downarrow \gamma(x) \qquad H, A, \gamma, c \downarrow c \qquad \frac{\gamma(x) \in A}{H, A, \gamma, [x] \downarrow H(\gamma(x))}$$

$$\frac{\textbf{pure } p(x_1, \ldots, x_n) \ldots \{ \textbf{ return } e; \ \} \qquad H, A, [x_1 \mapsto \gamma(y_1), \ldots, x_n \mapsto \gamma(y_n)], e \downarrow c}{H, A, \gamma, p(y_1, \ldots, y_n) \downarrow c}$$

$$\frac{H, A, \gamma, e_1 \downarrow c_1 \qquad H, A, \gamma[x \mapsto c_1], e_2 \downarrow c}{H, A, \gamma, \textbf{let } x := e_1 \textbf{ in } e_2 \downarrow c}$$

Fig. 9. Evaluation of expressions.

—The postcondition is well-defined, provided the precondition held in the function pre-state.

$$\Sigma_{<f} \vdash \forall h, a, h', a', x_1, \ldots, x_n \bullet \mathsf{Tr}(\alpha_1) \Rightarrow \mathsf{Df}(\alpha_2)[h'/h, a'/a].$$

—The body satisfies the function contract.

$$\Sigma_{<f} \vdash \forall h, a, x_1, \ldots, x_n \bullet \mathsf{Tr}(\alpha_1) \Rightarrow \mathsf{vc}(\overline{s}, \mathsf{Tr}(\alpha_2)).$$

Definition 3.5. A program is valid if all declarations are valid.

In the next section, we prove that executions of valid programs never fail.

4. SOUNDNESS

In this section, we provide an execution semantics for the language defined in the previous section (Section 4.1), define an interpretation for the logic defined in the previous section (Section 4.2), and prove that executions of valid programs never fail (Section 4.3). This section can be skipped on first reading.

4.1. Execution

We first define execution of expressions using big step semantics and afterwards define execution of statements.

4.1.1. Evaluation of Expressions. Evaluation of expressions is defined using big step semantics. More specifically, the judgement $H, A, \gamma, e \downarrow r$ states that expression e evaluates to r in heap H with allocated locations A and environment γ. Here—and in the remainder of this article—a heap is a total function from integers to integers, a set of allocated locations is a subset of $\mathbb{N} - \{0\}$, and an environment is a total function from variables to values. The result r is either an integer value or **error**, indicating the expression performed an illegal operation.

$H, A, \gamma, e \downarrow r$ is inductively defined in Figures 9 and 10. The definition is mostly standard. Note that evaluation of an expression fails only if the expression or one of its subexpressions dereferences a dangling pointer.

4.1.2. Execution of Statements. Execution of statements is defined using small step semantics. More specifically, the judgement $\sigma \rightarrow \sigma'$ denotes that program state σ can execute in one step to σ'. As shown in Definitions 4.1 and 4.2, a program state is either an **error** indicating the program performed an illegal operation or a triple consisting of a heap H, an access set A, and a continuation κ. κ represents the stack. A continuation is either **done** (an empty stack) or is an activation record consisting of an environment γ (mapping local variables to values), a sequence of statements, and a subsequent activation record.

Definition 4.1. A *continuation* κ is either:

$$\frac{\gamma(x) \notin A}{H, A, \gamma, [x] \downarrow \textbf{error}}$$

$$\frac{\textbf{pure}\ p(x_1, \ldots, x_n) \ldots \{\ \textbf{return}\ e;\ \} \qquad H, A, [x_1 \mapsto \gamma(y_1), \ldots, x_n \mapsto \gamma(y_n)], e \downarrow \textbf{error}}{H, A, \gamma, p(y_1, \ldots, y_n) \downarrow \textbf{error}}$$

$$\frac{H, A, \gamma, e_1 \downarrow \textbf{error}}{H, A, \gamma, \textbf{let}\ x := e_1\ \textbf{in}\ e_2 \downarrow \textbf{error}} \qquad \frac{H, A, \gamma, e_1 \downarrow c_1 \qquad H, A, \gamma[x \mapsto c_1], e_2 \downarrow \textbf{error}}{H, A, \gamma, \textbf{let}\ x := e_1\ \textbf{in}\ e_2 \downarrow \textbf{error}}$$

Fig. 10. Evaluation of expressions (error transitions).

$$\frac{\{l, \ldots, l + n - 1\} \cap A = \emptyset}{\begin{array}{c}(H, A, \textbf{exec}(\gamma, x := \textbf{cons}(x_1, \ldots, x_n);\ \overline{s}, \kappa)) \rightarrow \\ (H[l \mapsto \gamma(x_1), \ldots, l + n - 1 \mapsto \gamma(x_n)], A \cup \{l, \ldots, l + n - 1\}, \textbf{exec}(\gamma[x \mapsto l_1], \overline{s}, \kappa))\end{array}}$$

$$\frac{\gamma(x_1) \in A}{(H, A, \textbf{exec}(\gamma, [x_1] := x_2;\ \overline{s}, \kappa)) \rightarrow (H[\gamma(x_1) \mapsto \gamma(x_2)], A, \textbf{exec}(\gamma, \overline{s}, \kappa))}$$

$$\frac{H, A, \gamma, e \downarrow c}{(H, A, \textbf{exec}(\gamma, x := e;\ \overline{s}, \kappa)) \rightarrow (H, A, \textbf{exec}(\gamma[x \mapsto c], \overline{s}, \kappa))}$$

$$\frac{\textbf{void}\ f(x_1, \ldots, x_n) \ldots \{\overline{r}\}}{\begin{array}{c}(H, A, \textbf{exec}(\gamma, f(y_1, \ldots, y_n);\ \overline{s}, \kappa)) \rightarrow \\ (H, A, \textbf{exec}([x_1 \mapsto \gamma(y_1), \ldots, x_n \mapsto \gamma(y_n)], \overline{r}, \textbf{exec}(\gamma, \overline{s}, \kappa)))\end{array}}$$

$$\frac{\gamma(x_1) = \gamma(x_2)}{(H, A, \textbf{exec}(\gamma, \textbf{assert}\ x_1 = x_2;\ \overline{s}, \kappa)) \rightarrow (H, A, \textbf{exec}(\gamma, \overline{s}, \kappa))}$$

$$\frac{\gamma(x) \in A}{(H, A, \textbf{exec}(\gamma, \textbf{free}(x);\ \overline{s}, \kappa)) \rightarrow (H, A - \{\gamma(x)\}, \textbf{exec}(\gamma, \overline{s}, \kappa))}$$

$$(H, A, \textbf{exec}(\gamma, \mathsf{nil}, \kappa)) \rightarrow (H, A, \kappa)$$

Fig. 11. Execution of statements.

—**done**, indicating the program finished executing.
—**exec**$(\gamma, \overline{s}, \kappa)$, where γ is an environment, $\overline{s}$ is a statement list, and κ is a continuation. An environment is a total function from variables names to integers.

Definition 4.2. A *program state* σ is either:

—The error state, **error**. The error state indicates the program has performed an illegal operation, such as accessing a nonallocated memory location or an assertion failure.
—A triple (H, A, κ) consisting of:
  —A heap H, a total function from integers to integers;
  —An access set A, a subset of $\mathbb{N} - \{0\}$;
  —A continuation κ.

The relation $\rightarrow$ is defined in Figures 11 and 12. Again, the definition is mostly standard. Note that a program execution can end up in the **error** state because of four

$$\frac{\gamma(x_1) \notin A}{(H, A, \mathbf{exec}(\gamma, [x_1] := x_2;\ \overline{s}, \kappa)) \to \mathbf{error}} \qquad \frac{H, A, \gamma, e \downarrow \mathbf{error}}{(H, A, \mathbf{exec}(\gamma, x := e;\ \overline{s}, \kappa)) \to \mathbf{error}}$$

$$\frac{\gamma(x_1) \neq \gamma(x_2)}{(H, A, \mathbf{exec}(\gamma, \mathbf{assert}\ x_1 = x_2;\ \overline{s}, \kappa)) \to \mathbf{error}} \qquad \frac{\gamma(x) \notin A}{(H, A, \mathbf{exec}(\gamma, \mathbf{free}(x);\ \overline{s}, \kappa)) \to \mathbf{error}}$$

Fig. 12. Execution of statements (error transitions).

reasons: (1) writing a dangling pointer, (2) reading a dangling pointer (in a subexpression), (3) an assertion violation, or (4) freeing a dangling pointer.

The execution of a program starts in the initial state (Definition 4.3). We say a program execution goes wrong if the **error** state is reached (Definition 4.4) from the initial state. In the next section, we prove that executions of valid programs do not go wrong.

Definition 4.3. The *initial state* σ_{ini} of a program is

$$([\_ \mapsto 0], \emptyset, (\mathbf{exec}([\_ \mapsto 0], \overline{s}, \mathbf{done}))),$$

where $\overline{s}$ is the body of the program's main function. Initially, the heap maps each address to zero and the set of allocated addresses is empty.

Definition 4.4. $\sigma \to^* \sigma'$ if σ equals σ' or if there exists a state σ'' such that $\sigma \to \sigma''$ and $\sigma'' \to^* \sigma'$.

4.2. Interpretation

We interpret the logic using the interpretation $\mathcal{J}$. $\mathcal{J}$ maps each sort to a set.

sort	set
int	$\mathbb{Z}$
set	$\mathcal{P}(\mathbb{Z})$
heap	$\mathbb{Z} \to \mathbb{Z}$

$\mathcal{J}$ maps each function (respectively, predicate) symbol in the signature to a function (respectively, predicate). We interpret the built-in predicates and functions as the standard mathematical operations.

function symbol	interpretation
$select(h, l)$	$h(l)$
$store(h, l, v)$	$h[l \mapsto v]$
emptyset	$\emptyset$
$insert(s, l)$	$s \cup \{l\}$
$union(s_1, s_2)$	$s_1 \cup s_2$
$intersect(s_1, s_2)$	$s_1 \cap s_2$
zero	0
$succ(x)$	$x + 1$
$sum(x, y)$	$x + y$

predicate symbol	interpretation
$contains(s, l)$	$l \in s$
$subset(s_1, s_2)$	$s_1 \subseteq s_2$
$leq(x, y)$	$x \leq y$

We assume $\mathcal{J}$ is a model for $\Sigma_{\mathsf{prelude}}$. That is, $\mathcal{J} \models \Sigma_{\mathsf{prelude}}$. Before we define how $\mathcal{J}$ interprets the built-in functions, we define the interpretation of expressions and

assertions. We denote the interpretation of an expression e under variable assignment γ as $[\![e]\!]_\gamma$. We define expression interpretation as follows.

$$[\![e]\!]_\gamma = \begin{cases} c & \text{if } \gamma(h), \gamma(a), \gamma, e \downarrow c \\ 0 & \text{if } \gamma(h), \gamma(a), \gamma, e \downarrow \textbf{error} \end{cases}.$$

Note that in the preceding definition, we implicitly convert variable assignment γ to an environment by removing all pairs (x, v) where v is not an integer from γ. We use the same conversion in the remainder of this article. We define the required access set of an assertion α under variable assignment γ as $[\![\alpha]\!]^R_\gamma$. We define the required access set as follows.

$$\begin{aligned}
[\![\textbf{acc}(x)]\!]^R_\gamma &::= \{\gamma(x)\} \\
[\![\alpha_1 * \alpha_2]\!]^R_\gamma &::= [\![\alpha_1 \wedge \alpha_2]\!]^R_\gamma \\
[\![\alpha_1 \wedge \alpha_2]\!]^R_\gamma &::= [\![\alpha_1]\!]^R_\gamma \cup [\![\alpha_2]\!]^R_\gamma \\
[\![x_1 = x_2]\!]^R_\gamma &::= \emptyset \\
[\![q(y_1, \ldots, y_n)]\!]^R_\gamma &::= [\![\alpha]\!]^R_{\gamma'} \\
&\quad \text{where } \alpha \text{ is } q\text{'s body,} \\
&\quad x_1, \ldots, x_n \text{ are } q\text{'s parameters,} \\
&\quad \gamma' = [h \mapsto \gamma(h), a \mapsto \gamma(a), x_1 \mapsto \gamma(y_1), \ldots, x_n \mapsto \gamma(y_n)] \\
[\![\textbf{let } x := e \textbf{ in } \alpha]\!]^R_\gamma &::= [\![\alpha]\!]^R_{\gamma[x \mapsto [\![e]\!]_\gamma]}.
\end{aligned}$$

Finally, we denote the truth of an assertion α under variable assignment γ as $[\![\alpha]\!]_\gamma$. We define truth of an assertion as follows.

$$\begin{aligned}
[\![\textbf{acc}(x)]\!]_\gamma &::= \gamma(x) \in \gamma(a) \\
[\![\alpha_1 * \alpha_2]\!]_\gamma &::= [\![\alpha_1 \wedge \alpha_2]\!]_\gamma \text{ and } [\![\alpha_1]\!]^R_\gamma \cap [\![\alpha_2]\!]^R_\gamma = \emptyset \\
[\![\alpha_1 \wedge \alpha_2]\!]_\gamma &::= [\![\alpha_1]\!]_\gamma \text{ and } [\![\alpha_2]\!]_\gamma \\
[\![x_1 = x_2]\!]_\gamma &::= \gamma(x_1) = \gamma(x_2) \\
[\![q(y_1, \ldots, y_n)]\!]_\gamma &::= [\![\alpha]\!]_{\gamma'} \\
&\quad \text{where } \alpha \text{ is } q\text{'s body,} \\
&\quad x_1, \ldots, x_n \text{ are } q\text{'s parameters, and} \\
&\quad \gamma' = [h \mapsto \gamma(h), a \mapsto \gamma(a), x_1 \mapsto \gamma(y_1), \ldots, x_n \mapsto \gamma(y_n)] \\
[\![\textbf{let } x := e \textbf{ in } \alpha]\!]_\gamma &::= [\![\alpha]\!]_{\gamma[x \mapsto [\![e]\!]_\gamma]}.
\end{aligned}$$

We can now define the interpretation of the program-specific functions. That is, each function,

$$\textbf{pure func } p(x_1, \ldots, x_n) \textbf{ requires } \alpha_1; \textbf{ ensures } \alpha_2; \{ \textbf{return } e; \},$$

has a corresponding function symbol p. We interpret $\mathcal{J}(p)(H, A, c_1, \ldots, c_n)$ as $[\![p(x_1, \ldots, x_n)]\!]_\gamma$, where γ is $[h \mapsto H, a \mapsto A, x_1 \mapsto c_1, \ldots, x_n \mapsto c_n]$. For example, $\mathcal{J}(cell\_get)$ is a function with three arguments: a heap, an access set, and an integer. $\mathcal{J}(cell\_get)(H, A, c)$ equals $H(c)$ if $c \in A$; otherwise, the result equals 0.

Each predicate,

$$\textbf{predicate } q(x_1, \ldots, x_n) = \alpha,$$

has a corresponding predicate symbol q and function symbol q_{FP}. We interpret $\mathcal{J}(q)(H, A, c_1, \ldots, c_n)$ as $[\![q(x_1, \ldots, x_n)]\!]_\gamma$ and $\mathcal{J}(q_{\mathsf{FP}})(H, A, c_1, \ldots, c_n)$ as $[\![q(x_1, \ldots, x_n)]\!]^R_\gamma$, where γ is $[h \mapsto H, a \mapsto A, x_1 \mapsto c_1, \ldots, x_n \mapsto c_n]$. For example, the predicate $\mathcal{J}(cell)$ is a predicate with three arguments: a heap, an access set, and an integer. $\mathcal{J}(cell\_get)(H, A, c)$ holds only if $c \in A$.

Properties of $\mathcal{J}$. We now prove a number of properties about $\mathcal{J}$ that are crucial in order to establish soundness. First of all, we prove that the encoding of expressions

and assertions in verification conditions via Tr, Df, and Ras is consistent with the interpretation $\mathcal{J}$ (Theorem 4.6). Second, we demonstrate that $\mathcal{J}$ is a model for the axioms generated for predicates and pure methods (Theorem 4.11).

Definition 4.5. An expression e_1 or assertion α_1 is smaller than an expression e_2 or assertion α_2 if the largest pure function or predicate appearing in the former is smaller than the largest pure function appearing in the latter or if the former is syntactically smaller than the latter (assuming the first criterion does not distinguish them). A declaration d_1 is smaller than a declaration d_2 if d_1 appears before d_2 in the program text.

THEOREM 4.6. *Suppose π is a valid program.*

—*If e is an expression over π; γ is a variable assignment, $\mathcal{J} \models \Sigma_{<d}$ where d is the largest predicate or pure function appearing in e; and $\mathcal{J}, \gamma \models \mathsf{Df}(e)$, then $\gamma(h), \gamma(a), \gamma, e \downarrow c$ (with $c \neq$ **error**) and $\mathcal{J}, \gamma \models \mathsf{Tr}(e) = c$.*

—*If α is an assertion over π, γ is a variable assignment, $\mathcal{J} \models \Sigma_{<d}$, where d is the largest predicate or pure function appearing in α, $\mathcal{J}, \gamma \models \mathsf{Df}(\alpha)$, then $\mathcal{J}, \gamma \models \mathsf{Tr}(\alpha)$ if and only if $[\![\alpha]\!]_\gamma$. Moreover, if $\mathcal{J}, \gamma \models \mathsf{Tr}(\alpha)$, then $\mathcal{J}, \gamma \models \mathsf{Ras}(\alpha) = [\![\alpha]\!]^R_\gamma$.*

PROOF. By induction on the size of the assertion or expression. We first consider expressions, and afterwards we consider assertions. We highlight only the most interesting cases.

Suppose e is an expression over π; γ is a variable assignment, $\mathcal{J} \models \Sigma_{<d}$ where d is the largest predicate or pure function appearing in e; and $\mathcal{J}, \gamma \models \mathsf{Df}(e)$. Suppose $\gamma(h)$ equals H and $\gamma(a)$ equals A.

—Suppose the expression is $[x]$. It follows from the well-definedness of $[x]$ that $\mathcal{J}, \gamma \models x \in a$, and hence that $\gamma(x) \in A$. Therefore, evaluation of $[x]$ does not go wrong. It follows from the evaluation of expressions in Figure 9 that $[\![[x]]\!]_\gamma$ equals $H(\gamma(x))$. Similarly, it follows from the definition of Tr in Figure 6 that $\mathsf{Tr}([x])$ equals $select(H, \gamma(x))$. It immediately follows from the definition of $\mathcal{J}(select)$ that the theorem holds.

—Suppose the expression is $p(y_1, \ldots, y_n)$, where p is defined as follows.

$$\textbf{pure func } p(x_1, \ldots, x_n) \textbf{ requires } \alpha_1; \textbf{ ensures } \alpha_2; \{ \textbf{ return } e; \}.$$

It follows from the well-definedness of $p(x_1, \ldots, x_n)$ that $\mathcal{J}, \gamma' \models \mathsf{Tr}(\alpha_1)$, where $\gamma' = [h \mapsto H, a \mapsto A, x_1 \mapsto \gamma(y_1), \ldots, x_1 \mapsto \gamma(y_n)]$. π is a valid program, and hence p is a valid pure function. For that reason, the following holds.

$$\Sigma_{<p} \vdash \forall h, a, x_1, \ldots, x_n \bullet \mathsf{Tr}(\alpha_1) \Rightarrow \mathsf{Df}(e).$$

p is the largest pure method in $p(y_1, \ldots, y_n)$, and thus $\mathcal{J} \models \Sigma_{<p}$. Therefore, the following holds.

$$\mathcal{J} \models \forall h, a, x_1, \ldots, x_n \bullet \mathsf{Tr}(\alpha_1) \Rightarrow \mathsf{Df}(e).$$

It follows by instantiating the quantifier that

$$\mathcal{J}, \gamma' \models \mathsf{Tr}(\alpha_1) \Rightarrow \mathsf{Df}(e).$$

Since $\mathcal{J}, \gamma' \models \mathsf{Tr}(\alpha_1)$, it follows that

$$\mathcal{J}, \gamma' \models \mathsf{Df}(e).$$

We can now apply the induction hypothesis to e, as e is smaller than $p(y_1, \ldots, y_n)$. That is, evaluation of e does not go wrong and results in a value c. It follows from the definition of $\downarrow$ that $p(y_1, \ldots, y_n)$ does not go wrong and evaluates to c. It immediately follows from the definition of $\mathcal{J}(p)$ that the theorem holds.

Suppose α is an assertion over π, γ is a variable assignment, $\mathcal{J} \models \Sigma_{<d}$ where d is the largest predicate or pure function appearing in α, and $\mathcal{J}, \gamma \models \mathsf{Df}(\alpha)$. Suppose $\gamma(h)$ equals H and $\gamma(a)$ equals A.

—Suppose the assertion is $\mathbf{acc}(x)$. It follows from the definitions of assertion interpretation and Tr that

$$\mathcal{J}, \gamma \models \mathsf{Tr}(\mathbf{acc}(x)) \Leftrightarrow \gamma(x) \in A \Leftrightarrow [\![\mathbf{acc}(x)]\!]_\gamma.$$

Similarly, it follows from the definition of required access set and Ras that

$$\mathcal{J}, \gamma \models \mathsf{Ras}(\mathbf{acc}(x)) = \{\gamma(x)\} = [\![\mathbf{acc}(x)]\!]^R_\gamma.$$

—Suppose the assertion is $\alpha_1 * \alpha_2$. As the assertion is well-defined, it follows that

$$\mathcal{J}, \gamma \models \mathsf{Df}(\alpha_1) \wedge (\mathsf{Tr}(\alpha_1) \Rightarrow \mathsf{Df}(\alpha_2)).$$

We can apply the induction hypothesis to α_1, as $\mathcal{J}, \gamma \models \mathsf{Df}(\alpha_1)$ and α_1 is syntactically smaller than the assertion itself. Therefore (1),

$$\mathcal{J}, \gamma \models \mathsf{Tr}(\alpha_1) \Leftrightarrow [\![\alpha_1]\!]_\gamma,$$

and if α_1 holds then

$$\mathcal{J}, \gamma \models \mathsf{Ras}(\alpha_1) = [\![\alpha_1]\!]^R_\gamma.$$

We need to distinguish two cases: $[\![\alpha_1]\!]_\gamma$ is either false or true. If $[\![\alpha_1]\!]_\gamma$ is false, then the theorem immediately follows. If $[\![\alpha_1]\!]_\gamma$ is true, then $\mathcal{J}, \gamma \models \mathsf{Tr}(\alpha_1)$, and hence $\mathcal{J}, \gamma \models \mathsf{Df}(\alpha_2)$. Thus, we can apply the induction hypothesis to α_2, α_2 is syntactically smaller than the assertion itself. Therefore (2)

$$\mathcal{J}, \gamma \models \mathsf{Tr}(\alpha_2) \Leftrightarrow [\![\alpha_2]\!]_\gamma,$$

and if α_2 holds then

$$\mathcal{J}, \gamma \models \mathsf{Ras}(\alpha_2) = [\![\alpha_2]\!]^R_\gamma.$$

The theorem then follows from (1) and (2). □

LEMMA 4.7. *Suppose H is a heap, A and B are access sets such that $A \subseteq B$, γ is an environment. Extending the access set does not change the interpretation of expressions and assertions.*

—*Suppose e is an expression. If $H, A, \gamma, e \downarrow c$ with $c \neq$ **error**, then $H, B, \gamma, e \downarrow c$.*
—*Suppose α is an assertion. If $[\![\alpha]\!]_{\gamma[h \mapsto H, a \mapsto A]}$, then $[\![\alpha]\!]_{\gamma[h \mapsto H, a \mapsto B]}$ and $[\![\alpha]\!]^R_{\gamma[h \mapsto H, a \mapsto A]} = [\![\alpha]\!]^R_{\gamma[h \mapsto H, a \mapsto B]}$.*

PROOF. By induction on the size of the assertion or expression. □

LEMMA 4.8. *Suppose H is a heap, A is an access set, γ is an environment, and α is an expression. If α holds, then α's footprint is a subset of A. That is,*

$$[\![\alpha]\!]_{\gamma[h \mapsto H, a \mapsto A]} \Rightarrow [\![\alpha]\!]^R_{\gamma[h \mapsto H, a \mapsto A]} \subseteq A.$$

PROOF. By induction on the size of the assertion. □

Lemma 4.9 states that the truth of an assertion β that is well-defined in all contexts where α holds is not affected by modifications outside of the footprint of α and β. Moreover, the value of an expression e that is well-defined in all contexts where α holds is framed by α's footprint.

LEMMA 4.9. *Suppose π is a valid program, α is an assertion, the variables of α are a subset of $\{x_1, \ldots, x_n\}$, and the following hold.*

—*α is well-defined in all contexts.*

$$\mathcal{J} \models \forall h, a, x_1, \ldots, x_n \bullet \mathsf{Df}(\alpha).$$

—*α is self-framing.*

$$\begin{array}{c}\mathcal{J} \models \forall h, h', a, a', x_1, \ldots, x_n \bullet \\ \mathsf{Tr}(\alpha) \wedge (\forall l \in \mathsf{Ras}(\alpha) \bullet l \in a' \wedge \mathit{select}(h, l) = \mathit{select}(h', l)) \\ \Downarrow \\ \mathsf{Tr}(\alpha)[h'/h, a'/a].\end{array}$$

All of the following hold.

—*Suppose β is an assertion, the free variables of β are a subset of $\{x_1, \ldots, x_n\}$ and the following hold.*
  —*β is well-defined in all contexts where α holds.*

$$\mathcal{J} \models \forall h, a, x_1, \ldots, x_n \bullet \mathsf{Tr}(\alpha) \Rightarrow \mathsf{Df}(\beta).$$

  —*$\mathcal{J}$ is a model for $\Sigma_{<d}$ (where d is the largest predicate or pure function appearing in β).*

$$\mathcal{J} \models \Sigma_{<d}.$$

  Then, the following holds.

$$\begin{array}{c}\mathcal{J} \models \forall h, h', a, a', x_1, \ldots, x_n \bullet \\ \mathsf{Tr}(\alpha) \wedge \mathsf{Tr}(\beta) \wedge (\forall l \in \mathsf{Ras}(\alpha) \cup \mathsf{Ras}(\beta) \bullet l \in a' \wedge \mathit{select}(h, l) = \mathit{select}(h', l)) \\ \Downarrow \\ \mathsf{Tr}(\beta)[h'/h, a'/a] \wedge \mathsf{Ras}(\beta) = \mathsf{Ras}(\beta)[h'/h, a'/a].\end{array}$$

—*Suppose e is an expression, the free variables of e are a subset of $\{x_1, \ldots, x_n\}$ and the following hold.*
  —*e is well-defined in all contexts where α holds.*

$$\mathcal{J} \models \forall h, a, x_1, \ldots, x_n \bullet \mathsf{Tr}(\alpha) \Rightarrow \mathsf{Df}(e).$$

  —*$\mathcal{J}$ is a model for $\Sigma_{<d}$ (where d is the largest predicate or pure function appearing in e).*

$$\mathcal{J} \models \Sigma_{<d}.$$

  Then, the following holds.

$$\begin{array}{c}\mathcal{J} \models \forall h, h', a, a', x_1, \ldots, x_n \bullet \\ \mathsf{Tr}(\alpha) \wedge (\forall l \in \mathsf{Ras}(\alpha) \bullet l \in a' \wedge \mathit{select}(h, l) = \mathit{select}(h', l)) \\ \Downarrow \\ \mathsf{Tr}(e) = \mathsf{Tr}(e)[h'/h, a'/a].\end{array}$$

PROOF. By induction on the size of the assertion or expression. □

We now prove that in a valid program, the axioms corresponding to pure methods and predicates are true under the interpretation $\mathcal{J}$.

LEMMA 4.10. *Suppose d is a predicate or pure function in a valid program π. If $\mathcal{J} \models \Sigma_{<d}$, then $\mathcal{J} \models \Sigma_d^I \cup \Sigma_d^E$.*

PROOF. Suppose d is a predicate defined as follows:

$$\textbf{predicate } q(x_1, \ldots, x_n) = \alpha.$$

We have to prove that $\mathcal{J}$ is a model for the implementation, footprint implementation, frame, and footprint accessible axioms.

—*Implementation Axiom.* We have to show that

$$\mathcal{J} \models \forall h, a, x_1, \ldots, x_n \bullet q(h, a, x_1, \ldots, x_n) = \mathsf{Tr}(\alpha).$$

Assume that H is an arbitrary heap, A is an access set, and $c_1, \ldots, c_n$ are integers. It suffices to prove that

$$\mathcal{J}, \gamma \models q(h, a, x_1, \ldots, x_n) = \mathsf{Tr}(\alpha),$$

where $\gamma = [h \mapsto H, a \mapsto A, x_1 \mapsto c_1, \ldots, x_n \mapsto c_n]$. As π is a valid program, q's body is well-defined as

$$\Sigma_{<q} \vdash \forall h, a, x_1, \ldots x_n \bullet \mathsf{Df}(\alpha).$$

It follows from $\mathcal{J} \models \Sigma_{<q}$ and by instantiating the quantifier that

$$\mathcal{J}, \gamma \models \mathsf{Df}(\alpha).$$

From Theorem 4.6, it follows that

$$[\![\alpha]\!]_\gamma \Leftrightarrow \mathcal{J}, \gamma \models \mathsf{Tr}(\alpha)$$

and

$$[\![q(x_1, \ldots, x_n)]\!]_\gamma \Leftrightarrow \mathcal{J}, \gamma \models \mathsf{Tr}(q(x_1, \ldots, x_n)).$$

$\mathcal{J}$ is a model for the implementation axiom because $[\![q(x_1, \ldots, x_n)]\!]_\gamma = [\![\alpha]\!]_\gamma$.

—*Footprint Axiom.* Follows from Theorem 4.6, just like the implementation axiom.

—*Frame Axiom.* Follows from Lemma 4.9.

—*Footprint Accessible Axiom.* Follows from Lemmas 4.6 and 4.8.

The proofs for pure functions are similar to the proofs for predicates. □

THEOREM 4.11. *In a valid program, the interpretation $\mathcal{J}$ is a model for the axioms of all predicates and pure functions.*

$$\mathcal{J} \models \Sigma_\pi.$$

PROOF. Follows from Lemma 4.10. □

4.3. Soundness Proof

To prove soundness, we augment the program state with additional ghost information, as shown in Definitions 4.12, 4.13, and 4.14. In particular, each activation record (i.e., continuation) additionally stores the name of the function corresponding to that activation record together with its actual parameters, and the old values of the heap and access set on entry to that function. The augmented execution rules are largely similar to the original ones. The main difference is the call rule, where additional information is stored in the new activation record, as shown here.

$$\frac{\mathsf{body}(f) = \overline{r} \qquad \mathsf{params}(f) = x_1, \ldots, x_n}{\begin{array}{c}(H, A, \mathbf{exec}(\gamma,\ f(y_1, \ldots, y_n);\ \overline{s}, g(c_1, \ldots, c_n), H_0, A_0, \kappa)) \rightarrow \\ (H, A, \mathbf{exec}([x_1 \mapsto \gamma(y_1), \ldots], \overline{r},\ f(\gamma(y_1), \ldots, \gamma(y_n)), H, A, \\ \mathbf{exec}(\gamma, \overline{s}, g(c_1, \ldots, c_n), H_0, A_0, \kappa)))\end{array}}.$$

The other rules do not modify the additional ghost information.

Definition 4.12. An *augmented continuation* is either:

—**done**, indicating the program finished executing;
—**exec**$(\gamma, \overline{s}, f(c_1, \ldots, c_n), H_0, A_0, \kappa)$, where γ is an environment, $\overline{s}$ is a statement list, f is the name of the function corresponding to this activation record, $c_1, \ldots, c_n$ are its actual parameters, H_0 and A_0 are, respectively, the values of the heap and required access set on entry to the current function, and κ is an augmented continuation.

Definition 4.13. An *augmented program state* σ is either:

—**error**, indicating the program performed an illegal operation.
—(H, A, κ), where H is a heap, A is an access set, and κ is an augmented continuation.

Definition 4.14. The *augmented initial state* (denoted σ_{ini}) is

$$([\_ \mapsto 0], \emptyset, (\textbf{exec}([\_ \mapsto 0], \overline{s}, \mathit{main}, [\_ \mapsto 0], \emptyset, \textbf{done}))).$$

Our soundness Theorem 4.19 states that valid programs do not go wrong. That is, executions of valid program never end in the **error** state. To prove soundness, we define the concept of valid state shown in Definitions 4.15 and 4.16. The **error** state is not a valid program state. We show soundness (Theorem 4.19) by proving that the initial state is valid (Lemma 4.17) and by demonstrating that $\rightarrow$ preserves validity (Lemma 4.18).

Definition 4.15 states that a non-final continuation is valid if each activation record (1) satisfies its method contract and (2) no modifications were made to memory locations allocated before execution of the function but outside of its precondition's footprint. Property (2) is crucial for proving that the assumed frame conditions in the verification condition for function call hold in executions of valid programs.

Definition 4.15. An augmented continuation is valid with respect to a heap H and access A if the following hold.

—**done**. Always.
—**exec**$(\gamma, \overline{s}, f(c_1, \ldots, c_n), H_0, A_0, \kappa)$, where P and Q are respectively f's pre- and postcondition. All of the following hold.

(i) The postcondition is true.

$$\mathcal{J}, \gamma[h \mapsto H, a \mapsto A - (A_0 - [\![P]\!]^R_{\gamma[h \mapsto H_0, a \mapsto A_0]})] \models \underset{\mathsf{vc}(\overline{s}, \mathsf{Tr}(Q))}{\cdot}$$

(ii) The heap and access set are framed by the set of locations not required to be accessible by the method precondition such that

$$\mathcal{J}, \gamma[h \mapsto H, a \mapsto A, h_0 \mapsto H_0, a_0 \mapsto A_0 - [\![P]\!]^R_{\gamma[h \mapsto H_0, a \mapsto A_0]}] \models \underset{\forall l \in a_0 \bullet l \in a \wedge \mathit{select}(h, l) = \mathit{select}(h_0, l)}{\cdot}$$

(iii) The continuation κ extended with $f(c_1, \ldots, c_n)$ is valid with respect to H_0 and A_0. Extending **done** with $f(c_1, \ldots, c_n)$ equals **done**. Extending other continuations with $f(c_1, \ldots, c_n)$ corresponds to adding $f(c_1, \ldots, c_n)$ as the first statement in the continuation's statement list.

Definition 4.16. A program state σ is valid (denoted $\mathsf{valid}(\sigma)$) if the following hold.

—**error**. Never.
—(H, A, κ). κ is valid with respect to H and A.

The key to framing in implicit dynamic frames is the property that a function cannot modify an existing memory location unless the function's precondition requires that location to be accessible. As shown in Figure 8, this property is encoded via free postconditions in the verification condition for function calls. Validity of a continuation executing a function f shown in Definition 4.15 implies that (1) this frame property

has not been violated by f so far and (2) that it will continue to hold while f executes. More specifically, the second validity condition directly states that (1) holds, while (2) follows from the first validity condition. More specifically, the verification condition of the remaining statements and f's postcondition does not have to hold for the current access set but for a subset thereof. This subset does not include memory locations allocated before f unless they are required to be accessible by f's precondition.

LEMMA 4.17. *The initial state of a valid program is valid.*

LEMMA 4.18 (PRESERVATION). *In a valid program, $\rightarrow$ preserves validity. That is,*

$$\forall \sigma, \sigma' \bullet \mathsf{valid}(\sigma) \wedge \sigma \rightarrow \sigma' \Rightarrow \mathsf{valid}(\sigma').$$

PROOF. Suppose σ is

$$(H, A, (\gamma, \bar{r}, f(c_1, \ldots, c_n), H_0, A_0, \kappa)),$$

where $x_1, \ldots, x_n$, P, and Q, respectively, represent the parameters and pre- and postcondition of f. By case analysis on $\bar{r}$. Note that we typically only need to consider validity of the first continuation, as validity of nested continuations does not depend on the current values of H and A. We highlight the most interesting cases.

—Suppose the statement list is $x := e;\ \bar{s}$. It follows from the validity of σ that

$$\mathcal{J}, \gamma[h \mapsto H, a \mapsto A - (A_0 - [\![P]\!]^R_{\gamma[h \mapsto H_0, a \mapsto A_0]})] \models \mathsf{Df}(e) \wedge \mathsf{vc}(\bar{s}, \mathsf{Tr}(Q))[\mathsf{Tr}(e)/x].$$

As e is well-defined, it follows from Theorem 4.6 that $H, A - (A_0 - [\![P]\!]^R_{\gamma[h \mapsto H_0, a \mapsto A_0]}), \gamma, e \downarrow c$ and that $\mathcal{J}, \gamma[h \mapsto H, a \mapsto A - (A_0 - [\![P]\!]^R_{\gamma[h \mapsto H_0, a \mapsto A_0]})] \models \mathsf{Tr}(e) = c$ with $c \neq \mathbf{error}$. Moreover, it follows from Lemma 4.7 that $H, A, \gamma, e \downarrow c$. As a consequence, the variable assignment does not go wrong. Moreover,

$$\mathcal{J}, \gamma[h \mapsto H, a \mapsto A - (A_0 - [\![P]\!]^R_{\gamma[h \mapsto H_0, a \mapsto A_0]}), x \mapsto c] \models \mathsf{vc}(\bar{s}, \mathsf{Tr}(Q)).$$

Therefore, the first validity condition holds. The second validity condition holds because neither the heap nor the access set are modified. The third validity condition holds as the validity of nested continuations does not depend on the environment of the outermost continuation.

—Suppose the statement list is $[x_1] := x_2;\ \bar{s}$. It follows from the validity of σ that

$$\mathcal{J}, \gamma[h \mapsto H, a \mapsto A - (A_0 - [\![P]\!]^R_{\gamma[h \mapsto H_0, a \mapsto A_0]})] \models x_1 \in a \wedge \mathsf{vc}(\bar{s}, \mathsf{Tr}(Q))[store(h, x_1, x_2)/h].$$

It then follows from the definition of $\mathcal{J}(\in)$ that $\gamma(x_1) \in A - (A_0 - [\![P]\!]^R_{\gamma[h \mapsto H_0, a \mapsto A_0]})$, and hence that $\gamma(x_1) \in A$. As $\gamma(x_1)$ is allocated, the heap update does not go wrong. Moreover, it follows from the definition of $\mathcal{J}(store)$ that

$$\mathcal{J}, \gamma[h \mapsto H[\gamma(x_1) \mapsto \gamma(x_2)], a \mapsto A - (A_0 - [\![P]\!]^R_{\gamma[h \mapsto H_0, a \mapsto A_0]})] \models \mathsf{vc}(\bar{s}, \mathsf{Tr}(Q)).$$

Hence, the first validity condition holds. Because $\gamma(x_1) \in A - (A_0 - [\![P]\!]^R_{\gamma[h \mapsto H_0, a \mapsto A_0]})$, it follows that $\gamma(x_1) \notin A_0 - [\![P]\!]^R_{\gamma[h \mapsto H_0, a \mapsto A_0]}$. As a consequence, the second validity condition holds. The third validity condition holds as validity of nested continuations does not depend on the current values of the heap and access set.

—Suppose the statement list is $g(y_1, \ldots, y_m);\ \bar{s}$ where g is defined as

$$\mathbf{void}\ g(z_1, \ldots, z_m)\ \mathbf{requires}\ P_g;\ \mathbf{ensures}\ Q_g;\ \{\ \bar{t}\ \}.$$

Execution of a function call never goes wrong, hence $\sigma' \neq \textbf{error}$. It follows from the validity of σ that

$$\mathcal{J}, \gamma[h \mapsto H, a \mapsto A - (A_0 - [\![P]\!]^R_{\gamma[h \mapsto H_0, a \mapsto A_0]})] \models \mathsf{Tr}(P_g)(y_1/z_1, \ldots, y_m/z_m)$$

Therefore,

$$\mathcal{J}, \gamma'[h \mapsto H, a \mapsto A - (A_0 - [\![P]\!]^R_{\gamma[h \mapsto H_0, a \mapsto A_0]})] \models \mathsf{Tr}(P_g),$$

where $\gamma' = [z_1 \mapsto \gamma(y_1), \ldots, z_m \mapsto \gamma(y_m)]$. As the program is valid, g is a valid function. Therefore, g's implementation satisfies its specification:

$$\Sigma_{<g} \vdash \forall h, a, z_1, \ldots, z_m \bullet \mathsf{Tr}(P_g) \Rightarrow \mathsf{vc}(\bar{t}, \mathsf{Tr}(\overline{Q_g})).$$

It follows from Theorem 4.11 that $\mathcal{J} \models \Sigma_{<g}$. As a consequence, the following holds.

$$\mathcal{J} \models \forall h, a, z_1, \ldots, z_m \bullet \mathsf{Tr}(P_g) \Rightarrow \mathsf{vc}(\bar{t}, \mathsf{Tr}(\overline{Q_g})).$$

By instantiating the quantifier and because $\mathsf{Tr}(P_g)$ is true, it follows that

$$\mathcal{J}, \gamma'[h \mapsto H, a \mapsto A - (A_0 - [\![P]\!]^R_{\gamma[h \mapsto H_0, a \mapsto A_0]})] \models \mathsf{vc}(\bar{t}, \mathsf{Tr}(Q_g))$$

Therefore, the first validity condition holds for the new outermost continuation. The second validity condition holds because the current values of the heap and access set are equal to the old values of the heap and access set. The continuation performing the function call remains valid with respect to the current heap and access set.

—Suppose the statement list is empty. Returning from a function never goes wrong. If the continuation κ is **done**, then σ' is trivially valid. Otherwise, suppose that the continuation κ is equal to

$$(\gamma', \bar{t}, g(c'_1, \ldots, c'_m), H'_0, A'_0, \kappa').$$

We start by proving that the first validity condition holds. It follows from the validity of σ that the first validity condition of κ' extended with $f(c_1, \ldots, c_n)$ holds (1).

$$\begin{array}{c}
\mathcal{J}, \gamma'[h \mapsto H_0, a \mapsto A_0 - (A'_0 - [\![P_g]\!]^R_{\gamma'[h \mapsto H'_0, a \mapsto A'_0]})] \models \\
\mathsf{Tr}(P) \wedge (\forall h', a' \bullet \\
\mathsf{Tr}(Q)[h'/h, a'/a, y_1/x_1, \ldots, y_1/x_n] \wedge \\
(\forall l \bullet l \in a - \mathsf{Ras}(P[y_1/x_1, \ldots, y_n/x_n]) \Rightarrow a \in a' \wedge \mathit{select}(h, l) = \mathit{select}(h', l)) \wedge \\
\mathsf{Ras}(Q)[h'/h, a'/a, y_1/x_1, \ldots, y_1/x_n] \cap a \subseteq \mathsf{Ras}(P)[y_1/x_1, \ldots, y_n/x_n] \\
\Downarrow \\
\mathsf{vc}(\bar{t}, Q_g)[h'/h, a'/a]).
\end{array}$$

To demonstrate the first validity condition for σ', it suffices to instantiate h' with H and a' with $A - (A'_0 - [\![P_g]\!]^R_{\gamma[h \mapsto H'_0, a \mapsto A'_0]})$ in the preceding quantifier and to prove the three premises of the implication.

(1) It follows from the validity of σ that the first validity condition of κ holds:

$$\mathcal{J}, \gamma[h \mapsto H, a \mapsto A - (A_0 - [\![P]\!]^R_{\gamma[h \mapsto H_0, a \mapsto A_0]}) \models \mathsf{Tr}(Q).$$

It follows from the second validity condition of κ' that $(A'_0 - [\![P_g]\!]^R_{\gamma'[h \mapsto H'_0, a \mapsto A'_0]}) \subseteq A_0$. Moreover $[\![P]\!]^R_{\gamma[h \mapsto H_0, a \mapsto A_0]} \cap (A'_0 - [\![P_g]\!]^R_{\gamma'[h \mapsto H'_0, a \mapsto A'_0]}) = \emptyset$, because P holds in

access set $A_0 - (A'_0 - [\![P_g]\!]^R_{\gamma'[h\mapsto H'_0, a\mapsto A'_0]})$ and Lemma 4.8. As a consequence, $(A'_0 - [\![P_g]\!]^R_{\gamma'[h\mapsto H'_0, a\mapsto A'_0]}) \subseteq A_0 - [\![P]\!]^R_{\gamma[h\mapsto H_0, a\mapsto A_0]}$. It then follows from Lemma 4.7 that

$$\mathcal{J}, \gamma[h \mapsto H, a \mapsto A - (A'_0 - [\![P_g]\!]^R_{\gamma'[h\mapsto H'_0, a\mapsto A'_0]}) \models \mathsf{Tr}(Q).$$

(2) It follows from the validity of σ that the second validity condition of κ holds:

$$\mathcal{J}, \gamma[h \mapsto H, a \mapsto A, h_0 \mapsto H_0, a_0 \mapsto A_0 - [\![P]\!]^R_{\gamma[h\mapsto H_0, a\mapsto A_0]}] \models \forall l \in a_0 \bullet l \in a \wedge select(h, l) = select(h_0, l).$$

As P is well-defined (follows from the validity of the program and $\mathcal{J} \models \Sigma_\pi$) and true, it follows from Theorem 4.6 that

$$\mathcal{J}, \gamma[h \mapsto h_0, a \mapsto A_0] \models \mathsf{Ras}(P) = [\![P]\!]^R_{\gamma[h\mapsto H_0, a\mapsto A_0]}.$$

Therefore,

$$\mathcal{J}, \gamma[h \mapsto H, a \mapsto A, h_0 \mapsto H_0, a_0 \mapsto A_0] \models \forall l \in a_0 - \mathsf{Ras}(P)[h_0/h, a_0/a] \bullet l \in a \wedge select(h, l) = select(h_0, l).$$

The statement still holds if we subtract the same set from both a and 0.

$$\mathcal{J}, \gamma[h \mapsto H, a \mapsto A - [\![P_g]\!]^R_{\gamma'[h\mapsto H'_0, a\mapsto A'_0]}, h_0 \mapsto H_0, a_0 \mapsto A_0 - [\![P_g]\!]^R_{\gamma'[h\mapsto H'_0, a\mapsto A'_0]}] \models \forall l \in a_0 - \mathsf{Ras}(P)[h_0/h, a_0/a] \bullet l \in a \wedge select(h, l) = select(h_0, l).$$

(3) As Q holds in the function post-state, it follows from Lemma 4.8 that

$$[\![Q]\!]_{\gamma[h\mapsto H, a\mapsto A-(A_0-[\![P]\!]^R_{\gamma[h\mapsto H_0, a\mapsto A_0]})]} \subseteq A - (A_0 - [\![P]\!]^R_{\gamma[h\mapsto H_0, a\mapsto A_0]}).$$

As a consequence,

$$[\![Q]\!]_{\gamma[h\mapsto H, a\mapsto A-(A_0-[\![P]\!]^R_{\gamma[h\mapsto H_0, a\mapsto A_0]})]} \cap A_0 \subseteq [\![P]\!]^R_{\gamma[h\mapsto H_0, a\mapsto A_0]}.$$

Therefore,

$$[\![Q]\!]_{\gamma[h\mapsto H, a\mapsto A-(A_0-[\![P]\!]^R_{\gamma[h\mapsto H_0, a\mapsto A_0]})]} \cap A_0 - (A'_0 - [\![P_g]\!]^R_{\gamma'[h\mapsto H'_0, a\mapsto A'_0]}) \subseteq [\![P]\!]^R_{\gamma[h\mapsto H_0, a\mapsto A_0]}.$$

It follows from Theorem 4.6 that

$$\mathcal{J}, \gamma'[h \mapsto H_0, a \mapsto A_0 - (A'_0 - [\![P_g]\!]^R_{\gamma'[h\mapsto H'_0, a\mapsto A'_0]})] \models \mathsf{Ras}(Q)[h'/h, a'/a, y_1/x_1, \ldots, y_1/x_n] \cap a \subseteq \mathsf{Ras}(P)[y_1/x_1, \ldots, y_n/x_n].$$

We can now prove that the first validity condition holds for σ' by combining (1), (2), and (3) such that

$$\mathcal{J}, \gamma'[h \mapsto H, a \mapsto A - (A'_0 - [\![P_g]\!]^R_{\gamma'[h\mapsto H'_0, a\mapsto A'_0]})] \models \mathsf{vc}(\bar{t}, Q_g).$$

Thus, the first validity condition holds for σ'. We now prove the second validity condition. It follows from the validity of σ that the second validity condition for κ holds:

$$\mathcal{J}, \gamma[h \mapsto H, a \mapsto A, h_0 \mapsto H_0, a_0 \mapsto A_0 - [\![P]\!]^R_{\gamma[h\mapsto H_0, a\mapsto A_0]}] \models \forall l \in a_0 \bullet l \in a \wedge select(h, l) = select(h_0, l).$$

It also follows from the validity of σ that the second validity condition for κ' holds:

$$\mathcal{J}, \gamma[h \mapsto H_0, a \mapsto A_0, h_0 \mapsto H'_0, a_0 \mapsto A'_0 - [\![P_g]\!]^R_{\gamma[h\mapsto H'_0, a\mapsto A'_0]}] \models \forall l \in a_0 \bullet l \in a \wedge select(h, l) = select(h_0, l).$$

Suppose l is an arbitrary element of $A'_0 - [\![P_g]\!]^R_{\gamma'[h\mapsto H'_0, a\mapsto A'_0]}]$. It follows from the second validity condition for κ' that $l \in A_0$ and $H'_0(l) = H_0(l)$. As $l \notin [\![P]\!]^R_{\gamma[h\mapsto H_0, a\mapsto A_0]}]$, it

follows from the second validity condition of κ that $l \in A$ and $H(l) = H_0(l)$, and hence that $H'_0(l) = H(l)$. Therefore, the second validity condition holds. The third validity condition for σ' holds as neither H'_0 nor A'_0 are modified. We may conclude that σ' is valid. □

THEOREM 4.19 (SOUNDNESS). *Valid programs do not go wrong.*

$$\sigma_{\mathsf{ini}} \not\rightarrow^* \textbf{error}.$$

PROOF. Follows from validity of the initial state (Lemma 4.17) and preservation of validity (Lemma 4.18) for valid programs. □

5. EXAMPLES

In this section, we show how our approach handles certain common specification and programming patterns. All programs shown below have been verified using our verifier prototype (see Section 6).

We will write our code examples in C#-style syntax. In particular, method calls are statically bound unless the callee is marked **virtual** or **override**.

5.1. ArrayList

An invariant defines what it means for an object to be in a consistent state. For example, consider the class *ArrayList* from Figure 13. An *ArrayList* object is consistent if its internal array, *items*, is non-null and the number of elements, *count*, lies between zero and the length of the array. In implicit dynamic frames, predicates play the role of invariants. That is, predicates typically not only describe what memory locations are part of the data structure, but also constrain their values.

Our approach does not impose any built-in rules that indicate when invariants must hold. Instead, if a developer wants an object to be consistent at a certain program point, then he or she should explicitly say so. The absence of a (potentially restrictive) methodology for object invariants was also advocated by Parkinson [Parkinson 2007]. For example, the method *add* requires the invariant to hold an entry to the method and guarantees that it holds again when the method returns. *add*'s method contract specifies this property by including *valid*() in both its pre- and postcondition.

Note that the predicate *valid* includes **acc**(*items.elems*). We assume each array has an *elems* ghost field which holds the elements of the array. Holding the permission to access the *elems* field gives an activation record the right to read and update the array elements. For example, the method *get* is allowed to mention *items*[*index*] since *valid* includes **acc**(*items.elems*). Note that the predicate *valid* can read *items.Length*, without requiring any corresponding permission as the length of an array, is immutable. Finally, the variable i in a quantifier $\forall\, i \in (a : b) \bullet \ldots$ ranges from a (inclusive) to b (exclusive).

The postconditions of *add* and *remove* contain **old** expressions. In general, **old**(e) denotes the pre-state value of the expression e. More specifically, if **old**(e) occurs within a postcondition, then the expression denotes the value of e right before execution of the method; if **old**(e) occurs within a loop invariant, it denotes the value of e right before the condition of the loop is checked for the first time. For example, **old**(*size*()) in *add*'s postcondition denotes the size of the list right before execution of *add*. More formally, we can extend the definitions of Tr and Df of Figure 6 with **old** expressions as follows.

$$\begin{aligned} \mathsf{Tr}(\textbf{old}(e)) &\equiv \mathsf{Tr}(e)[h_{old}/h, a_{old}/a]; \\ \mathsf{Df}(\textbf{old}(e)) &\equiv \mathsf{Df}(e)[h_{old}/h, a_{old}/a]. \end{aligned}$$

Here h_{old} and a_{old} denote the pre-state values of the heap and access set.

We have omitted **old** expressions from the formalization to streamline the presentation. The encoding of the **old** construct we use in our implementation is similar to

```
class ArrayList<T> {
  T[] items; int count;

  predicate valid() = acc(items) * acc(count) *
    acc(items.elems) * 0 ≤ count * count ≤ items.Length;

  pure int size()
    requires valid();
    ensures 0 ≤ result;
  { return count; }

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

  ArrayList()
    ensures valid() * size() = 0;
  { items := new T[4]; }

  void add(T item)
    requires valid();
    ensures valid() * size() = old(size()) + 1 * get(size() − 1) = item;
    ensures ∀ i ∈ (0 : size() − 1) • get(i) = old(get(i));
  {
    if(count = items.Length) {
      T[] tmp := new T[2 * items.length + 1];
      Array.copy(items, tmp, items.Length);
      items := tmp;
    }
    items[count++] := item;
  }

  T remove(int index)
    requires valid() * 0 ≤ index * index < size();
    ensures valid() * size() = old(size()) − 1;
    ensures ∀i ∈ (0 : index) • get(i) = old(get(i));
    ensures ∀i ∈ (index : size()) • get(i) = old(get(i + 1));
  {
    T tmp := items[index];
    Array.copy(items, index + 1, items, index, −−count − index); }
    return tmp;
  }
}
```

Fig. 13. The class *ArrayList*. The predicate *valid* plays the role of the object invariant.

```
class Stack<T> {
  ArrayList<T> elements;

  predicate valid() = acc(elements) * elements.valid();

  pure int size()
    requires valid();
    ensures 0 ≤ result;
  { return elements.size(); }

  pure T get(int index)
    requires valid() * 0 ≤ index * index < size();
  { return elements.get(index); }

  Stack()
    ensures valid() * size() = 0;
  { elements := new ArrayList<T>(); }

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

  T pop()
    requires valid() * 0 < size();
    ensures valid() * size() == old(size()) − 1;
    ensures ∀ i ∈ (0 : size()) • get(i) = old(get(i));
    ensures result = old(get(size() − 1));
  { return elements.remove(elements.size() − 1); }
}
```

Fig. 14. The class *Stack*.

the one used in other verification tools, such as the Spec# program verifier [Barnett et al. 2004] and ESC/Java [Flanagan et al. 2002]. In particular, the fact that postconditions can mention **old** expressions must be taken into account in the definition of valid function and in the verification condition for function calls by substituting the actual values of the pre-state heap and access set for h_{old} and a_{old}.

5.2. Stack

Classes typically build upon other classes to implement their interface. For example, a stack can be implemented on top of a list which holds the stack's elements. Such objects that internally use other helper objects are sometimes called *aggregate objects*. In ownership-based approaches, the helper object is said to be owned by the aggregate object.

As an example, consider the class *Stack* from Figure 14. This class implements stacks on top of the class *ArrayList* from Figure 13. As shown in the definition of the predicate *valid*, the consistency of a *Stack* object implies the consistency of the

internal list. Note that the invariant uses a separating conjunction to express that the validity of the internal list is independent from the value of the field *elements*. This independence is crucial in the correctness proof of *Stack* for two reasons. First, it ensures that assignments to *elements* do not invalidate or change the return values of the pure methods of the internal list. Second, the separating conjunction guarantees that *elements*'s mutators cannot modify the *elements* field of the stack. For example, the call $elements.add(o)$ in the method *push* cannot set *elements* to **null**, since the permission to update this field is not included in $elements.valid()$.

Our approach 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 example, consider the following method *getElements*.

$$
\begin{array}{l}
\mathit{ArrayList}\ \mathit{getElements}() \\
\quad \textbf{requires}\ \mathit{valid}(); \\
\quad \textbf{ensures}\ \mathit{valid}() \wedge \mathit{result}.\mathit{valid}(); \\
\{\ \textbf{return}\ \mathit{elements};\ \}
\end{array}
$$

This method leaks a reference to the internal list to client code. However, clients will not able to establish that the validity of the *Stack* object is independent from the validity of the list returned by *getElements* because the method's postcondition uses a normal conjunction instead of a separating one. In other words, client code cannot falsely assume that the footprint of the *Stack* object is disjoint from the footprint of the *ArrayList* object. As a consequence, when client code updates the helper object, it loses all information about the aggregate object. Therefore, clients cannot falsely assume that the state of an aggregate object is not affected when one of the helper objects is modified. Clients can, however, safely call pure methods on the helper object without losing information about the aggregate object.

As explained in Section 3.5.2, each predicate has a corresponding footprint. This footprint is the set of memory locations required to be accessible by that predicate. The contents of the footprint can change over time. For example, the set of locations required to be accessible by $valid()$ is different before and after invoking the following *switch* method.

$$
\begin{array}{l}
\textbf{void}\ \mathit{switch}(\mathit{Stack}\ \mathit{other}) \\
\quad \textbf{requires}\ \mathit{valid}() * \mathit{other}.\mathit{valid}(); \\
\quad \textbf{ensures}\ \mathit{valid}() * \mathit{other}.\mathit{valid}(); \\
\{ \\
\quad \mathit{ArrayList}\ \mathit{tmp} := \mathit{elements}; \\
\quad \mathit{elements} := \mathit{other}.\mathit{elements}; \\
\quad \mathit{other}.\mathit{elements} := \mathit{tmp};\ \} \\
\}
\end{array}
$$

Exchanging an internal helper object, as in the method *switch*, is sometimes called *ownership transfer*, since control of the helper object transfers from one aggregate object to another. No additional methodological machinery is needed to perform ownership transfer.

Neither *Stack*'s invariant nor the preconditions of *getElements* and *switch* explicitly mention non-nullness conditions. The reason these conditions can be omitted is that the encoding of access and instance predicate assertions includes a nullness check.

$$
\begin{array}{rcl}
\mathsf{Tr}(\textbf{acc}(e.f)) & \equiv & \mathsf{Tr}(e) \neq \mathit{null} \wedge (\mathsf{Tr}(e),\ f) \in a \\
\mathsf{Tr}(e_0.q(e_1, \ldots, e_n)) & \equiv & \mathsf{Tr}(e_0) \neq \mathit{null} \wedge q(h, a, \mathsf{Tr}(e_0), \ldots, \mathsf{Tr}(e_n)).
\end{array}
$$

For example, the body of *Stack*'s invariant is equivalent to

$$\textbf{this} \neq \textbf{null} \wedge \textbf{acc}(\mathit{elements}) * \mathit{elements} \neq \textbf{null} \wedge \mathit{elements}.\mathit{valid}().$$

Stack's interface is completely implementation-independent. In particular, the interface does not expose that the class is implemented in terms of an *ArrayList* object. Therefore, if we were to change *Stack*'s implementation, for example, by using a linked list instead of an array list, client code would not be affected. Thus, if one makes internal changes to *Stack*, one has to reverify that this new implementation satisfies the specification. However, one does not have to reverify any client code.

5.3. Iterator

The Iterator design pattern [Gamma et al. 1994, Chapter 5] has typically been challenging for ownership-based approaches because of sharing. More specifically, a list can have multiple iterators. Each iterator depends on the list, but no single iterator exclusively owns the list. In other words, the list is shared among its iterators.

As an example, consider the class *Iterator* from Figure 15. Each iterator has two fields: *list* and *index*. The former field is a reference to the corresponding list, while the latter is the index of the next element to be returned by the iterator. To be able to declare $list.size$ as an upper bound for *index*, the predicate $Iterator.valid$ includes $list.valid()$. However, including the validity of the list in the validity of the iterator means that multiple iterators for the same list are not disjointly valid. More specifically, 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. In particular, how can we ensure that $i_1.valid()$ is preserved when $i_2.next()$ is called?

The key to solving this problem is the fact that iterators share only the footprint of their list and that the locations in the list's footprint are not modified by *next*. To indicate *next* does not modify the locations in the list's footprint, the method's postcondition includes the conjunct **untouched**($getList().valid()$). **untouched** is a two-state assertion that can only be used in postconditions and loop invariants. In general, **untouched**($q(x_1, \ldots, x_n)$) states that (1) all memory locations in q's footprint remain accessible and (2) that those locations are not modified. More formally, we can extend the definitions of Tr, Df, and Ras from Figure 7 to **untouched** as follows:

$$\begin{aligned}
\mathsf{Tr}(\textbf{untouched}(q(x_1, \ldots, x_n))) &\equiv \forall l \in q_{\mathsf{FP}}(h_{old}, a_{old}, x_1, \ldots, x_n) \bullet l \in a \wedge h(l) = h_{old}(l);\\
\mathsf{Df}(\textbf{untouched}(q(x_1, \ldots, x_n))) &\equiv q(h_{old}, a_{old}, x_1, \ldots, x_n) \wedge q(h, a, x_1, \ldots, x_n);\\
\mathsf{Ras}(\textbf{untouched}(q(x_1, \ldots, x_n))) &\equiv \emptyset.
\end{aligned}$$

As in the translation of **old** expressions, h_{old} and a_{old} denote the pre-state values of the heap and access set. In our example, $i_1.valid()$ 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*.

The method *hasDuplicates* of Figure 16 shows that multiple iterators can concurrently iterate over the same list without interfering with each other. In particular, the call $i_2.next()$ in the inner loop does not affect $i_1.valid()$ nor $i_1.nextIndex()$. Note that a loop that modifies annotation must not be provided explicitly by the developer but can be inferred from the loop invariant. More specifically, the body of the loop can only modify locations required to be accessible by the loop invariant. Furthermore, the pre-state in **untouched** assertions in loop invariants is the state right before the first iteration of the loop.

The specification of Figure 15 allows client code to update the list at any time. However, doing so causes all information about the invariants of iterators to be lost, as the footprint of $Iterator().valid()$ includes the footprint of $getList().valid()$. Therefore, client code cannot falsely assume it is safe to call *next* after removing elements from the list. Note that clients can also modify the list via $Iterator.remove$. In that case, only the iterator making the modification remains valid.

Parkinson [2005], Malecha and Morrisett [2010], and Haack and Hurlin [2009] propose separation logic specifications for the iterator design pattern. Their specifications

```
class Iterator<T> {
  ArrayList<T> list; int index;

  predicate valid() =
    acc(list) * acc(index) * list.valid() * 0 ≤ index * index ≤ list.size;

  pure ArrayList getList()
    requires valid(); ensures result.valid();
  { return list; }

  pure int nextIndex()
    requires valid(); ensures 0 ≤ result * result ≤ getList().size();
  { return index; }

  pure boolean hasNext()
    requires valid();
    ensures result = (nextIndex() < getList().size());
  { return index ≠ list.size; }

  Iterator(ArrayList<T> list, int index)
    require list.valid() * 0 ≤ index * index ≤ list.size();
    ensures valid() ∧ getList() = list ∧ untouched(list.valid()) ∧ nextIndex() = index;
  { this.list := list; this.index := index; }

  T next()
    requires valid() * hasNext();
    ensures valid() * getList() = old(getList());
    ensures untouched(getList().valid());
    ensures nextIndex() = old(nextIndex()) + 1 * result = getList().get(old(nextIndex()));
  { return list.items[index++]; }

  void remove()
    requires valid() * 0 < nextIndex();
    ensures valid() * getList() = old(getList()) * nextIndex() = old(nextIndex()) − 1;
    ensures getList().size() = old(getList().size()) − 1;
    ensures ∀i ∈ (0 : nextIndex()) • getList().get(i) = old(getList().get(i));
    ensures ∀i ∈ (nextIndex() : getList().size()) • getList().get(i) = old(getList().get(i + 1));
  { list.remove(−−index); }
}
```

Fig. 15. The class *Iterator*.

are based on fractional permissions [Boyland 2003]. More specifically, their specifications support multiple iterators for the same list by including only a fraction of the list predicate in the iterator predicate. This fraction of the list predicate permits the methods *hasNext* and *next* to read the fields of the corresponding list. An advantage of our approach is that fractions are not required. A disadvantage, however, is that

```
boolean hasDuplicates(ArrayList<T> l)
  requires l.valid();
  ensures l.valid() ∗ untouched(l.valid());
  ensures result = (∃i, j ∈ (0 : l.size()); i ≠ j ∧ l.get(i) = l.get(j));
{
  Iterator<T> i1 := new Iterator<T>(l, 0);
  while(i1.hasNext())
    invariant i1.valid();
    invariant untouched(l.valid()) ∗ i1.getList() = l;
    invariant ∀i ∈ (0 : l.size()), j ∈ (0 : i1.nextIndex()) • i = j ∨ l.get(i) ≠ l.get(j);
  {
    T o1 := i1.next();
    Iterator<T> i2 := new Iterator<T>(l, i1.nextIndex());
    int tmp := i1.nextIndex();
    while(i2.hasNext())
      invariant i2.valid() ∗ untouched(l.valid()) ∗ i2.getList() = l;
      invariant tmp ≤ i2.nextIndex();
      invariant ∀i ∈ (tmp : i2.nextIndex()) • o1 ≠ l.get(i);
    {
      T o2 := i2.next();
      if(o1 = o2) { return true; }
    }
  }
  return false;
}
```

Fig. 16. The method *hasDuplicates* implemented via two concurrent iterators.

we do not support multiple threads that concurrently iterate over a single collection (without additional synchronization), as only one of the threads can hold $list.valid()$ at any point in time. A slightly different specification in higher-order separation logic is given by Krishnaswami [2006]. In Krishnaswami's specification, the iterator predicate is parametrized with a predicate P describing the state of the list. When an iterator is created, the state of the list P is set in the iterator predicate. The iterator can only be used as long as the corresponding list is in state P.

5.4. LinkedList

Many verification tools [Flanagan et al. 2002; Barnett et al. 2004; Dahlweid et al. 2009; Leino 2010; Leino and Müller 2009; Pariente and Ledinot 2010] use first-order theorem provers, such as Z3 [de Moura and Bjørner 2008] and Simplify [Detlefs et al. 2005] to discharge proof obligations. A disadvantage of these first-order provers is that they do not automatically perform induction or transitive closure. However, induction is typically a crucial step when proving properties of linked data structures, such as linked lists or trees. Therefore, verifying linked data structures using first-order provers is challenging.

Two approaches have been proposed to address this challenge: (1) using ghost state and quantifiers to avoid induction [Zee et al. 2008] or (2) using recursive predicates [Reynolds 2002] and lemma methods [Jacobs et al. 2010] to perform induction. The former approach is discussed in Section 5.4.1 and the latter in Section 5.4.2.

```
class Node<T> { T val; Node<T> next; }

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

  predicate valid() = acc(head) * acc(nodes) *
    (∀n ∈ nodes • acc(n.next) ∧ acc(n.value)) *
    (∀i, j ∈ (0 : |nodes|) • i = j ∨ nodes[i] ≠ nodes[j]) *
    (∀i ∈ (0 : |nodes| − 1) • nodes[i].next = nodes[i + 1]) *
    (head = null ? nodes = nil :
      nodes[0] = head ∧ nodes[|nodes| − 1].next = null);

  pure int size()
    requires valid();
    ensures 0 ≤ result;
    returning |nodes|;
  {
    int c := 0; Node<T> curr := head;
    pure while(curr ≠ null)
      invariant valid() * 0 ≤ c;
      invariant curr = null ? c = |nodes| : curr = nodes[c];
    { curr := curr.next; c++; }
    return c;
  }

  pure T get(int index)
    requires valid() * 0 ≤ index * index < size();
    returning nodes[index];
  {
    int c := 0; Node<T> curr := head;
    pure while(c < index)
      invariant valid() * 0 ≤ c * c ≤ index;
      invariant curr = nodes[c];
    { curr := curr.next; c++; }
    return curr;
  }

  LinkedList()
    ensures valid() * size() = 0;
  { head := null; nodes := nil; }
```

Fig. 17. The class *LinkedList* specified via ghost fields (part 1).

5.4.1. Specification via Ghost Fields. The prototypical example of a linked data structure is a linked list. As an example, consider the singly linked list from Figures 17 and 18. In addition to the standard *head* field, the class *LinkedList* declares a ghost field called *nodes*—a mathematical list representing the sequence of *Node* objects reachable from

```
void add(T item)
  requires valid();
  ensures valid() * size() = old(size()) + 1 * get(size() - 1) = item;
  ensures ∀ i ∈ (0 : size() - 1) • get(i) = old(get(i));
{
  if(head = null) {
    head := new Node<T>(item, null);
    nodes := seq(head);
  } else {
    int c := 0; Node<T> curr := head;
    pure while(curr.next ≠ null)
      invariant valid() * curr ≠ null;
      invariant 0 ≤ c * c < |nodes| * curr = nodes[c];
    { curr := curr.next; c++; }
    curr.next := new Node<T>(item, null);
    nodes := nodes @ seq(curr.next);
  }
 }
}
```

Fig. 18. The class *LinkedList* specified via ghost fields (part 2).

head. We use the following notation for mathematical lists. **nil** and **seq**$(x_1, \ldots, x_n)$ respectively denote the empty list and the list with elements $x_1, \ldots, x_n$. $|l|$ and $l[i]$ respectively denote the length and ith element of list l. Finally, @ represents the append operation.

The specification of *LinkedList* avoids recursive definitions (and therefore also induction) by quantifying over the nodes in the mathematical list in the invariant. More specifically, the invariant holds only if (1) *head* and *node* are accessible, (2) all elements of *nodes* are non-null and their fields are accessible, (3) the elements are distinct, (4) the ghost list is consistent with the values of the next pointers, and (5) either head is **null** and *nodes* is empty or *head* points to the first element of the sequence and the *next* field of the last *Node* object points to **null**. In other words, the invariant relates the ghost state (stored in *nodes*) to the actual, real fields. To keep the ghost state in sync with the real fields and to preserve the invariant, the ghost fields must be updated after each modification of the data structure. For example, the constructor initializes *nodes* to the empty sequence. Similarly, both branches of the if statement in the method *add* update *nodes* in order to reflect the addition of a new node. Having to update the ghost fields is a minor disadvantage of the approach discussed in this section.

The syntax of Figure 5 enforces that the body of a pure method consists of a single return statement returning a side-effect-free expression. As explained in Section 3.4, a pure method's implementation axiom is derived from this expression. However, the body of the pure method *size* of Figure 17 does not consist of a single return statement. Instead, the body consists of a sequence of side-effect-free statements and a **returning** annotation is provided which links the return value to the ghost list. This annotation can be considered to be an additional postcondition when verifying the body itself. Moreover, instead of deriving the implementation axiom from the implementation, it is derived from the **returning** annotation. Note that this annotation is not part of the public specification, and for that reason—like the implementation axiom—it is not

```
class Node<T> { T val; Node<T> next; }

class LinkedList<T> {
  Node head;

  static predicate lseg(Node<T> from, Node<T> to) =
    from = to ? true : acc(from.val) * acc(from.next) * lseg(from.next, to);

  pure static int lseg_length(Node<T> from, Node<T> to)
    requires lseg(from, to);
    ensures 0 ≤ result;
  { return from = to ? 0 : lseg_length(from.next, to) + 1; }

  pure static T lseg_get(Node<T> from, Node<T> to, int index)
    requires lseg(from, to) * 0 ≤ index * index < lseg_length(from, to);
  { return index = 0 ? from.val : lseg_get(from.next, to, index − 1); }

  predicate valid() = acc(head) * lseg(head, null);

  pure int size()
    requires valid();
    ensures 0 ≤ result;
  { return lseg_length(head, null); }

  pure T get(int index)
    requires valid() * 0 ≤ index * index < size();
  { return lseg_get(head, null, index); }

  LinkedList()
    ensures valid() * size() = 0;
  { head := null; }
```

Fig. 19. The class *LinkedList* specified via recursive predicates and methods (part 1).

included in any way in the pure method's external theory. The **pure** annotation on the loop in the body of *size* indicates the loop does not have side effects.

5.4.2. Specification via Recursive Predicates and Methods. Instead of relying on ghost state and quantification, one can specify linked data structures via recursive definitions. As an example, consider the class *LinkedList* from Figures 19 and 20. The key element in the specification of this class is the recursive predicate *lseg* which represents list segments. More specifically, $lseg(from, to)$ holds if either *from* equals *to* or if *from* is non-null, its fields are accessible, and there is a valid list segment starting at $from.next$ and ending in *to*. The recursive pure methods *lseg_length* and *lseg_get* respectively return the length and the ith element of a list segment. As shown in the definition of the predicate *valid*, a *LinkedList* object is consistent if *head* is the start of a valid list segment ending in **null**. The implementations of *size* and *get* internally use *lseg_length* and *lseg_get*.

A disadvantage of this approach is that the axioms generated for recursive predicates and pure methods are prone to matching loops [Detlefs et al. 2005, Section 5]. Matching loops can dramatically increase verification time. Moreover, proving correctness of a

```
lemma static void appendLemma(Node<T> a, Node<T> b, Node<T> c)
  requires lseg(a, b) * lseg(b, c) * (c ≠ null ⇒ acc(c.next));
  ensures lseg(a, c) * (c ≠ null ⇒ acc(c.next) ∧ c.next = old(c.next));
  ensures lseg_length(a, c) = old(lseg_length(a, b) + lseg_length(b, c));
  ensures ∀i ∈ (0 : old(lseg_length(a, b))) • lseg_get(a, c, i) = old(lseg_get(a, b, i));
  ensures ∀i ∈ (0 : old(lseg_length(b, c))) •
    lseg_get(a, c, old(lseg_length(a, b)) + i) = old(lseg_get(b, c, i));
{ if(a ≠ b) appendLemma(a.next, b, c); }

void add(T item)
  requires valid();
  ensures valid() * size() = old(size()) + 1 * get(size() − 1) = item;
  ensures ∀ i ∈ (0 : size() − 1) • get(i) = old(get(i));
{
  if(head = null) {
    head := new Node<T>(item, null);
  } else {
    Node<T> curr := head; int currIndex :=0;
    while(curr.next ≠ null)
      invariant curr ≠ null * acc(head) * lseg(head, curr) * lseg(curr, null);
      invariant old(lseg_length(head, null)) =
        lseg_length(head, curr) + lseg_length(curr, null);
      invariant currIndex = lseg_length(head, curr);
      invariant ∀i ∈ (0 : lseg_length(head, curr))•
        lseg_get(head, curr, i) = old(lseg_get(head, null, i));
      invariant ∀i ∈ (0 : lseg_length(curr, null))•
        lseg_get(curr, null, i) = old(lseg_get(head, null, i + currIndex));
    {
      Node<T> oldcurr := curr; curr := curr.next; currIndex++;
      appendLemma(head, oldcurr, curr);
    }
    curr.next := new Node<T>(item, null);
    appendLemma(head, curr, null);
  }
}
```

Fig. 20. The class *LinkedList* specified via ghost fields (part 2).

program specified with recursive definitions typically requires induction. For example, the correctness of the method *add* of Figure 20 relies on the property that two list segments, $lseg(a, b)$ and $lseg(b, \mathbf{null})$, can be combined into a single list segment $lseg(a, \mathbf{null})$. 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 [Jacobs et al. 2010] in the program itself. For example, the lemma

method *appendLemma* shown next proves the aforementioned property. A more general variant of this lemma is shown and used in Figure 20.

```
lemma static void appendLemma(Node<T> a, Node<T> b)
  requires lseg(a, b) * lseg(b, null);
  ensures lseg(a, null);
{
  if(a = b) {
  } else {
    appendLemma(a.next, b);
  }
}
```

The contract of *appendLemma* corresponds to a theorem which states that a program state with two list segments, $lseg(a, b)$ and $lseg(b, \mathbf{null})$, is equivalent to a program state with a single list segment, $lseg(a, \mathbf{null})$. *appendLemma*'s body encodes an inductive proof, where the then branch of the if statement corresponds to the base case, and the recursive call in the else branch corresponds to the inductive step. Calling the lemma method (e.g., in *add*) then corresponds to applying the theorem.

Lemma methods and invocations of such methods are annotations. To guarantee that it is sound to erase these annotations, we must make sure that (1) lemmas do not modify the program state and (2) that they terminate. We enforce the former restriction syntactically: a lemma method may not assign to non-ghost fields and can only call other lemmas and pure methods. To ensure termination, we check that for each invocation of a lemma m_1 inside a lemma m_2, either m_1 appears before m_2 in the program text[2], or the footprint of m_1's precondition is a strict subset of the footprint of m_2's precondition. In our example, the recursive call in *appendLemma* is safe, as the size of the precondition's footprint decreases. In particular, the footprint of the precondition of the recursive call does not include $from.val$ and $from.next$.

The body of *add* of Figure 20 uses a loop to find the last node in the list. However, it is possible to use recursion instead. Interestingly, discharging the verification condition for an alternative, recursive implementation does not require induction (and hence no lemmas). More specifically, in a recursive implementation, an inductive argument is not necessary to discharge the verification condition itself but is part of the soundness proof. In particular, in the soundness proof, one has to show that it is safe to assume the postcondition of a recursive call when verifying the recursive method itself (typically via induction on the length of the execution). Note that even for an iterative implementation, it is possible to avoid certain lemmas by using a different specification (a loop pre- and postcondition instead of a loop invariant) and proof rule for loops, as explained by Tuerk [2010].

The specification shown in Figure 19—particularly the predicate *lseg*—is based on and largely similar to the separation logic specification for linked lists proposed by Reynolds [2002]. However, instead of pure methods, Reynolds uses an additional argument to *lseg* to describe the state of a list segment.

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

```
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

[2]The static ordering induced by the program text only applies to statically bound lemmas.

detect such dangerous predicates and pure methods. In Sections 3 and 4, we ensure consistency by imposing a simple but restrictive rule: predicates and pure methods can only call predicates and pure methods defined earlier in the program text (see Definition 3.1). This rule guarantees that predicates and pure methods terminate, which in turn implies the consistency of their axiomatization. However, this rule is overly restrictive. For example, it forbids the predicate $lseg$ and the pure method $lseg\_length$.

The list implementations shown in Figures 13, 17, and 19 have the same specification. As such, the implementations can be used interchangeably without having to reverify client code.

We define more flexible rules and explain how they affect the proofs of Section 4 to support the program of Figure 19. More specifically, we additionally allow self-recursion within pure functions and predicates under certain conditions. A pure function can recursively call itself, provided the recursion terminates. We enforce termination by generating additional proof obligations when verifying well-definedness of the return expression. In particular, we check at each recursive call that the size of the precondition's footprint decreases. For example, $lseg\_length$ terminates as the footprint for the recursive call does not include the permission to access **this**.$value$. When verifying whether the return expression is well-defined, we use Df' instead of Df to check the measure.

$$\begin{aligned}\mathsf{Df}'(p(x_1,\ldots,x_n)) &= \mathsf{Tr}(P)[y_1/x_1,\ldots,y_n/x_n] \wedge \neg(\mathsf{Ras}(P) \subseteq \mathsf{Ras}(P)[y_1/x_1,\ldots,y_n/x_n]) \\ &\quad \text{if } p \text{ is the method under verification} \\ \mathsf{Df}'(e) &= \mathsf{Df}(e).\end{aligned}$$

Here P is p's precondition. Df' enforces that the size of the precondition's footprint decreases by checking that there exists a memory location that is an element of the footprint of the caller's precondition but not in the footprint of the callee's precondition. A predicate can recursively mention itself, provided the first argument of the recursive call is a memory read $[x]$ and an access assertion $\mathbf{acc}(x)$ syntactically occurs to the left of the recursive call in a different separating conjunct. We take recursive pure functions and predicates into account in Section 4.2 as follows.

Definition 5.1. We define a partial order between tuples (H, γ, ϕ) where ϕ is an assertion or an expression, as the lexicographic ordering of the following three components.

—The footprint of ϕ under H and γ.
—The maximum of the program locations of the predicates and pure methods called in phi, or $\bot$ if none.
—The syntactic size of ϕ.

Note that this is a well-founded order. We perform syntactic checks and emit proof obligations such that all predicate and pure function call chains descend down this well-founded order. We define the meaning of expressions and assertions by induction on this order.

We also modify the definition of smaller (Definition 4.5) as follows.

Definition 5.2. An expression or assertion ϕ_1 is smaller than an expression or assertion ϕ_2 if for all heaps H and all environments γ, the tuple (H, γ, ϕ_1) is ranked below (H, γ, ϕ_2) in the partial order.

Proofs based on the size of an expression or assertion now use this novel definition.

Note that the rules just outlined can be generalized to other measures. Ensuring the consistency of the encoding of pure methods—in particular in the presence of dynamic binding—is an active area of research [Darvas and Leino 2007; Leino and

```
class Node {
  int total; Node left, right, parent; set<Node> nodes;

  predicate valid() = acc(total) * acc(left) * acc(right) * acc(parent);

  pure Node left()
    requires valid();
  { return left; }

  specpublic static predicate comp(set<Node> nodes, Node except) =
    (∀* n ∈ nodes • n.valid()) *
    (∀ n ∈ nodes•
      (n.left() = null ∨ (n.left() ∈ nodes ∧ n.left().parent() = n)) *
      (n.right() = null ∨ (n.left() ≠ null ∧ n.right() ∈ nodes ∧ n.right().parent() = n)) *
      (n.parent() = null ∨ (n.parent() ∈ nodes ∧ (n.parent().left() = n ∨ n.parent().right() = n))) *
      (n = except ∨ n.total() = 1 + (n.left() = null ? 0 : n.left().total())+
        (n.right() = null ? 0 : n.right().total())));

  specpublic predicate composite() = acc(nodes) ∧ this ∈ nodes ∧
    (∀ n ∈ nodes • acc(n.nodes) * n.nodes = nodes) *
    comp(nodes, null);
```

Fig. 21. The composite design pattern (part 1).

Middelkoop 2009; Rudich et al. 2008]. We specifically chose very simple rules in our formalization to be able to focus on proving soundness of the permission system and to avoid unnecessary clutter in the proof.

5.5. Composite

Composite data structures [Gamma et al. 1994, Chapter 4] typically represent part-whole hierarchies. As an example, consider the class *Node* of Figures 21 and 22. Each *Node* object has at most two children, referred to by *left* and *right*, and has a reference to its parent (except for the root). Moreover, each node has a field called *total*. The invariant that we wish to impose is that the *total* field of each node *n* holds the number of elements in the subtree rooted at *n*. However, enforcing this invariant is tricky, as modifications to a composite structure do not have to start at the root but can be applied directly to any node in tree. For example, the method *addLeft* adds a child to a node, but this node can be anywhere in the tree. As adding additional children to a node breaks the invariants of its ancestors, *addLeft* notifies the ancestors of the change via the method *fixTotal*. In particular, each ancestor can update its *total* field to reflect the addition of additional subnodes.

Specifying and verifying the composite design pattern was posed as a challenge for verification [Leavens et al. 2007], and was the topic of a special track at the 2008 Specification and Variational Component Boxed Systems (SAVCBS) workshop. The reason why the composite design pattern is challenging is that the owners-as-dominators policy imposed by several verification methodologies is in conflict with the fact that modifications can start at any node in the tree.

```
  void fixTotal(int c, set<Node> nodes)
    requires this ∈ nodes ∗ comp(nodes, this);
    requires total() + c = 1 + (left() = null ? 0 : left().total()) +
      (right() = null ? 0 : right().total());
    ensures comp(nodes, null);
    ensures ∀n ∈ nodes • n.right() = old(n.right()) ∧ n.left() = old(n.left()) ∧
      n.parent() = old(n.parent());
  { total += c; if(parent ≠ null) fixTotal(parent, nodes); }

  void addLeft(Node other)
    requires composite() ∗ other.composite();
    requires left() = null ∧ other.parent() = null;
    ensures composite() ∗ nodes = old(nodes ∪ other.nodes);
    ensures ∀n ∈ nodes • n.right() = old(n.right()) ∧ (n = this ∨ n.left() = old(n.left())) ∧
      (n = other ∨ n.parent() = old(n.parent()));
  {
    right := other; other.parent := this;
    foreach (n ∈ nodes ∪ other.nodes) { n.nodes := nodes ∪ other.nodes; }
    fixTotal(other.total, nodes);
  }
}
```

Fig. 22. The composite design pattern (part 2).

Several authors have proposed specifications for the composite design pattern [Rosenberg et al. 2010; Summers and Drossopoulou 2010; Jacobs et al. 2008]. Our solution is as follows. The core of our specification is the predicate *comp*. More specifically, $comp(s, e)$ holds if the nodes in set s represent a valid composite data structure. In particular, each node n in s must be valid and its *left*, *right*, and *parent* pointers must be bidirectional. Moreover, the invariant that $n.total()$ returns the number of nodes in the subtree rooted at n holds for each node n except for node e. This exception represents the insight that while an update is in progress, the invariant about total is broken only for a single node. For example, *fixTotal*'s precondition requires $comp(nodes, \textbf{this})$, indicating that the invariant about total holds for all nodes except for **this**. Note that when *except* is **null** (as in the predicate *composite*), then the invariant holds for all nodes.

Each node has a ghost field *nodes*. This ghost field is the set of *Node* objects reachable from **this** via *left*, *right*, and *parent* pointers. In other words, *nodes* is the set of all nodes in the same composite structure as **this**. The predicate $o.composite()$ then holds if (1) $o.nodes$ is accessible, (2) the *nodes* fields of all nodes in $o.nodes$ are accessible, and (3) the set $o.nodes$ represents a valid composite data structure without any exceptions (according to the predicate *comp*). The pre- and postcondition of *addLeft* use the predicate *composite*.

We consider both *comp* and *composite* to be public predicates (indicated by **specpublic**). This means the bodies of these predicates are visible to client code. Therefore, the invariant that $n.total()$ returns the total number of nodes in the subtree rooted at n is visible and can be used by clients. In other words, we consider this invariant to be part of the public specification. However, our specification still abstracts

over representation details of the composite data structure. In particular, the predicate *valid* hides the internals of each *Node* object. As a consequence, we change the internal representation of the class *Node* without endangering client code.

Note that the body of *comp* consists of an iterated star assertion (indicated by $\forall^*$). Contrary to an ordinary quantifier ($\forall$), the body of an iterated star is separately valid for each value in the set being quantified over. In our example, the iterated star describes not only that each node in *nodes* is valid, but also that the footprints of the *valid* predicates of all *Node* objects in the set are mutually disjoint. More formally, $\forall^* x \in s \bullet \alpha$ is encoded in the verification logic as follows.

$$\begin{gathered}\mathsf{Tr}(\forall^* x \in s \bullet \alpha) \equiv \\ (\forall x \in \mathsf{Tr}(s) \bullet \alpha) \wedge \\ (\forall x_1, x_2 \in \mathsf{Tr}(s) \bullet x_1 = x_2 \vee \mathsf{Ras}(\alpha)[x_1/x] \cap \mathsf{Ras}(\alpha)[x_2/x] = \emptyset).\end{gathered}$$

What is the required access set of an iterated star assertion? Informally, the required access set is the union of the required access sets of α for all elements in the set: $\bigcup_{x\ in \mathsf{Tr}(s)} \mathsf{Ras}(\alpha)$. However, $\bigcup$ is not a first-order concept. Inspired by the encoding of comprehensions by Leino and Monahan [2009], we encode the required access set of an iterated star as follows. For each iterated star in the program text, we extend the signature of the logic with a function symbol $union_i$ (where i is unique for each iterated star) with sort $heap \times set \times set \times int_1 \times \cdots \times int_k \rightarrow set$. 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 parameter is the set quantified over, and the remaining parameters are the free variables appearing in the body. Several axioms describe the behavior of $union_i$. For example, we add an axiom that states that a set is disjoint from a union only if it is disjoint from all the elements:

$$\begin{gathered}\forall h, a, s_1, x_1, \ldots, x_n, s_2 \bullet union_i(h, a, s_1, x_1, \ldots, x_n) \cap s_2 = \emptyset \\ \Leftrightarrow \\ (\forall x \in s_1 \bullet \mathsf{Ras}(\alpha) \cap s_2 = \emptyset).\end{gathered}$$

Whenever two different iterated stars are sufficiently similar, we generate only a single function, instead of two. Two iterated stars are sufficiently similar if they differ only in their quantified variable or in their range. Such similar iterated stars typically occur in loop invariants and in postconditions.

The composite design pattern can be regarded as a special case of the observer pattern. More specifically, one can consider each node as a subject where that node's parent is an observer. To ensure that the observer remains in sync with the subject (i.e., to ensure that the invariant about total is preserved), the subject must notify the observer when its state changes (i.e., when additional children are added). In our example, *addLeft* notifies the parent by calling *fixTotal*. As shown in Figure 25 in Section 6, we also verified the observer pattern using our verifier prototype. The implementation and specification of the observer pattern is available on our website (see Section 6).

5.6. Recell

Inheritance is a key component of the object-oriented programming paradigm that allows a class to be defined as an extension of one or more existing classes. For example, consider the class *Recell* from Figure 23. *Recell* extends its super class *Cell*. More specifically, a *Recell* object behaves exactly like a *Cell* but can additionally undo the last call to *setX*. This example is based on Parkinson and Bierman [2008, Figure 1].

Reasoning about inheritance in verification is challenging, since the code to be executed for a call is not determined at compile time but at runtime (depending on the

```
class Cell {
  int x;

  virtual predicate valid() =
    acc(x);

  pure virtual getX()
    requires valid();
  { return x; }

  Cell()
    ensures valid() * getX() = 0;
  { x := 0; }

  virtual void setX(int v)
    requires valid();
    ensures valid();
    ensures getX() = v;
  { x := v; }
}
```

(a)

```
class Recell : Cell {
  int bak;

  override predicate valid() =
    acc(bak) * base.valid();

  pure virtual GetBak()
    requires valid();
  { return bak; }

  Recell()
    ensures valid() * getX() = 0;
  { base(); }

  override void setX(int v)
    requires valid();
    ensures valid();
    ensures getX() = v ∧ GetBak() = bak;
  { bak := base.getX(); base.setX(v); }
}

  virtual void Undo()
    requires valid();
    ensures valid();
    ensures getX() = old(GetBak());
  { base.setX(bak); }
}
```

(b)

Fig. 23. The class *Cell* and its subclass *Recell*.

dynamic 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 informally explain how the approach described so far can be extended to support inheritance. Our treatment of inheritance is similar to earlier proposals by Parkinson and Bierman [2008], Leavens et al. [2007], and Jacobs and Piessens [2007].

Method calls can both be statically and dynamically bound, depending on the method itself and the calling context. For example, a call to *getX* inside client code is dynamically bound, while the call to *getX* in the body of *Recell.setX* is statically bound. In C#, a method invocation is dynamically bound only if the callee is marked **virtual** (or **override**) and the invocation is not a base call; otherwise, the invocation is statically bound. To distinguish statically bound calls from dynamically bound ones, we introduce an additional function (respectively, predicate) symbol for each virtual pure method (respectively, predicate). More specifically, for a pure method p defined in class C, the signature not only includes a symbol $C.p$ but also a symbol $C.p_D$. The former symbol is used to encode statically bound calls, while the latter symbol is used for dynamically bound calls. A similar, additional symbol is introduced for predicates.

The relation 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 equals C.

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

Note that type is a first-order function that represents the dynamic type of its argument. Furthermore, whenever a method $D.p$ overrides $C.p$, we add the following axiom to our theory.

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

This 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 and predicates whose receiver is syntactically equal to **this** are treated differently in method contracts and in code and annotations (other than function contracts). In normal code and non-contract annotations, such calls follow the standard rules that guide binding. However, if the this-call appears in the 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—including virtual methods—are verified under the assumption that the corresponding method is called statically. This assumption is sound, provided that in each subclass, each superclass method is either overridden or a default implementation with the same contract as the superclass method that simply calls the superclass method is generated (only for verification) and verified. Indeed, if the actual call is statically bound, then the contracts used by the caller and the callee match. 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 23, Recell overrides each virtual method except getX. If a class does not override a virtual method m, we generate a default method override which simply calls the superclass and inherits the super class contract as is. This method is subject to verification, just as any other method.

For verification to be modular, we must ensure that adding new subclasses cannot break existing code. We do so by checking that the contract of an overriding method is compatible [Parkinson and Bierman 2008] with the contract of the overridden method. More specifically, for each method $D.m$ overriding a method $C.m$, we verify that a dynamically bound call to $D.m$ satisfies the contract of $C.m$, assuming that the dynamic type of the receiver is D.

As argued by Parkinson and Bierman [2008], the rules for subclassing previously outlined generalize the standard subtyping restrictions. For example, consider the class DCell from Figure 24. DCell is a valid subclass of Cell, even though $\mathit{DCell.setX}$'s precondition places an additional restriction on the parameter v, and its postcondition does not imply the superclass postcondition. This kind of subclassing is safe, as $o.\mathit{valid}()$ never holds for an object with dynamic type DCell. That is, one can never pass an object with dynamic type DCell to a method that expects a valid instance of Cell. This example is based on Parkinson and Bierman [2008, Figure 5].

All examples used in Parkinson and Bierman [2008] (Cell, Recell, $\mathit{SubRecell}$, DCell, and TCell) have been verified using our verifier prototype. Moreover, the approach to inheritance just described also provides a solution to the challenges 8 and 9 on

```
class DCell : Cell {
  override predicate valid() = false;

  virtual predicate dvalid() = base.valid();

  pure override getX()
    requires dvalid();
  { return base.getX(); }

  DCell()
    ensures dvalid() * getX() = 0;
  { x := 0; }

  override void setX(int v)
    requires dvalid() * 0 ≤ v;
    ensures dvalid() * getX() = v × v;
  { x := v × v; }
}
```

Fig. 24. A subclass of *Cell* which is not a behavioral subtype.

reasoning about function objects described in Leavens et al. [2007]. The challenge examples 'Formatter' and 'TapeDrive' have been verified using our verifier prototype and are available online (see Section 6). Finally, note that higher-order predicates can be simulated via (ghost) interfaces. For example, consider the interface *Predicate* shown here.

interface *Predicate*<*T*> { **predicate** *apply*(*T o*); }.

An object that implements *Predicate* can be passed around as a higher-order predicate. For example, Figure 30 in Appendix C shows a *deep* ArrayList that not only keeps track of the elements themselves, but also of their validity.

6. EXPERIENCE

To show that the verification approach described in Section 3 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 Figure 25. The experiments were executed on a desktop computer with a Pentium Core Duo 2.66 Ghz processor and 4GB of memory running Windows Vista. To discharge the verification conditions, we used the Z3 [de Moura and Bjørner 2008] theorem prover (version 2.19). The verifier itself and the programs shown in Figure 25 can be downloaded from `http://people.cs.kuleuven.be/jan.smans/vericool3`.

The experiments in Figure 25 demonstrate that the verification conditions in the implicit dynamic frames approach can be discharged automatically by a standard first-order theorem prover. However, for some programs, the performance was less than stellar. For example, the Visitor pattern—a fairly complex but small program—takes over six minutes. Moreover, our experiments have also shown that verification times can be sensitive to small changes in the input. For example, swapping two statements can mean the difference between 10 and 100 seconds. In our experience, these problems are not unique to the approach proposed in this article but seem to be inherent to verification condition generation-based approaches that encode the heap

program	time taken	source
Cell	0.2	[Parkinson and Bierman 2005] [Distefano and Parkinson 2008] [Jacobs and Piessens 2007]
ArrayList	0.5	[Smans et al. 2008]
Stack	0.7	[Smans et al. 2008]
Iterator	42	[Kassios 2006]
LinkedList	96 (ghost fields) (rec. defs.)	[Reynolds 2002] [Kassios 2006] [Leino and Müller 2004]
Resource Pool	25	[Distefano and Parkinson 2008]
Marriage	0.3	[Leino and Müller 2004]
MasterClock	0.4	[Barnett and Naumann 2004]
Subject-Observer	248	[Leino and Schulte 2007] [Middelkoop et al. 2008] [Parkinson 2007]
Composite	4	[Leavens et al. 2007] [Rosenberg et al. 2010] [Jacobs et al. 2008]
Recell, TCell, DCell, SubRecell	37	[Parkinson and Bierman 2008]
Visitor (framing only)	379	[Distefano and Parkinson 2008]
Formatter	0.2	[Leavens et al. 2007]
TapeDrive	0.5	[Leavens et al. 2007]

Fig. 25. Table showing the time taken (in seconds) to verify each program.

as an updatable map in general. An encouraging observation in this respect is that ongoing theorem-proving research continues to yield more powerful and more efficient theorem provers. An interesting research question for our approach is whether the verification conditions—in particular the encoding of the frame properties—can be made more efficient to alleviate these disadvantages.

7. RELATED WORK

Separation Logic. Our approach was heavily influenced by separation logic [O'Hearn et al. 2001; Reynolds 2002; Parkinson and Bierman 2005; Parkinson and Bierman 2008]. In particular, the idea of using permissions to infer an upper bound on the set of memory locations modifiable and/or readable by a method is based on separation logic's frame rule. Our access assertion $\mathbf{acc}(e.f)$ corresponds to the separation logic points-to assertion $e.f \mapsto \_$, and Parkinson and Bierman's abstract predicates inspired our predicates. As an example, consider the class *Cell* annotated with separation logic specifications of Figure 26. The behavior of *Cell*'s methods is specified via the predicate *cell*. However, while the implicit dynamic frames approach uses pure methods to describe the state of an object, separation logic uses predicate parameters instead. In general, implicit dynamic frames combines ideas from conventional verification approaches with separation logic. That is, in many conventional verification approaches such as JML [Burdy et al. 2005], Eiffel [EIFFEL 2006], Code Contracts [Barnett et al. 2010] and Spec# [Barnett et al. 2004], an assertion is (more or less) a boolean expression from the host programming language. In particular, assertions can mention heap-dependent expressions, such as method calls, pointer dereferences, and old expressions. The advantages of these approaches are (1) that the syntax and operational semantics of assertions is easy to explain to developers, as they are already familiar with the host programming language, and (2) that these specifications are typically amenable to runtime checking. Experience has shown, however, that statically checking conformance with a conventional specification is challenging. In

```
predicate cell(Cell c, int v) = c.x ↦ v;

class Cell {
  int x;

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

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

  void setX(int v)
    requires cell(this, _);
    ensures cell(this, v);
  { x := 0; }
}
```

Fig. 26. The class *Cell* with separation logic annotations.

particular, framing is typically a thorny issue. Separation logic has proven to be a powerful alternative to these conventional approaches. For example, separation logic has been used to statically verify correctness of algorithms and data structures, such as composite [Jacobs et al. 2008] and visitor [Distefano and Parkinson 2008], considered to be challenging for conventional approaches. We consider implicit dynamic frames to combine the best of both worlds: the assertion syntax is close to the host programming language, and its permissions provide an elegant, yet flexible solution to the frame problem.

Berdine et al. [2005] demonstrate that a fragment of separation logic is amenable to automatic static checking by building a verifier, called Smallfoot, for a small, procedural language. Smallfoot uses symbolic execution instead of verification condition generation for checking that a program satisfies its specification. The symbolic state maintained during symbolic execution consists of a number of pure formulae, describing constraints between variables, and a number spatial formulae, describing which locations are accessible and what values they hold. Smallfoot frames method invocations by performing *frame inference*. More specifically, when verifying a call, the spatial formulae corresponding to the callee's precondition are removed from the symbolic heap. The chunks that remain are called the frame. Afterwards, the chunks corresponding to the postcondition are added to this frame. In implicit dynamic frames, the effect of method calls—and the corresponding permission transfer—is encoded by assigning fresh values to (sometimes called *havocking*) the heap and access set. Framing relies on the method's free postconditions and on the axioms generated for predicates and pure methods. The verification condition generation-based verification approach described in Section 3 can be considered to be an alternative way of proving that a program satisfies a separation logic-like specification. In particular, the generated verification conditions are first-order formulas that can be discharged by off-the-shelf theorem provers such as Z3 [de Moura and Bjørner 2008].

jStar [Distefano and Parkinson 2008] and VeriFast [Jacobs et al. 2010] extend the basic ideas of Berdine et al. [2005] to full-fledged programming languages like Java and C. However, neither Smallfoot, jStar, nor VeriFast support non-separating conjunction between spatial assertions. As explained in Section 3, our approach does support such

non-separating conjunction. For example, consider the code snippet shown here.

$$
\begin{array}{l}
\mathit{ArrayList}\ l := \mathit{new}\ \mathit{ArrayList}(); \\
\mathit{Iterator}\ \mathit{iter} := \textbf{new}\ \mathit{Iterator}(l); \\
\textbf{assert}\ \mathit{iter.valid}() \wedge l.\mathit{valid}(); \\
\textbf{int tmp} := l.\mathit{size}(); \\
l.\mathit{add}(\textbf{null});
\end{array}
$$

On line 3, the footprint of the list and the iterator overlap, but both objects are valid. As such, our tool can prove that the assertion on line 3 holds. This assertion cannot be not expressed (without introducing additional helper predicates) in Berdine et al. [2005], Distefano and Parkinson [2008], and Jacobs et al. [2010]. Because of the non-separating conjunction, we can immediately call pure methods on the list (as on line 4) without having to rewrite the symbolic state. Calling pure methods does not break the assertion. Similarly, we can immediately call mutator methods (as on line 5) without having to perform any rewriting, yet doing so invalidates the iterator (as expected).

VeriFast [Jacobs et al. 2010] requires developers to explicitly fold and unfold predicates via ghost statements. Such manual folding and unfolding of predicates is not required in our approach. In particular, the theorem prover can apply the implementation axiom to fold and unfold predicates whenever required during the proof. Smallfoot [Berdine et al. 2005] does not require developers to perform explicit folding and unfolding, but it only supports a fixed set of built-in predicates. jStar [Distefano and Parkinson 2008] rewrites the symbolic state automatically, as required during the proof, based on rules that can be specified by developers in a separate file.

Our approach—and in our experience other verification condition generation-based approaches—has three disadvantages with respect to Berdine et al. [2005], Jacobs et al. [2010], and Distefano and Parkinson [2008]. First of all, discharging the verification condition can take a long time, even for relatively short methods. We believe these long verification times are mainly caused by the extensive use of quantifiers needed to encode frame conditions. Secondly, verification condition generation tends to be unpredictable: small changes in the input can have significant impact on verification time. Finally, it can be hard to determine why verification fails. That is, the theorem prover typically only reports the error location (e.g., "assertion might not hold") but offers no help in finding the cause of the error. In particular, there is no symbolic execution trace that can be inspected to diagnose the error. For example, VeriFast supports a symbolic debugger that can be used to inspect the symbolic states on the path to the error. In recent work, Müller and Ruskiewicz [2011] try to remedy the last disadvantage by generating inputs suitable for debugging based on the counter example generated by the theorem prover. Similarly, the VCC model viewer [Dahlweid et al. 2009] allows developers to inspect the generated counter example to diagnose verification errors.

In addition to jStar and VeriFast, Smallfoot has inspired several other tools. For example, Heap-Hop [Villard et al. 2009] is an extension of Smallfoot targeted at proving memory safety and deadlock freedom of concurrent programs that rely on message-passing. HIP [Nguyen et al. 2007] is a variant of Smallfoot that focuses on automatically proving size properties (in addition to shape properties). Tuerk [2009] has developed HOLFoot, a port Smallfoot to HOL. He has mechanically proven that HOLFoot is sound. An interesting feature of HOLFoot is that one can resort to interactive proofs in HOL4 when the tool is unable to construct a proof automatically.

As it is hard for fully automatic tools to prove full functional correctness, several researchers [Nanevski et al. 2008; Hobor et al. 2008; Tuch et al. 2007] have used separation logic within interactive proof assistents. While this approach typically requires more input from the developer, it has resulted in a number of impressive achievements.

For example, Tuch et al. [2007] report on verifying the L4 microkernel. Most embeddings of separation logic in proof assistents focus on the *-fragment of separation logic (i.e., without non-separating conjunction). However, Nanevski et al. [2010] show that it is possible to reason about non-separating conjunction in these interactive provers by explicitly reasoning about disjointness (as we do).

Separation logic-based shape analysis tools such as Space Invader [Calcagno et al. 2009] and Xisa [Rival and Chang 2011] show great promise. For example, Space Invader has proven memory safety of 58.4 percent of the procedures in the Linux kernel. The main advantage of these static analyzers is that no annotations are required, as they perform a non-modular, interprocedural analysis. The focus of these tools is proving memory safety instead of full functional correctness.

Van Staden et al. [2010] also recognize the gap between executable specifications (boolean expressions of the host programming language) on the one hand and separation logic assertions on the other. However, instead of adding heap-dependent expressions to separation logic, they propose proving that each executable specification will evaluate to true, based on a separation logic specification.

The access set tracked in our verification approach resembles the coloring of objects used for runtime checking of separation logic specification in SLICK [Nguyen et al. 2008].

Dynamic Frames and Regional Logic. The dynamic frames approach [Kassios 2006; Banerjee et al. 2008; Smans et al. 2008; Leino 2010; Schoeller 2007] 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*, pure methods, or ghost fields that denote sets of locations. Dynamic frames correspond to the predicate footprints of our approach. As an example, consider the dynamic frames version of the class $Cell$ of Figure 27. $setX$'s modifies clause indicates that the method can potentially modify all locations in the dynamic frame $footprint()$. Moreover, $setX$'s last postcondition encodes the swinging pivot property. The contract of each pure method includes a reads clause, indicating that its return value depends only on locations in $footprint()$. All the latter effect annotations (highlighted with a yellow 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 yellow 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 allows us to infer 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 the class $ArrayList$ of Figure 13, 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.

In implicit dynamic frames and separation logic, modifies and reads clauses can be inferred, provided the verifier performs an access check at each field access. When verifying concurrent programs, the latter check must be performed anyhow to guarantee the absence of data races. A modifies clause would only duplicate the information that can already be inferred from the method precondition. In other words, framing comes for free in concurrent programs and—in our opinion—there's no reason to keep using modifies clauses.

```
class Cell {
  int x;

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

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

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

  Cell()
    modifies ∅;
    ensures valid() ∧ getX() = v;
  { x := 0; }

  void setX(int v)
    requires valid();
    modifies footprint();
    ensures valid() ∧ getX() = v;
    ensure fresh(footprint() \ old(footprint()));
  { x := 0; }
}
```

Fig. 27. The class *Cell* with dynamic frames annotations. Annotations highlighted must be provided a checked explicitly in the dynamic frames approach.

Implicit Dynamic Frames. This article improves upon and extends our earlier work presented at the European Conference on Object-Oriented Programming (ECOOP) [Smans et al. 2009] in three main ways. First, we have applied implicit dynamic frames to several additional example programs (e.g., the composite pattern). We describe the specification of these programs in detail in Section 5 and report on new experiments with our verifier prototype in Section 6. Second, we provide a simple soundness proof. Third, we have thoroughly revised and updated our comparison with related work.

The implicit dynamic frames approach has already sparked the interest of other authors. For example, Leino et al. [Leino and Müller 2009; Leino et al. 2010] extend our approach with fractions and support for concurrency. They have implemented their variant of implicit dynamic frames in a verifier called Chalice. Similar to the access set a used in our verification conditions, they use a permission mask $\mathcal{P}$ that maps addresses to permissions. However, they encode permission transfer in a slightly different way. More specifically, when assuming (*inhaling* in their terminology) or checking (*exhaling* in their terminology) an assertion, they transfer one subassertion at a time, while we compute the required access set and transfer all permissions in one go. Their encoding makes it more difficult to encode predicates and pure methods as functions in the

verification logic, where folding and unfolding can be done automatically via axioms. At the time of writing, the current approach for predicates and pure methods in Chalice is still in flux. Moreover, they do not support non-separating conjunction between spatial assertions. Finally, they provide no formal soundness proof.

Parkinson and Summers [2011] have investigated the relationship between separation logic and implicit dynamic frames. One of the main conclusions of their investigation is that (a subset of) separation logic specifications can be checked by translating them to implicit dynamic frames.

Smans et al. [2010] show that implicit dynamic frames specifications can be checked more efficiently via a symbolic execution-based verification algorithm (as in Smallfoot [Berdine et al. 2005]) instead of via verification condition generation. However, predicate definitions must be folded and unfolded explicitly, only separating conjunction is supported (normal conjunction is not), and it is not clear if the symbolic execution-based approach can handle the more complex examples, such as the composite pattern that can be verified via verification condition generation.

Other Approaches. Reasoning about method calls in specifications—and in particular, framing their return values—was posed as a challenge for verification by Leavens et al. [2007]. Various researchers have attacked well-formedness of pure method specifications [Leino and Middelkoop 2009; Rudich et al. 2008], framing of return values [Darvas and Leino 2007; Jacobs and Piessens 2007; Smans et al. 2008], and allowing certain side effects in pure methods [Darvas and Leino 2007]. Just like most of these researchers, we encode pure methods as functions in the verification logic. The contribution of this article (with respect to existing approaches for dealing with pure methods) is the insight that a pure method's return value can be framed by inspecting the permissions required by its precondition.

In Leino et al. [2002], 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 Leino et al. [2002] cannot. For example, the former restriction rules out sharing of representation objects, as is the case in the iterator pattern.

In the universe type system [Müller 2001] and the Spec# methodology [Barnett et al. 2003; Leino and Müller 2004], abstractions (pure methods, invariants, or model fields) can depend on the fields of owned objects and the fields of peers (i.e., objects with the same owner as the receiver), provided the abstraction is visible to the peer. For example, the method $hasNext$ of an iterator would have to be visible to the list class. Our approach has no such restriction. In general, the Boogie methodology features more rules and methodological machinery than our approach. In particular, the concepts object invariant and owned object (rep-object in their terminology) are built-in. As a consequence, the Boogie methodology is more complex and less expressive than implicit dynamic frames. Jacobs et al. [2008] extend the basic Boogie methodology to concurrent programs. Just like us, they infer framing information from method preconditions. More specifically, their verification conditions keep track of a set of accessible locations A. As in our approach, a callee cannot modify an existing locations if that location is retained by the caller during the call. However, Jacobs et al. [2008] inherits the disadvantages of the underlying Boogie methodology.

Verification of Java-programs with JML-like [Burdy et al. 2005] annotations has received considerable attention in the research community [Burdy et al. 2005; Beckert et al. 2007; Flanagan et al. 2002]. 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.

8. CONCLUSION

Implicit dynamic frames is a marriage between conventional specification formalisms and separation logic. Like most conventional verifiers but contrary to separation logic-based ones, conformance with an implicit dynamic frames specification can be mechanically checked via verification condition generation. These verification conditions can be discharged by standard first-order provers. Unlike most separation logic-based provers, we support heap-dependent expressions and non-separating conjunction in assertions. Just as in separation logic, effect annotations must not be provided explicitly by developers but can, instead, be inferred from the permissions required by method preconditions. We have implemented our approach in a verifier prototype and have used this prototype to prove correctness of a number of programming patterns considered to be challenging in related work. Finally, the approach has been proven sound.

In both separation logic and implicit dynamic frames, a memory location is either writable, readable, or not accessible at all. However, in some programs—in particular concurrent ones—more sophisticated permission policies are required. As shown by Owicki and Gries [1976] and more recently by Jacobs and Piessens [2011], arbitrarily complex policies can be expressed by using ghost state and fractional permissions. However, using ghost state does not lead to intuitive proofs. We consider designing a verification approach that supports flexible, user-defined permissions one of the main challenges for verification. Separation logic and implicit dynamic frames would both be limited, special cases of this more general approach. Concurrent abstract predicates [Dinsdale-Young et al. 2010] form a promising direction in this area.

APPENDIX

A. A BAD IMPLEMENTATION OF CELL

```
class Cell {
  static Cell global;
  int x;

  Cell() {
    if(global = null) { global := this; }
    x := 0;
  }

  int getX()
  { return x; }

  void setX(int v) {
    if(global ≠ this) { global.x :=v; }
    x := v;
  }
}
```

Fig. 28. An implementation of class *Cell* which satisfies the contracts of Figure 1 but causes the assertion in Figure 2(b) to fail.

B. ENCODING THE CELL EXAMPLE

```
module {
  predicate cell(c) = acc(c);

  pure func cell_get(c)
    requires cell(c);
    ensures true;
  { return [c]; }

  func cell_create(res)
    requires acc(res);
    ensures acc(res) ∗ cell([res]) ∧ cell_get([res]) = 0;
  { tmp := cons(0); [res] := tmp; }

  func cell_set(c, v)
    requires cell(c);
    ensures cell(c) ∧ cell_get(c) = v;
  { [c] := v; }

  func cell_dispose(c)
    requires cell(c);
    ensures true;
  { free(c); }
}

module {
  func main()
    requires true;
    ensures true;
  {
    res := cons(0);
    cell_create(res); c1 := [res]; cell_set(c1, 5);
    cell_create(res); c2 := [res]; cell_set(c2, 10);
    assert cell_get(c1) = 5;
    cell_dispose(c1); cell_dispose(c2); free(res);
  }
}
```

Fig. 29. An encoding of the program of Figure 4 in the small, imperative language of Figure 5. **true** is syntactic sugar for the assertion $0 = 0$. Return values of (nonpure) functions are passed explicitly via the heap (at the address stored in the parameter *res*).

C. SIMULATING HIGHER-ORDER PREDICATES

```
class ArrayList<T> {
  int size; T[] items; Predicate<T> P;

  predicate valid() = acc(size) * acc(items) * acc(P) * P ≠ null *
    items ≠ null * acc(items.elems) * 0 ≤ size * size ≤ items.Length *
    (∀* i ∈ (0 : size) • P.apply(items[i]));

  pure Predicate<T> getP()
    requires valid(); ensures result ≠ null;

  pure int size()
    requires valid(); ensures 0 ≤ result;

  pure int get(int index)
    requires valid() * 0 ≤ index * index < size(); ensures getP().apply(result);

  ArrayList( Predicate<T> P )
    requires P ≠ null;
    ensures valid() * getP() = P;

  void add(T o)
    requires valid() * getP().apply(o);
    ensures valid();
    ensures ...;

  T remove(int index)
    requires valid() * 0 ≤ index * index < size();
    ensures valid() * getP().apply(result);
    ensures ...;
}
```

Fig. 30. A *deep ArrayList* that not only stores the elements themselves but also keeps track of their validity. The desired validity condition can be selected on a per-instance basis, as each *ArrayList* instance is parametrized with a predicate P. The implementation is omitted, as it is equal to the one shown in Figure 13.

REFERENCES

Banerjee, A., Naumann, D., and Rosenberg, S. 2008. Regional logic for local reasoning about global invariants. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*.

Barnett, M., DeLine, R., Fähndrich, M., Leino, K. R. M., and Schulte, W. 2003. Verification of object-oriented programs with invariants. *J. Obj. Technol. 3,* 6.

Barnett, M., Fahndrich, M., and Logozzo, F. 2010. Embedded contract languages. In *Proceedings of the ACM Symposium on Applied Computing (SAC)*.

Barnett, M., Leino, K. R. M., and Schulte, W. 2004. The Spec# programming system: An overview. In *Proceedings of the Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart devices (CASSIS)*.

Barnett, M. and Naumann, D. A. 2004. Friends need a bit more: Maintaining invariants over shared state. In *Proceedings of the International Conference on Mathematics of Program Construction (MPC)*.

Beckert, B., Hähnle, R., and Schmitt, P. H. 2007. *Verification of Object-Oriented Software: The KeY Approach*. Springer-Verlag.

Berdine, J., Calcagno, C., and O'Hearn, P. 2005. Symbolic execution with separation logic. In *Proceedings of the Asian Symposium on Programming Languages and Systems (APLAS)*.

Bornat, R. 2000. Proving pointer programs in hoare logic. In *Proceedings of the International Conference on Mathematics of Program Construction (MPC)*.

Boyland, J. 2003. Checking interference with fractional permissions. In *Proceedings of the International Symposium on Static Analysis*.

Burdy, L., Cheon, Y., Cok, D., Ernst, M., Kiniry, J., Leavens, G. T., Leino, K. R. M., and Poll, E. 2005. An overview of JML tools and applications. *Int. J. Softw. Tools Technol. Transfer 7,* 3.

Calcagno, C., Distefano, D., OHearn, P., and Yang, H. 2009. Compositional shape analysis by means of BI-abduction. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*.

Dahlweid, M., Moskal, M., Santen, T., Tobies, S., and Schulte, W. 2009. VCC: Contract-based modular verification of concurrent C. In *Proceedings of the International Conference on Software Engineering (ICSE)*.

Darvas, Á. and Leino, K. R. M. 2007. Practical reasoning about invocations and implementations of pure methods. In *Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE)*.

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

Detlefs, D., Nelson, G., and Saxe, J. B. 2005. Simplify: A theorem prover for program checking. *J. ACM 52,* 3.

Dijkstra, E. W. 1975. Guarded commands, nondeterminacy and formal derivation of programs. *Comm. ACM 18,* 8.

Dinsdale-Young, T., Dodds, M., Gardner, P., Parkinson, M., and Vafeiadis, V. 2010. Concurrent abstract predicates. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*.

Distefano, D. and Parkinson, M. 2008. jStar: Towards practical verification for Java. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*.

EIFFEL 2006. Eiffel: Analysis, design and programming language. Standard ECMA-367, ECMA International.

Flanagan, C., Leino, K. R. M., Lillibridge, M., Nelson, G., Saxe, J. B., and Stata, R. 2002. Extended static checking for Java. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Haack, C. and Hurlin, C. 2009. Resource usage protocols for iterators. *J. Obj. Technol. 8,* 4.

Hobor, A., Appel, A. W., and Nardelli, F. Z. 2008. Oracle semantics for concurrent separation logic. In *Proceedings of the European Symposium on Programming (ESOP)*.

Jacobs, B. and Piessens, F. 2007. Inspector methods for state abstraction. *J. Obj. Technol. 6,* 5.

Jacobs, B. and Piessens, F. 2011. Expressive modular fine-grained concurrency specification. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*.

Jacobs, B., Piessens, F., Smans, J., and Leino, K. R. M. 2008. A programming model for concurrent object-oriented programs. *ACM Trans. Program. Lang. Syst. 31,* 1.

Jacobs, B., Smans, J., and Piessens, F. 2008. Verifying the composite pattern using separation logic. In *Proceedings of the Specification and Verification of Component-Based Systems—Challenge Track (SAVCBS)*.

Jacobs, B., Smans, J., and Piessens, F. 2010. A quick tour of the VeriFast program verifier. In *Proceedings of the Asian Symposium on Programming Languages and Systems (APLAS)*.

Kassios, Y. 2006. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In *Proceedings of the International Symposium on Formal Methods (FM)*.

Krishnaswami, N. R. 2006. Reasoning about iterators with separation logic. In *Proceedings of the Specification and Verification of Component-Based Systems (SAVCBS)*.

Leavens, G. T. 2006. JMLŠs rich, inherited specifications for behavioral subtypes. In *Proceedings of the International Conference on Formal Engineering Methods (ICFEM)*.

Leavens, G. T., Leino, K. R. M., and Müller, P. 2007. Specification and verification challenges for sequential object-oriented programs. *Formal Aspects Comput. 19,* 2.

LEINO, K. R. M. 2010. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR)*.

LEINO, K. R. M. AND MIDDELKOOP, R. 2009. Proving consistency of pure methods and model fields. In *Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE)*.

LEINO, K. R. M. AND MONAHAN, R. 2009. Reasoning about comprehensions with first-order smt solvers. In *Proceedings of the ACM Symposium on Applied Computing (SAC)*.

LEINO, K. R. M. AND MÜLLER, P. 2004. Object invariants in dynamic contexts. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*.

LEINO, K. R. M. AND MÜLLER, P. 2009. A basis for verifying multithreaded programs. In *Proceedings of the European Symposium on Programming (ESOP)*.

LEINO, K. R. M., MÜLLER, P., AND SMANS, J. 2010. A basis for verifying multithreaded programs. In *Proceedings of the European Symposium on Programming (ESOP)*.

LEINO, K. R. M. AND NELSON, G. 2002. Data abstraction and information hiding. *ACM Trans. Program. Lang. Sys. 24,* 5.

LEINO, K. R. M., POETZSCH-HEFFTER, A., AND ZHOU, Y. 2002. Using data groups to specify and check side effects. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*.

LEINO, K. R. M. AND SCHULTE, W. 2007. Using history invariants to verify observers. In *Proceedings of the European Symposium on Programming (ESOP)*.

MALECHA, G. AND MORRISETT, G. 2010. Mechanized verification with sharing. In *Proceedings of the International Colloquium on Theoretical Aspects of Computing (ICTAC)*.

MIDDELKOOP, R., HUIZING, C., KUIPER, R., AND LUIT, E. J. 2008. nvariants for non-hierarchical object structures. In *Proceedings of the Brazilian Symposium on Formal Methods (SBMF)*.

MÜLLER, P. 2001. Modular specification and verification of object-oriented programs. Ph.D. dissertation, FernUniversität Hagen.

MÜLLER, P. AND RUSKIEWICZ, J. N. 2011. Using debuggers to understand failed verification attempts. In *Proceedings of the International Symposium on Formal Methods (FM)*.

NANEVSKI, A., MORRISETT, G., SHINNAR, A., GOVEREAU, P., AND BIRKEDAL, L. 2008. Ynot: Reasoning with the awkward squad. In *Proceedings of the International Conference on Functional Programming (ICFP)*.

NANEVSKI, A., VAFEIADIS, V., AND BERDINE, J. 2010. Structuring the verification of heap-manipulating programs. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*.

NGUYEN, H. H., DAVID, C., QIN, S., AND CHIN, W.-N. 2007. Automated verification of shape and size properties via separation logic. In *Proceedings of the Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*.

NGUYEN, H. H., KUNCAK, V., AND CHIN, W.-N. 2008. Runtime checking for separation logic. In *Proceedings of the Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*.

O'HEARN, P., REYNOLDS, J., AND YANG, H. 2001. Local reasoning about programs that alter data structures. In *Proceedings of the International Workshop on Computer Science Logic (CSL)*.

OWICKI, S. AND GRIES, D. 1976. Verifying properties of parallel programs: an axiomatic approach. *Comm. ACM 19,* 5.

PARIENTE, D. AND LEDINOT, E. 2010. Formal verification of industrial C code using Frama-C: A case study. In *Proceedings of the International Conference on Formal Verification of Object-Oriented Software (FoVeOOS)*.

PARKINSON, M. 2005. Local reasoning for Java. Ph.D. thesis, University of Cambridge.

PARKINSON, M. 2007. Class invariants: The end of the road? In *Proceedings of the International Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming (IWACO)*.

PARKINSON, M. AND BIERMAN, G. 2005. Separation logic and abstraction. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*.

PARKINSON, M. AND BIERMAN, G. 2008. Separation logic, abstraction and inheritance. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*.

PARKINSON, M. AND SUMMERS, A. 2011. The relationship between separation logic and implicit dynamic frames. In *Proceedings of the European Symposium on Programming (ESOP)*.

REYNOLDS, J. 2002. Separation logic: A logic for shared mutable data structures. In *Proceedings of the Symposium on Logic in Computer Science (LICS)*.

RIVAL, X. AND CHANG, B.-Y. E. 2011. Calling context abstraction with shapes. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*.

ROSENBERG, S., BANERJEE, A., AND NAUMANN, D. A. 2010. Local reasoning and dynamic framing for the composite pattern and its clients. In *Proceedings of the International Conference on Verified Software: Theories, Tools and Experiments*.

RUDICH, A., DARVAS, Á., AND MÜLLER, P. 2008. Checking well-formedness of pure-method specifications. In *Proceedings of the International Symposium on Formal Methods (FM)*.

SCHOELLER, B. 2007. Making classes provable through contracts. Ph.D. ETH Zurich.

SMANS, J., JACOBS, B., AND PIESSENS, F. 2009. Implicit dynamic frames: Combining dynamic frames and separation logic. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*.

SMANS, J., JACOBS, B., AND PIESSENS, F. 2010. Heap-dependent expressions in separation logic. In *Proceedings of the International Conference on Formal Techniques for Distributed Systems (FMOODS/FORTE)*.

SMANS, J., JACOBS, B., PIESSENS, F., AND SCHULTE, W. 2008. An automatic verifier for Java-like programs based on dynamic frames. In *Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE)*.

SUMMERS, A. J. AND DROSSOPOULOU, S. 2010. Considerate reasoning and the composite design pattern. In *Proceedings of the Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*.

TUCH, H., KLEIN, G., AND NORRISH, M. 2007. Types, bytes, and separation logic. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*.

TUERK, T. 2009. A formalization of Smallfoot in HOL. In *Proceedings of the International Conference on Theorem Proving in Higher-Order Logics (TPHOLs)*.

TUERK, T. 2010. Local reasoning about while-loops. In *Proceedings of the International Conference on Verified Software: Theories, Tools and Experiments—Theory Workshop (VS-Theory)*.

VAN STADEN, S., CALCAGNO, C., AND MEYER, B. 2010. Verifying executable object-oriented specifications with separation logic. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*.

VILLARD, J., LOZES, É., AND CALCAGNO, C. 2009. Proving copyless message passing. In *Proceedings of the Asian Symposium on Programming Languages and Systems (APLAS)*.

ZEE, K., KUNCAK, V., AND RINARD, M. C. 2008. Full functional verification of linked data structures. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*.

Received July 2011; accepted January 2012