Verifying object-oriented programs with higher-order separation logic in Coq

Jesper Bengtson, Jonas Braband Jensen, Filip Sieczkowski, and Lars Birkedal

IT University of Copenhagen

Abstract. We present a shallow Coq embedding of a higher-order separation logic with nested triples for an object-oriented programming language. Moreover, we develop novel specification and proof patterns for reasoning in higher-order separation logic with nested triples about programs that use interfaces and interface inheritance. In particular, we show how to use the higher-order features of the Coq formalisation to specify and reason modularly about programs that (1) depend on some unknown code satisfying a specification or that (2) return objects conforming to a certain specification. All of our results have been formally verified in the interactive theorem prover Coq.

1 Introduction

Separation Logic [12,16] is a Hoare-style program logic for modular reasoning about programs that use shared mutable data structures. *Higher-order* separation logic [3] (HOSL) is an extension of separation logic that allows for quantification over predicates in both the assertion logic (the logic of pre- and post-conditions) and the specification logic (the logic of Hoare triples). HOSL was proposed with the purposes of (1) reasoning about data abstraction via quantification over resource invariants, and (2) making formalisations of separation logic easier by having one general expressive logic in which it is possible to define predicates, etc., needed for applications. In this article we explore these two purposes further; we discuss each in turn.

The first purpose (data abstraction) has been explored for a first-order language [4], for higher-order languages [9,11], and for reasoning about generics and delegates in object-oriented languages (without interfaces and without inheritance) [18]. In this article we show how HOSL can be used for modular reasoning about interfaces and interface-based inheritance in an object-oriented language like Java or $C\sharp$. Our current work is part of a research project in which we aim to formally specify and verify the C5 generic collection library [8], which is an extensive collection library that is used widely in practice and whose implementation makes extensive use of shared mutable data structures. A first case-study of one of the C5 data structures is described in [7]. C5 is written in $C\sharp$ and is designed mainly using interface inheritance, rather than class-to-class inheritance; different collection modules are related via an inheritance hierarchy among interfaces. For this reason we focus on verifying object-oriented programs that use interfaces and interface-based inheritance.

We explore the second purpose (formalisation) by developing a Coq formalisation of HOSL for an object-oriented class-based language and show through verified examples how it can be used to reason about interfaces and inheritance.

Our formalisation makes use of ideas from abstract separation logic [6] and thus consists of a general treatment of the assertion logic that works for many models and for a general operationally-inspired notion of semantic command. Our general treatment of the logic is also rich enough to cover so-called nested triples [17], which are useful for reasoning about unknown code, either in the form of closures or delegates [18] or, as we show here, in the form of code matching an interface. To reason about object-oriented programs, we instantiate the general development with the heap model for our object-oriented language and derive suitable proof rules for the language. This approach makes it easier in the future to experiment with other storage models and languages, e.g., variants of separation logic with fractional permissions.

Summary of contributions. We formalize a shallow Coq embedding of a higher-order separation logic for an object-oriented programming language. We have designed a system that allows us to write programs together with their specifications, and then prove that each program conforms to its specification. All meta-theoretical results have been verified in Coq[1].

We introduce a pattern for interface specifications that allows for a modular design. An interface specification is parametrised in such a way that any class implementing the interface can be given a suitably expressive specification by a simple instantiation of the interface specification. Moreover, we show how to use nested triples to, e.g., write postconditions in the assertion logic that require a returned object to match a certain specification. Our approach enables us to verify dynamically dispatched method calls, where the dynamic types of the objects are unknown.

Outline. The rest of this article is structured as follows. In Section 2 we demonstrate the patterns we use for writing interfaces by providing a small example program that uses interface inheritance and proving that it conforms to its specification. In Section 3 we cover the language and memory-model independent kernel of our Coq formalisation. In Section 4 we specialise our system to handle Java-like programs by providing constructs and a suitable memory model for a subset of Java. Section 5 covers related work, and Section 6 concludes.

2 Reasoning with interfaces

To demonstrate how our logic is applied, we will use the example of a class Cell that stores a single value and which is extended by a subclass Recell that maintains a backup of the last overwritten value and has an undo operation. This example is originally due to Abadi and Cardelli [1]; a variant of it was also used

[1] The Coq development accompanying this article can be found at `http://itu.dk/people/birkedal/papers/hosl_coq-201105.tar.gz`

```
interface ICell {
  int get();
  void set(int v);
}

class ProxySet {
  static void proxySet(ICell c, int v) {
    c.set(v);
  }
}

class Cell implements ICell {
  int value;

  Cell() { }
  int get() {
    return this.value;
  }
  void set(int v) {
    this.value = v;
  }
}
```

```
interface IRecell extends ICell {
  void undo();
}

class Recell implements IRecell {
  Cell cell;
  int bak;

  Recell() {
    this.cell = new Cell();
  }
  int get() {
    return this.cell.get();
  }
  void set(int v) {
    this.bak = this.cell.get();
    this.cell.set(v);
  }
  void undo() {
    this.cell.set(this.bak);
  }
}
```

Fig. 1. Java code for the Cell-Recell example with interface inheritance.

by Parkinson and Bierman [14] to show how their logic deals with class-to-class inheritance.

We add to this example a method proxySet, which calls the set method of a given object reference. It is a challenge to give a single specification to this method that is powerful enough to expose any additional side effects the set method might have in arbitrary subclasses. We will see in this section how our specification style achieves this, and it is sketched in Section 5 how this compares to related work.

Our model programming language is a subset of both Java and *C♯*. It leaves out class-to-class inheritance and focuses on interface inheritance. This mode of inheritance captures the essential object-oriented aspect of dynamic dispatch, while the code-reuse aspect has to be explicitly encoded with class composition. A Java implementation of the Cell-Recell example can be found in Figure 1.

2.1 Interface ICell

Interface ICell from Figure 1 is modelled as a parametrised specification that states conditions for whether a class C behaves "Cell-like". In the following, *val* denotes the type of program values, in our case the union of integers, Booleans and object references. Also, $\mathit{UPred}(\mathit{heap})$ is the type of logical propositions over heaps, i.e., the spatial component of the assertion logic (see Section 3.1 for the

precise definition).

$$\begin{aligned} \mathit{ICell} \triangleq\ & \lambda C : \mathit{classname}.\quad \lambda T : \mathit{Type}.\quad \lambda R : \mathit{val} \to T \to \mathit{UPred}(\mathit{heap}). \\ & \lambda g : T \to \mathit{val}.\quad \lambda s : T \to \mathit{val} \to T. \\ & (\forall t : T.\ C{::}\mathsf{get}(\mathsf{this}) \mapsto \{\widehat{R}\ \mathsf{this}\ t\}\_\{\mathsf{r}.\ \widehat{R}\ \mathsf{this}\ t \wedge \mathsf{r} = g\ t\}) \wedge \quad (1) \\ & (\forall t : T.\ C{::}\mathsf{set}(\mathsf{this}, \mathsf{x}) \mapsto \{\widehat{R}\ \mathsf{this}\ t\}\_\{\widehat{R}\ \mathsf{this}\ (\widehat{s}\ t\ \mathsf{x})\}) \wedge \quad (2) \\ & (\forall t, v.\ g\ (s\ t\ v) = v) \quad (3) \end{aligned}$$

There is some notation to explain here. *ICell* is a function that takes five arguments and returns a result of type *spec*, which is the type of specifications. The logical connectives at the outer level ($\wedge$ and $\forall$) thus belong to the specification logic. The parameter R is the representation predicate of class C, so $R\ c\ t$ intuitively means that c is a reference to an object that is mathematically modelled by the value t of type T. The parameters g and s are functions that describe how get and set inspect and transform this mathematical value. They are constrained by (3) to ensure that get will actually return the value set with set.

The notation $C{::}m(\bar{p}) \mapsto \{P\}\_\{r.\ Q\}$ from (1) and (2) specifies that method m of class C has precondition P and postcondition Q. The arguments in a call will be bound to the names $\bar{p}$ in P and Q, and the return value will be bound to r in Q. We support both static and dynamic methods, where dynamic methods have an additional first argument, as seen in (1) and (2). The precise definition is given in Section 4.2.

The notation $\widehat{f}$ from (1) and (2) lifts a function f such that it operates on expressions, including program variables, rather than operating directly on *val*. It is a technical point that can be ignored for a first understanding of this example, but it is crucial for making HOSL work in a stack-based language. Details are in Section 3.2.

The type of T refers to the *Type* universe hierarchy in Coq.

2.2 Method proxySet

Consider method proxySet from Figure 1. Operationally, calling $\mathsf{proxySet}(c, v)$ does the same as calling $c.\mathsf{set}(v)$, and we seek a specification that reflects this. It is crucial for modularity that proxySet can be specified and verified only once and then used with any implementation of ICell that may be defined later. We give it the following specification.

$$\begin{aligned} \mathit{ProxySet\_spec} \triangleq\ & \forall C, T, R, g, s.\ \mathit{ICell}\ C\ T\ R\ g\ s \to \\ & \forall t : T.\ \mathsf{ProxySet}{::}\mathsf{proxySet}(\mathsf{c}, \mathsf{x}) \mapsto \{\mathsf{c} : C \wedge \widehat{R}\ \mathsf{c}\ t\}\_\{\widehat{R}\ \mathsf{c}\ (\widehat{s}\ t\ \mathsf{x})\} \end{aligned}$$

The assertion $\mathsf{c} : C$ means that the object referenced by c is of class C. Thus, the caller of proxySet can pass in an object reference of any class C as long as C can be shown to satisfy *ICell*.

This specification is as powerful as that of set in *ICell* since it essentially forwards it. Any class that behaves Cell-like should be able to encode the behaviour of its set method by a suitable choice of R and s. We will see in Section 2.6 that it, for instance, is possible to pass in a Recell and deduce how proxySet affects its backup value.

2.3 Class Cell

A Java implementation of Cell can be found in Figure 1. We model constructors as static methods that allocate the object before running the initialisation code and return the allocated object, which is what happens in the absence of class-to-class inheritance.

We give class Cell the following specification, which is a conjunction of what we will call an *interface specification* and a *class specification*. These correspond respectively to the *dynamic* and *static* specifications in [14].

$$\mathit{Cell\_spec} \triangleq \exists R_{\mathrm{Cell}}.\ \mathit{ICell}\ \mathsf{Cell}\ \mathit{val}\ R_{\mathrm{Cell}}\ (\lambda v.\ v)\ (\lambda \_, v.\ v) \wedge \mathit{Cell\_class}\ R_{\mathrm{Cell}}$$

where

$$\begin{aligned}\mathit{Cell\_class} \triangleq\ & \lambda R_{\mathrm{Cell}} : \mathit{val} \to \mathit{val} \to \mathit{UPred}(\mathit{heap}).\\ & \mathsf{Cell{::}new}() \mapsto \{\mathit{true}\}\_\{\exists v.\ \widehat{R_{\mathrm{Cell}}}\ \mathsf{this}\ v\} \wedge\\ & (\forall v.\ \mathsf{Cell{::}get(this)} \mapsto \{\widehat{R_{\mathrm{Cell}}}\ \mathsf{this}\ v\}\_\{\mathsf{r}.\ \widehat{R_{\mathrm{Cell}}}\ \mathsf{this}\ v \wedge \mathsf{r} = v\}) \wedge\\ & (\forall v.\ \mathsf{Cell{::}set(this, x)} \mapsto \{\widehat{R_{\mathrm{Cell}}}\ \mathsf{this}\ v\}\_\{\widehat{R_{\mathrm{Cell}}}\ \mathsf{this}\ \mathsf{x}\})\end{aligned}$$

The representation predicate R_{Cell} is quantified such that its definition is visible only while proving the specifications of Cell, thus hiding the internal representation of the class from clients [4,13].

It is crucial that R_{Cell} is quantified outside both the class and the interface specification such that the representation predicate is the same in the two. In practice, a client will allocate a Cell by calling new, which establishes R_{Cell}; later, to model casting the object reference to its interface type, the client knows that *ICell* holds for this same R_{Cell}.

The specifications of get and set in *Cell_class* are identical to their counterparts in *ICell* when C, T, R, g, and s, are instantiated as in *Cell_spec*. In general, the class specification can be more precise than the interface specification, similarly to the dynamic and static specifications of [14].

To prove *Cell_spec*, the existential R_{Cell} is chosen as $\lambda c, v.\ c.\mathsf{value} \mapsto v$. We can then show that *Cell_class* R_{Cell} holds by verifying the method bodies of get, set and init, and the correctness of get and set can be used as a lemma in proving the interface specification. In this way, each method body is verified only once.

2.4 Interface IRecell

To show the analogy to interface inheritance at the specification level, we examine an interface for classes that behave Recell-like. The Java code for that is IRecell in Figure 1. The specification corresponding to this interface follows the same pattern as *ICell*:

$$\begin{aligned}\mathit{IRecell} \triangleq\ & \lambda C : \mathit{classname}.\quad \lambda T : \mathit{Type}.\quad \lambda R : \mathit{val} \to T \to \mathit{UPred}(\mathit{heap}).\\ & \lambda g : T \to \mathit{val}.\quad \lambda s : T \to \mathit{val} \to T.\quad \lambda u : T \to T.\\ & \mathit{ICell}\ C\ T\ R\ g\ s \wedge && (4)\\ & (\forall t : T.\ C{::}\mathsf{undo(this)} \mapsto \{\widehat{R}\ \mathsf{this}\ t\}\_\{\widehat{R}\ \mathsf{this}\ (u\ t)\}) \wedge && (5)\\ & (\forall t, v.\ g\ (u\ (s\ t\ v)) = g\ t) && (6)\end{aligned}$$

Notice that interface extension is modelled by referring to *ICell* in (4). We do not have to respecify get and set since they were already general enough in *ICell* due to it being parametric in g and s. Note how equation (6) specifies the abstract behaviour of undo via g and s.

There is a pattern to how we construct a specification-logic interface predicate from a Java interface declaration. For each method $m(x_1, \ldots, x_n)$, we add a parameter $f_m : T \to val^n \to (val \times T)$. The product $(val \times T)$ can be replaced with just val or T if the method should have no side effects or no return value, respectively. We then add a method specification of the form:

$$\forall t : T.\ C{::}m(\bar{p}) \mapsto \{\widehat{R}\ \mathsf{this}\ t\}\_\{\mathsf{r}.\ \widehat{R}\ \mathsf{this}\ (\pi_2\ (\widehat{f}_m\ \bar{p}\ t)) \wedge \mathsf{r} = \pi_1\ (\widehat{f}_m\ \bar{p}\ t)\}.$$

2.5 Class Recell

The specification of class Recell follows the same pattern as with Cell:

$$\begin{array}{rll} \mathit{Recell\_spec} \triangleq & \exists R_{\mathrm{Recell}} : \mathit{val} \to \mathit{val} \to \mathit{val} \to \mathit{UPred}(\mathit{heap}). & \\ & \mathit{IRecell}\ \mathsf{Recell}\ (\mathit{val} \times \mathit{val})\ R\ g\ s\ u \wedge \mathit{Recell\_class}\ R_{\mathrm{Recell}} & \\ \text{where} & R\ =\ \lambda \mathit{this}, (v, b).\ R_{\mathrm{Recell}}\ \mathit{this}\ v\ b, & g\ =\ \lambda(v, b).\ v, \\ & s\ =\ \lambda(v, b), v'.\ (v', v), & u\ =\ \lambda(v, b).\ (b, b), \end{array}$$

and *Recell_class* is defined analogously to *Cell_class*.

2.6 Class World

The correctness of the above specifications only matters if it enables client code to instantiate and use the classes. The client code in World demonstrates this:

```
class World {
  static ICell make() {
    Recell r = new Recell();
    r.set(5);
    ProxySet::proxySet(r, 3);
    r.undo();
    return r;
  }

  static void main() {
    ICell c = World::make();
    assert c.get() == 5;
  }
}
```

The body of make demonstrates the use of proxySet. Operationally, it should be clear that r has the value 3 and the backup value 5 after the call to proxySet. This can also be proved in our logic despite using a specification of proxySet that was verified without knowledge of Recell and its backup field.

Upon returning from make, we choose to forget that the returned object is really a Recell, upcasting it to ICell. Its precise class is not needed by the caller, main, which only needs to know that the returned object will return 5 from get.

We capture the interaction between these two methods with the following specification, in which $\mathit{FunI} : \mathit{spec} \to \mathit{UPred}(\mathit{heap})$ injects the specification logic

into the logic of propositions over heaps, thus generalising the concept of nested triples. Section 3.5 describes *FunI* in more detail.

$$\mathit{World\_spec} \triangleq \mathsf{World::main}() \mapsto \{\mathit{true}\}\_\{\mathit{true}\} \wedge$$
$$\mathsf{World::make}() \mapsto \{\mathit{true}\}\_\left\{ \begin{array}{l} \mathsf{r}.\ \exists C, T, R, g, s.\ \widehat{\mathit{FunI}}\ (\mathit{ICell}\ C\ T\ R\ g\ s) \wedge \\ \quad \exists t.\ \widehat{R}\ \mathsf{r}\ t \wedge g\ t = 5 \wedge \mathsf{r} : C \end{array} \right\}$$

The make method is specified to return an object whose class C is unknown, but we know that C satisfies *ICell*.

This pattern of returning an object of an unknown type that satisfies a particular specification often comes up in object-oriented programming: think of the method on a collection that returns an iterator, for example. The essence of this pattern is to have a parametrised specification $S : \mathit{classname} \to \mathit{spec}$ and a method specified as $D::m() \mapsto \{\mathit{true}\}\_\{\mathsf{r}.\ \exists C.\ \mathsf{r} : C \wedge \widehat{\mathit{FunI}}\ (S\ C)\}$. A more straightforward alternative to such a specification – one that does not require an embedding of the specification logic in the assertion logic – would be $\exists C.\ S\ C \wedge D::m() \mapsto \{\mathit{true}\}\_\{\mathsf{r}.\ \mathsf{r} : C\}$. However, this restricts the body of m to only being able to return objects of one class. The method body cannot, for example, choose at run time to return either a C_1 or a C_2, where both C_1 and C_2 satisfy S. We find that the most elegant way to allow the method body to make such a choice is to embed the specification in the postcondition.

Using the notion of validity from Definition 5 in Section 3.4 we can now prove that the whole program will behave according to specification:

Theorem 1. (*ProxySet_spec* $\wedge$ *Cell_spec* $\wedge$ *Recell_spec* $\wedge$ *World_spec*) *is valid.*

3 Abstract representation

The core of our system is designed to be language independent. To allow for different memory models, we adopt the notion of separation algebras from Calcagno et al. [6]; we can then instantiate an assertion logic with any separation algebra suitable for the problem at hand. Commands are modelled as relations on the program state, which in turn consists of a mutable stack and a heap. Finally, we define an expressive specification logic that can be used to reason about semantic commands.

We use set-theoretic notation to describe our formalisation as this makes the theories easier to read; in Coq we model these sets as functions into *Prop*, which is the sort of propositions in Coq.

3.1 Uniform predicates

Definition 1 (Separation algebra). *A separation algebra is a partial, cancellative, commutative monoid* $(\Sigma, \circ, \mathbf{1})$ *where* Σ *is the carrier,* $\circ$ *is the monoid operator, and* $\mathbf{1}$ *is the unit element.*

Intuitively, Σ can be thought of as a type of heaps, and the $\circ$-operator as composition of disjoint heaps. Hence we refer to the elements of Σ as heaps. Two heaps are compatible, written $h_1 \# h_2$ if $h_1 \circ h_2$ is defined. A heap h_1 is a subheap of a h_2, written $h_1 \sqsubseteq h_2$, if there exists an h_3 such that $h_2 = h_1 \circ h_3$. We will commonly refer to a separation algebra by its carrier Σ.

A uniform predicate [5] over a separation algebra is a predicate on heaps and natural numbers; it is upwards closed in the heaps and downwards closed in the natural numbers.

$$\mathit{UPred}(\Sigma) \triangleq \{p \subseteq \Sigma \times \mathbb{N} \mid \forall g, m.\ \forall h \sqsupseteq g.\ \forall n \leq m.\ (g, m) \in p \to (h, n) \in p)\}$$

The upward closure in heaps ensures that we have an intuitionistic separation logic as is desirable for garbage-collected languages.

The natural numbers are used to connect the uniform predicates with the step-indexed specification logic – this connection will be covered in Section 3.5.

We define the standard connectives for the uniform predicates as in [5]:

$$\begin{array}{rcl@{\qquad}rcl}
\mathit{true} & \triangleq & \Sigma \times \mathbb{N} & \mathit{false} & \triangleq & \emptyset \\
p \wedge q & \triangleq & p \cap q & p \vee q & \triangleq & p \cup q \\
\forall x : U.\ f & \triangleq & \bigcap_{x:U} f\ x & \exists x : U.\ f & \triangleq & \bigcup_{x:U} f\ x \\
p \to q & \triangleq & \{(h, n) \mid \forall g \sqsupseteq h.\ \forall m \leq n.\ (g, m) \in p \to (g, m) \in q\} & & & \\
p * q & \triangleq & \{(h_1 \circ h_2, n) \mid h_1 \# h_2 \wedge (h_1, n) \in p \wedge (h_2, n) \in q\} & & & \\
p \mathrel{-\!\!*} q & \triangleq & \{(h, n) \mid \forall m \leq n.\ \forall h_1 \# h.\ (h_1, m) \in p \to (h \circ h_1, m) \in q\} & & &
\end{array}$$

For the quantifiers, U is of type *Type*, i.e. the sort of types in Coq, and f is any Coq function from U to $\mathit{UPred}(\Sigma)$. This allows us to quantify over *any* member of *Type* in Coq.

3.2 Stacks

Stacks are functions from variable names to values: $\mathit{stack} \triangleq \mathit{var} \to \mathit{val}$.

Two stacks are said to agree on a set V of variables if they assign the same value to all members of V: $s \simeq_V s' \triangleq \forall x \in V.\ s\ x = s'\ x$. In order to define operators that take values from the stack as arguments we introduce the notion of a *stack monad*. This approach is similar to that of Varming and Birkedal [20].

$$\mathit{sm}\ T \triangleq \{(f : \mathit{stack} \to T,\ V : \mathcal{P}(\mathit{var})) \mid \forall s, s'.\ s \simeq_V s' \to f\ s = f\ s'\}$$

Intuitively, V is an over-approximation of the free program variables in f. For any $m = (f, V) \in \mathit{sm}\ T$, we write $m\ s$ to mean $f\ s$ and $\mathit{fv}\ m$ to mean V.

Theorem 2. *sm is a monad with return operation* $\lambda x : T.\ ((\lambda\_.\ x), \emptyset)$ *and bind operation* $\lambda m : \mathit{sm}\ T.\ \lambda f : T \to \mathit{sm}\ U.\ ((\lambda s.\ f\ (m\ s)\ s),\ \mathit{fv}\ m \cup \bigcup_{t \in T} \mathit{fv}\ (f\ t))$.

We use the stack monad to model expressions (which can be evaluated to values using data from the stack), pure assertions (that represent logical propositions that are evaluated without using the heap), and assertions (that represent logical propositions that are evaluated using both the heap and the stack).

$$\mathit{expr} \triangleq \mathit{sm}\ \mathit{val} \qquad \mathit{pure} \triangleq \mathit{sm}\ \mathit{Prop} \qquad \mathit{asn}(\Sigma) \triangleq \mathit{sm}\ \mathit{UPred}(\Sigma)$$

We create an assertion logic by lifting all connectives from $\mathit{UPred}(\Sigma)$ into $\mathit{asn}(\Sigma)$. The definitions and properties of the liftings follow from the fact that sm is a monad (Theorem 2).We prove that both the uniform predicates and the assertions model separation logic [3].

Theorem 3. *For any separation algebra Σ, $\mathit{UPred}(\Sigma)$ and $\mathit{asn}(\Sigma)$ are complete BI-algebras.*

The stack monad is also used for the lifting operator $\widehat{f}$ that was introduced in Section 2.1. The operator takes a function f, and returns a function $\widehat{f}$ where any argument type T that is passed to f is replaced with $\mathit{sm}\ T$, and any return type U with $\mathit{sm}\ U$. As an example, the representation predicate R in the specification for ICell, which has type $\mathit{val} \to T \to \mathit{UPred}(\mathit{heap})$, is lifted to $\widehat{R}$ in the assertion-logic formulas of the specification. The resulting type for $\widehat{R}$ is $\mathit{sm}\ \mathit{val} \to \mathit{sm}\ T \to \mathit{sm}\ \mathit{UPred}(\mathit{heap})$, i.e. $\mathit{expr} \to \mathit{sm}\ T \to \mathit{asn}(\mathit{heap})$.

We have to make this lifting explicit in specifications because it restricts how program variables behave under substitution. We have that $(\widehat{f}\ e)[e'/x] = \widehat{f}\ (e[e'/x])$ for any $f : \mathit{val} \to \mathit{UPred}(\Sigma)$, but it is not the case that $(g\ e)[e'/x] = g\ (e[e'/x])$ for any $g : \mathit{expr} \to \mathit{asn}(\Sigma)$ because $g\ e$ may have more free program variables than those appearing in e, whereas $\widehat{f}\ e$ cannot, by construction. To make HOSL useful in a stack-based language, where such substitutions are commonplace, we therefore typically quantify over functions into $\mathit{UPred}(\Sigma)$ that we then lift to $\mathit{asn}(\Sigma)$ where needed.

3.3 Semantic commands

To obtain a language-independent core, we model commands as indexed relations on program states (each consisting of a stack and a heap) – a semantic command will relate, in a certain number of steps, a state either to another state or to an error. The only requirements we impose on these commands are that they do not relate to anything in zero steps, and that they satisfy a frame property that will allow us to infer a frame-rule for all semantic commands. Intuitively, the semantic commands can be seen as abstractions of rules of a step-indexed big-step operational semantics. More formally, we have the following definitions.

Definition 2 (pre-command). *A* pre-command $\tilde{c}$ *relates an initial state to either a terminal state or the special* **err** *state:*

$$\mathit{precmd} \triangleq \mathcal{P}(\mathit{stack} \times \Sigma \times ((\mathit{stack} \times \Sigma) \uplus \{\mathbf{err}\}) \times \mathbb{N})$$

We write $(s, h, \tilde{c}) \rightsquigarrow^n x$ to mean that $(s, h, x, n) \in \tilde{c}$.

Definition 3 (Frame property). *A pre-command $\tilde{c}$ has the frame property in case the following holds. If $(s, h_1, \tilde{c}) \not\rightsquigarrow^n$ **err** and $(s, h_1 \circ h_2, \tilde{c}) \rightsquigarrow^n (s', h')$ then there exists h_1' such that $h' = h_1' \circ h_2$ and $(s, h_1, \tilde{c}) \rightsquigarrow^n (s', h_1')$.*

Definition 4 (Semantic command). *A semantic command satisfies the frame property and does not evaluate to anything in zero steps.*

$$\mathit{semcmd} \triangleq \{\hat{c} \in \mathit{precmd} \mid \hat{c} \text{ has the frame property} \wedge \forall s, h, x.\ (s, h, \hat{c}) \not\rightsquigarrow^0 x\}$$

To facilitate the encoding of imperative programming languages in our framework, we create the following semantic commands that can be used as building blocks for that purpose. These commands are similar to the ones found in [6].

$$\mathbf{id} \qquad \mathbf{seq}\ \hat{c}_1\ \hat{c}_2 \qquad \hat{c}_1 + \hat{c}_2 \qquad \hat{c}^* \qquad \mathbf{assume}\ P \qquad \mathbf{check}\ P$$

Intuitively, these semantic commands are defined as follows: The **id**-command is the identity command – it does nothing; the **seq**-command executes two commands in sequence; the +-operator nondeterministically executes one of two commands; the *-command executes a command an arbitrary amount of times; the **assume**-command assumes a pure assertion that can be used to prove correctness of future commands; the **check**-command works like the **id**-command as long as a pure assertion can be inferred. Recall that pure assertions are logical formulas that are evaluated without using the heap.

Theorem 4. **id**, **seq**, $+$, *, **assume**, *and* **check** *are semantic commands.*

3.4 Specification logic

With the assertion logic and the semantic commands in place, we can define the specification logic. Semantically, a specification is a downwards-closed set of natural numbers; this allows us to reason about (mutually) recursive programs via step-indexing.

$$\mathit{spec} \triangleq \{S \subseteq \mathbb{N} \mid \forall m, n.\ m \leq n \wedge n \in S \to m \in S\}$$

The set *spec* is a complete Heyting algebra under the subset ordering, i.e., logical entailment ($\models$) is modelled as subset inclusion. Hence a specification S is *valid* if $S = \mathbb{N}$.

Given assertions P and Q, and semantic command $\hat{c}$, we define a Hoare triple specification:

$$\begin{aligned}\{P\}\hat{c}\{Q\} \triangleq \{n \mid \forall m \leq n.\ \forall k \leq m.\ \forall s, h.\ (h, m) \in P\ s \to (s, h, \hat{c}) \not\rightsquigarrow^k \mathbf{err} \wedge \\ \forall h', s'.\ (s, h, \hat{c}) \rightsquigarrow^k (s', h') \to (h', m - k) \in Q\ s'\}\end{aligned}$$

A program is proved correct by proving that its specification is valid:

Definition 5. *A specification is valid, written* $\models S$*, when true* $\models S$*.*

3.5 Connecting the assertion logic with the specification logic

We define an embedding of the specification logic into the assertion logic as follows:

$$\mathit{FunI} : \mathit{spec} \to \mathit{UPred}(\Sigma) \triangleq \lambda S.\ \Sigma \times S.$$

Lemma 1. *FunI is monotone, preserves implication, and has a left and a right adjoint, when spec and UPred(Σ) are treated as poset categories.*

From the second part of this lemma it follows that *FunI* preserves both finite and infinite conjunctions and disjunctions, which entails that all specification logic connectives are preserved by the translation.

3.6 Recursion

The specification connectives defined in the previous section are not enough for our purposes. When proving a program correct (by proving a formula of the form $\models S$), it is commonplace that the proof of one part of specification in S requires other parts of S – a typical example is recursive method calls, where the specification of the method called must be available in the context during its own verification. To accomplish this, we borrow the *later* operator ($\triangleright$) from Gödel-Löb logic (see [2]).

$$\triangleright S \triangleq \{n+1 \mid n \in S\} \cup S$$

This operator can be used via the Löb rule, which allows us to do induction on the step-indexes of the semantic commands.

$$\frac{\Gamma \wedge \triangleright S \models S \qquad 0 \in \Gamma \to 0 \in S}{\Gamma \models S}\ \textsc{Löb}$$

In the inductive case $\triangleright S$ is found on the left hand side of the turnstile and can hence be used to prove S.

4 Instantiation to an object-oriented language

We define a Java-like language with syntax of programs $\mathcal{P}$ shown below. The language is untyped and does not need syntax for interfaces; these exist in the specification logic only.

We use a shallow embedding for expressions, which we denote with e, as shown in Section 3.2.

$$\begin{array}{rcl}
\mathcal{P} & ::= & \mathcal{C}^* \qquad\qquad f \in \text{(field names)} \\
\mathcal{C} & ::= & \textbf{class}\ C\ f^*\ (m(\bar{x})\{c;\ \textbf{return}\ e\})^* \\
c & ::= & x := \textbf{alloc}\ C \mid x := e \mid x := y.f \mid x.f := e \mid x := y.m(\bar{e}) \\
 & \mid & x := C{::}m(\bar{e}) \mid \textbf{skip} \mid c_1;\ c_2 \mid \textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2 \\
 & \mid & \textbf{while}\ e\ \textbf{do}\ c \mid \textbf{assert}\ e
\end{array}$$

In order to provide a concrete instance of the assertion logic, we construct a separation algebra of concrete heaps. The carrier set is $\mathit{heap} \triangleq (\mathit{ptr} \times \mathit{field}) \xrightarrow[\text{fin}]{} \mathit{val}$, with the values defined as the union of integers, Booleans and object references. The partial composition $h_1 \circ h_2$ is defined as $h_1 \cup h_2$ if $\text{dom}\ h_1 \cap \text{dom}\ h_2 = \emptyset$; otherwise the result is undefined. The unit of the algebra is the empty map, emp. We denote this separation algebra $(\mathit{heap}, \circ, \mathit{emp})$ with heap. The points-to predicate is defined as $v.f \mapsto v' \triangleq \{(h, n) \mid h \sqsupseteq [(v, f) \mapsto v']\}$.

$$\frac{}{\textbf{skip} \sim_{\mathsf{sem}} \textbf{id}}\ \textsc{Skip-Sem} \qquad \frac{c_1 \sim_{\mathsf{sem}} \hat{c_1} \qquad c_2 \sim_{\mathsf{sem}} \hat{c_2}}{c_1;\ c_2 \sim_{\mathsf{sem}} \textbf{seq}\ \hat{c_1}\ \hat{c_2}}\ \textsc{Seq-Sem}$$

$$\frac{c \sim_{\mathsf{sem}} \hat{c}}{\textbf{while}\ e\ \textbf{do}\ c \sim_{\mathsf{sem}} \textbf{seq}\ (\textbf{seq}\ (\textbf{assume}\ e)\ \hat{c})^*\ (\textbf{assume}\ \neg e)}\ \textsc{While-Sem}$$

$$\frac{c_1 \sim_{\mathsf{sem}} \hat{c}_1 \qquad c_2 \sim_{\mathsf{sem}} \hat{c}_2}{\textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2 \sim_{\mathsf{sem}} (\textbf{seq}\ (\textbf{assume}\ e)\ \hat{c}_1) + (\textbf{seq}\ (\textbf{assume}\ \neg e)\ \hat{c}_2)}\ \textsc{If-Sem}$$

Fig. 2. The skip, sequential composition, conditional and loop cases of the semantics relation

4.1 Semantics of the programming language

We define the semantics of the programming language commands by relating them to semantic commands instantiated with *heap* as the separation algebra. We write $c \sim_{\mathsf{sem}} \hat{c}$ to denote that the syntactic command c is related to the semantic command $\hat{c}$. The $\sim_{\mathsf{sem}}$ relation can be thought of as a function; it is defined as a relation only because this was more straightforward in Coq.

The commands **skip**, ;, **if**, and **while** can be related directly to composites of the general semantic commands, defined in Section 3.3. The definition of $\sim_{\mathsf{sem}}$ for these commands can be found in Figure 2. For the remaining commands, new semantic commands must be created.

In particular, for method calls, we define a semantic command

$$\textbf{call}\ x\ C{::}m(\bar{e})\ \textbf{with}\ c\ \hat{c}$$

that, intuitively, calls method m of class C with arguments $\bar{e}$ and assigns the return value to x; the command c is the method body, and $\hat{c}$ is its corresponding semantic command. This semantic command works uniformly for both static and dynamic methods, since in the dynamic case we can pass the object reference as an additional argument. The definition of this semantic command is shown in Figure 3. The definition makes use of a predicate

$$C{::}m(\bar{p})\{c; \textbf{return}\ r\} \in \mathcal{P}$$

which holds in case method m in class C has parameters $\bar{p}$ and method body c in program $\mathcal{P}$. The program parameter $\mathcal{P}$ has been left implicit in the other rules. The notation $[\bar{p} \mapsto (\bar{e}\ s)]$ denotes a finite map that associates each p in $\bar{p}$ with the e at the corresponding position in $\bar{e}$ evaluated in stack s.

The requirement that the method body is related to the semantic command is not enforced by the construction of the semantic command, but rather by the definition of $\sim_{\mathsf{sem}}$ for respectively static and dynamic method calls:

$$\frac{c \sim_{\mathsf{sem}} \hat{c}}{x := C{::}m(\bar{e}) \sim_{\mathsf{sem}} \textbf{call}\ x\ C{::}m(\bar{e})\ \textbf{with}\ c\ \hat{c}}\ \textsc{SCall-Sem}$$

$$\frac{([\bar{p} \mapsto (\bar{e}\ s)], h, \hat{c}) \rightsquigarrow^n (s', h') \qquad C{::}m(\bar{p})\{c; \mathbf{return}\ r\} \in \mathcal{P} \qquad |\bar{p}| = |\bar{e}|}{(s, h, \mathbf{call}\ x\ C{::}m(\bar{e})\ \mathbf{with}\ c\ \hat{c}) \rightsquigarrow^{n+1} (s[x \mapsto (r\ s')], h')}\ \textsc{Call}$$

$$\frac{C{::}m(\bar{p})\{c; \mathbf{return}\ r\} \notin \mathcal{P}}{(s, h, \mathbf{call}\ x\ C{::}m(\bar{e})\ \mathbf{with}\ c\ \hat{c}) \rightsquigarrow^1 \mathbf{err}}\ \textsc{Call-Fail1}$$

$$\frac{C{::}m(\bar{p})\{c; \mathbf{return}\ r\} \in \mathcal{P} \qquad |\bar{p}| \neq |\bar{e}|}{(s, h, \mathbf{call}\ x\ C{::}m(\bar{e})\ \mathbf{with}\ c\ \hat{c}) \rightsquigarrow^1 \mathbf{err}}\ \textsc{Call-Fail2}$$

$$\frac{([\bar{p} \mapsto (\bar{e}\ s)], h, \hat{c}) \rightsquigarrow^n \mathbf{err} \qquad C{::}m(\bar{p})\{c; \mathbf{return}\ r\} \in \mathcal{P} \qquad |\bar{p}| = |\bar{e}|}{(s, h, \mathbf{call}\ x\ C{::}m(\bar{e})\ \mathbf{with}\ c\ \hat{c}) \rightsquigarrow^{n+1} \mathbf{err}}\ \textsc{Call-Fail3}$$

Fig. 3. Semantic call commands.

$$\frac{c \sim_{\mathsf{sem}} \hat{c} \qquad y : C}{x := y.m(\bar{e}) \sim_{\mathsf{sem}} \mathbf{call}\ x\ C{::}m(y, \bar{e})\ \mathbf{with}\ c\ \hat{c}}\ \textsc{DCall-Sem}$$

4.2 Syntactic Hoare triples and the concrete assertion logic

Hoare triples for syntactic commands are defined in the following manner:

$$\{P\}c\{Q\} \triangleq \forall \hat{c}.\ c \sim_{\mathsf{sem}} \hat{c} \to \{P\}\hat{c}\{Q\}.$$

From this definition we infer and prove sound Hoare rules for all commands of our language. To define the rule for method calls we first define the predicate that asserts the specification of methods, introduced in Section 2.1.

$$C{::}m(\bar{p}) \mapsto \{P\}\_\{r.\ Q\} \triangleq \exists c, e.\ wf(\bar{p}, r, P, Q, c) \wedge C{::}m(\bar{p})\{c; \mathbf{return}\ e\} \in \mathcal{P} \\ \wedge \{P\}c\{Q[e/r]\},$$

where *wf* is a predicate to assert the following static properties: the method parameter names do not clash; the pre- and postcondition do not use any stack variables other than the method parameters and this (the postcondition may also use the return variable); the method body does not modify the values of the method parameters or this.

Selected proof rules for syntactic commands are shown in Figure 4. Note the use of the *later* operator ($\triangleright$) in the method call rule; this means that this method call rule will often be used in connection with the Löb rule.

Theorem 5. *The rules in Figure 4 are sound with respect to the operational semantics.*

$$\frac{}{\models \{P\}\mathbf{skip}\{P\}}\ \textsc{Skip} \qquad \frac{}{\{P\}c_1\{Q\} \wedge \{Q\}c_2\{R\} \models \{P\}c_1;\ c_2\{R\}}\ \textsc{Seq}$$

$$\frac{}{\{P \wedge e\}c_1\{Q\} \wedge \{P \wedge \neg e\}c_2\{Q\} \models \{P\}\mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\{Q\}}\ \textsc{If}$$

$$\frac{}{\{P \wedge e\}c\{P\} \models \{P\}\mathbf{while}\ e\ \mathbf{do}\ c\{P \wedge \neg e\}}\ \textsc{While} \qquad \frac{P \vdash e}{\models \{P\}\mathbf{assert}\ e\{P\}}\ \textsc{Assert}$$

$$\frac{}{\models \{\mathit{true}\}x :=\ \mathbf{alloc}\ C\{\forall^* f \in \mathit{fields}(C).\ x.f \mapsto \mathit{null}\}}\ \textsc{Alloc}$$

$$\frac{}{\models \{P\}x := e\{\exists v.\ P[v/x] \wedge x = e[v/x]\}}\ \textsc{Assign} \qquad \frac{}{\models \{x.f \mapsto \_\}x.f := e\{x.f \mapsto e\}}\ \textsc{Write}$$

$$\frac{P \vdash y.f \mapsto e}{\models \{P\}x := y.f\{\exists v.\ P[v/x] \wedge x = e[v/x]\}}\ \textsc{Read}$$

$$\frac{\Gamma \models \triangleright C::m(\bar{p}) \mapsto \{P\}\_\{r.\ Q\} \qquad |\bar{p}| = |y, \bar{e}|}{\Gamma \models \{y : C \wedge P[y, \bar{e}/\bar{p}]\}x := y.m(\bar{e})\{\exists v.\ Q[x, y[v/x], \bar{e}[v/x]/r, \bar{p}]\}}\ \textsc{DCall}$$

$$\frac{\Gamma \models \triangleright C::m(\bar{p}) \mapsto \{P\}\_\{r.\ Q\} \qquad |\bar{p}| = |\bar{e}|}{\Gamma \models \{P[\bar{e}/\bar{p}]\}x := C::m(\bar{e})\{\exists v.\ Q[x, \bar{e}[v/x]/r, \bar{p}]\}}\ \textsc{SCall}$$

$$\frac{P \vdash P' \qquad Q' \vdash Q}{\{P'\}c\{Q'\} \models \{P\}c\{Q\}}\ \textsc{Consequence} \qquad \frac{\forall x \in \mathit{fv}\ R.\ c\ \mathit{does\ not\ modify}\ x}{\{P\}c\{Q\} \models \{P * R\}c\{Q * R\}}\ \textsc{Frame}$$

$$\frac{P \vdash P' \qquad Q' \vdash Q \qquad \mathit{fv}\ P \subseteq \mathit{fv}\ P' \qquad \mathit{fv}\ Q \subseteq \mathit{fv}\ Q'}{C::m(\bar{p}) \mapsto \{P'\}\_\{r.\ Q'\} \models C::m(\bar{p}) \mapsto \{P\}\_\{r.\ Q\}}\ \textsc{Consequence-MSpec}$$

$$\frac{\mathit{fv}\ R \subseteq \{\mathsf{this}\} \cup \bar{p}}{C::m(\bar{p}) \mapsto \{P\}\_\{r.\ Q\} \models C::m(\bar{p}) \mapsto \{P * R\}\_\{r.\ Q * R\}}\ \textsc{Frame-MSpec}$$

Fig. 4. Specification logic rules for syntactic Hoare triples

5 Related work

Formalisations of higher-order separation logic have been proposed before, e.g. by Varming and Birkedal [20], who developed an Isabelle/HOL formalisation of HOSL for partial correctness for a simple imperative language with first-order mutually recursive procedures, using a denotational semantics of the programming language, and by Preoteasa [15], who developed a PVS formalisation for total correctness using a predicate-transformer semantics for a similar programming language.

Parkinson and Bierman treated an extended version of the Cell-Recell example in [14], improving upon their earlier work in [13]. Their approach is to tailor the specification logic to build in a form of quantification over families of rep-

resentation predicates following a fixed pattern determined by the inheritance tree of the program. This construction is known as *abstract predicate families* (APFs).

Where our logic allows quantification over a representation type T, as used in Section 2.1, APFs have a built-in notion of variable-arity predicates to achieve same effect: representation predicates of a subclass can add parameters to the representation predicate they inherit. Class Cell defines a two-parameter representation predicate family Val, which is extended to three arguments in Recell. A Recell r having value 2 and backup field 1 would be asserted as $Val(r, 2, 1)$. This assertion implies $Val(r, 2)$, which in turn implies $\exists b. Val(r, 2, b)$ if it is known that r is a Recell. Thus, casting to the two-argument representation predicate that would be necessary for calling $\{\exists v. Val(\mathsf{c}, v)\}$ $\mathsf{proxySet}(\mathsf{c}, \mathsf{x})$ $\{Val(\mathsf{c}, \mathsf{x})\}$ will lose any information about the backup field.

The logic of Parkinson and Bierman was extended by van Staden and Calcagno [19] to handle multiple inheritance, abstract classes and controlled leaking of facts about the abstract representation of either a single class or a class hierarchy. Using the latter feature, we observe that their logic can also be used to reason about the example in Section 2, by using parameters g and s to give a precise specification of proxySet. Instead of being functions, g and s would be abstract predicate families whose first argument would be an object reference used only for selecting the correct member of the APF.

Compared to the logics based on abstract predicate families, our logic allows families of not just predicates but also types, functions, class names or any other type that can be quantified over in Coq. This gives us strong typing of logical variables, and all this works without building it into the logic and requiring that quantifications and proofs follow the shape of the inheritance tree.

6 Conclusion and Future Work

We have presented a Coq implementation of a generic framework for higher-order separation logic. In this framework, instantiated with a simple object-oriented language, we have shown how HOSL can be used to reason about interfaces and interface inheritance.

Future work includes developing better support for automation via better use of tactics. Our Coq proofs of example programs are cluttered with manual reordering of the context because we do not yet have tactics to automate this. We also plan to integrate the current tool with an Eclipse front-end that is currently being researched within our project [10]. Moreover, we plan to use the tool for formal verification of interesting data structures from the C5 collection library.

Although it is not necessary for the code we mostly want to verify, proper support for class-to-class inheritance in both the logic and the design pattern would enable more direct comparison with related work. It would also make our Java subset more similar to actual Java.

Acknowledgements We are grateful for useful discussions with Hannes Mehnert, Matthew Parkinson, Peter Sestoft, Kasper Svendsen, and Jacob Thamsborg.

This worked was supported in part by the ToMeSo project, funded by the Danish Research Council.

References

1. M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1st edition, 1996.
2. A. W. Appel, P.-A. Melliès, C. D. Richards, and J. Vouillon. A very modal model of a modern, major, general type system. In *Proceedings of POPL*, 2007.
3. B. Biering, L. Birkedal, and N. Torp-Smith. BI hyperdoctrines and higher-order separation logic. In *Proceedings of ESOP*, pages 233–247, 2005.
4. B. Biering, L. Birkedal, and N. Torp-Smith. BI-hyperdoctrines, higher-order separation logic, and abstraction. *ACM Trans. Program. Lang. Syst.*, 29(5), 2007.
5. L. Birkedal, B. Reus, J. Schwinghammer, K. Støvring, J. Thamsborg, and H. Yang. Step-indexed kripke models over recursive worlds. In *Proceedings of POPL*, 2011.
6. C. Calcagno, P. W. O'Hearn, and H. Yang. Local action and abstract separation logic. In *Proceedings of LICS*, pages 366–378, 2007.
7. J. Jensen, L. Birkedal, and P. Sestoft. Modular verification of linked lists with views via separation logic. *Journal of Object Technology*, 2011. To Appear. Preliminary version in FTfJP'10, available at www.itu.dk/people/birkedal/papers/views.pdf.
8. N. Kokholm and P. Sestoft. The C5 generic collection library for C# and CLI. Technical Report ITU-TR-2006-76, IT University of Copenhagen, 2006.
9. N. R. Krishnaswami, J. Aldrich, L. Birkedal, K. Svendsen, and A. Buisse. Design patterns in separation logic. In *In Proceedings of TLDI*, pages 105–116, 2009.
10. H. Mehnert. Kopitiam: modular incremental interactive full functional static verification of java code. In M. Bobaru, K. Havelund, G. Holzmann, and R. Joshi, editors, *Proceedings of the Third NASA Formal Methods Symposium (NFM 2011)*. NASA, April 2011.
11. A. Nanevski, A. Ahmed, G. Morrisett, and L. Birkedal. Abstract predicates and mutable ADTs in hoare type theory. In *In Proc. of ESOP*, pages 189–204, 2007.
12. P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Proceedings of CSL*, pages 1–19, 2001.
13. M. J. Parkinson and G. M. Bierman. Separation logic and abstraction. In *Proceedings of POPL*, pages 247–258, 2005.
14. M. J. Parkinson and G. M. Bierman. Separation logic, abstraction and inheritance. In *Proceedings of POPL*, pages 75–86, 2008.
15. V. Preoteasa. Frame rules for mutually recursive procedures manipulating pointers. *Theoretical Computer Science*, 2009.
16. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of LICS*, pages 55–74, 2002.
17. J. Schwinghammer, L. Birkedal, B. Reus, and H. Yang. Nested Hoare triples and frame rules for higher-order store. In *Proceedings of CSL*, 2009.
18. K. Svendsen, L. Birkedal, and M. Parkinson. Verifying generics and delegates. In *Proceedings of ECOOP*, pages 175–199, 2010.
19. S. van Staden and C. Calcagno. Reasoning about multiple related abstractions with multistar. In *Proceedings of OOPSLA*, pages 504–519, 2010.
20. C. Varming and L. Birkedal. Higher-order separation logic in Isabelle/HOLCF. *Electr. Notes Theor. Comput. Sci.*, 218:371–389, 2008.