Charge!

A framework for higher-order separation logic in Coq

Jesper Bengtson, Jonas Braband Jensen, and Lars Birkedal

IT University of Copenhagen

Abstract. We present a comprehensive set of tactics for working with a shallow embedding of a higher-order separation logic for a subset of Java in Coq. The tactics make it possible to reason at a level of abstraction similar to pen-and-paper separation-logic proof outlines. In particular, the tactics allow the user to reason in the embedded logic rather than in the concrete model, where the stacks and heaps are exposed. The development is generic in the choice of heap model, and most of the development is also independent of the choice of programming language.

1 Introduction

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 postconditions) and the specification logic (the logic of Hoare triples). Higher-order separation logic has proved useful for modular reasoning about programs that use shared mutable data structures and data abstraction, via quantification over resource invariants, and for reasoning about various forms of higher-order programming (higher-order functions, code pointers, interfaces in object-oriented programming) [4,10,12,15,8,2].

This paper describes Charge!, a separation-logic verification tool that aims to (1) prove full functional correctness of Java-like programs using higher-order separation logic, (2) produce machine-checkable correctness proofs, (3) work as close as possible to how informal separation logic proofs are carried out on pen and paper, and (4) automate tedious first-order reasoning where possible.

The first and second goal virtually mandate that we build the tool inside an existing proof assistant for higher-order logic such as Coq. All other tools with these properties that we know of [6,16,11,1] have been built this way; building them from the ground up instead would be a very ambitious undertaking.

To achieve the third goal, it is important that the user has the feeling of reasoning in the program logic, rather than in the model of the logic. For separation logic, this entails reasoning about the linear fragments of the logic, like on paper, rather than reasoning about concrete machine states, such as the stack or the heap, and disjointness properties of various heap fragments. In our own earlier work [2], based on a shallow embedding of higher-order separation logic in the Coq proof assistant, this goal was achieved and the programs were verified without the user ever seeing an explicit heap or a stack in the proof context. However,

as the built-in tactics of Coq are not designed to handle the linear fragments of separation logic, our proofs were long, tedious, and to a large degree focusing mainly on rewriting the proof state modulo associativity and commutativity of separating conjunction.

In this article, we improve on that work and present a comprehensive set of tactics for working with a shallow embedding of a higher-order separation logic for a subset of Java. The tactics make it possible to reason at a level of abstraction similar to pen and paper proofs. In particular, the tactics allow the user to reason in the embedded logic rather than in the concrete model, where the stacks and heaps are exposed. The use of these tactics significantly reduce the size of the proofs compared to our earlier development [2].

The remainder of this paper is structured as follows. In Section 2, we discuss how we deal with program variables in Charge! – how the relation between logical variables and program variables is made as transparent as possible in a proof assistant. In Section 3, we present an extended example of how a user can prove correctness of an implementation of binary search trees using Charge!. Then, in Section 4 we present the most prominent tactics in our development and discuss how they affect the goal states in Coq. Sections 5 covers specifics on how the tactics are implemented and Section 6 touches on how Charge! can be adapted to new languages. Section 7 covers related work, and Section 8 concludes.

This paper is structured top-down: we only include the definitions that are required for the exposition and focus on describing what the tactics do before going into how they do it. The interested reader can find all of the core definitions of the assertion and specification logics as well as examples on how these are used to verify object-oriented programs in [2]. Our Coq development can be downloaded from http://itu.dk/research/charge.

2 Program variables

A key to approaching pen-and-paper style of reasoning is to handle program variables naturally. We feel that the variables-as-resource approach [13] is too unnatural for this purpose. The approach of McCreight [11], while practical, also looks very different from pen-and-paper proofs. Our philosophy is that a user must be able to blur the boundaries between logical variables and program variables, as is often done on paper, in a proof assistant. In this section we discuss how we make this blurring formal.

It is typical in a richly typed separation logic to distinguish between program variables, whose values are restricted to the types offered by the programming language, and logical variables, which can be lists, functions, and other types offered by the logic. Program variables occurring in triples $\{P\}c\{Q\}$ refer to local (stack) variables in c. For example, the heap-write rule can be written

$$\frac{}{\{x.f \mapsto\!\!\!\cdot\; e\}\; x.f := e'\; \{x.f \mapsto\!\!\!\cdot\; e'\}}$$

Here, $x : \mathit{var}$ is a program variable name, typically a pointer to an object, $e, e' : \mathit{expr}$ are programming language expressions, and $f : \mathit{field}$ is a field name.

The expression $x.f \mapsto\!\!\!\!\cdot\; e$ reads that the expression e can be found at the memory address $x.f$. There are free program variables in these assertions since x is itself a program variable, and e and e' may contain program variables.

In pen-and-paper theories of Hoare logic, it is often imagined that the fragment of mathematics required for the proof, such as the theory of lists, is recreated inside the assertion logic in a version where program variables may occur in assertions. When encoding this in a proof assistant, it is not enough to imagine it, and actually doing it would be far too much work. We want to reuse existing theories as they are, and we want to build new theories without being concerned about program variables before there even is a program.

This includes the theory of heap assertions, which is independent of program variables [17,11] even though the two are very often defined together and become inseparable [1,8]. One primitive in this theory is the points-to predicate $(\mapsto) : \mathit{val} \to \mathit{field} \to \mathit{val} \to (\mathit{heap} \to \mathit{Prop})$[1], where *val* is the type of data values for the programming language under consideration. To use the points-to predicate in pre and postconditions, like we saw in the heap-write rule above, we *lift* it to $(\mapsto\!\!\!\!\cdot) : \mathit{expr} \to \mathit{open}\ \mathit{field} \to \mathit{expr} \to \mathit{open}\ (\mathit{heap} \to \mathit{Prop})$.

An *open* T is intuitively a T that may have free program variables:

$$\mathit{open}\ T \triangleq \mathit{stack} \to T \qquad \mathit{stack} \triangleq \mathit{var} \to \mathit{val} \qquad \mathit{expr} \triangleq \mathit{open}\ \mathit{val}$$

In general, the operator lift_n will lift constants and functions of type $(T_1 \to \cdots \to T_n \to U)$ into $(\mathit{open}\ T_1 \to \cdots \to \mathit{open}\ T_n \to \mathit{open}\ U)$. Unfolding the definition of the lifted points-to predicate $(\mapsto\!\!\!\!\cdot)$ makes the write-rule read

$$\frac{}{\{(\mathit{lift}_3\ (\mapsto))\ (\mathit{ve}\ x)\ (\mathit{lift}_0\ f)\ e\}\ x.f := e'\ \{(\mathit{lift}_3\ (\mapsto))\ (\mathit{ve}\ x)\ (\mathit{lift}_0\ f)\ e'\}}$$

where *ve* is the injection from variable names to expressions. Further unfolding the definitions of lift_3 and *ve*, the rule reads

$$\frac{}{\{\lambda s.\ (s\ x).f \mapsto (e\ s)\}\ x.f := e'\ \{\lambda s.\ (s\ x).f \mapsto (e'\ s)\}}$$

Note that this is only an exposition – the user will never see an explicit stack in the proof context. We see that nothing very deep is involved in this treatment of program variables, but it offers some very convenient properties. First, it avoids an explicit mentioning of stacks s. Second, the property of substitutions that $(e_1.f \mapsto\!\!\!\!\cdot\; e_2)\{^e/_x\} = e_1\{^e/_x\}.f \mapsto\!\!\!\!\cdot\; e_2\{^e/_x\}$ follows from the definition of the lifting and is independent of the definition of $\mapsto$. As long as all operators in an assertion are lifted, substitutions will propagate automatically over the connectives and be applied when they reach the program variables – Coq does this computationally, hence it is very fast, and the tactics do not have to reason about any meta-theoretical properties of substitution.

[1] For the sake of exposition, we assume that our heap assertions are of type $\mathit{heap} \to \mathit{Prop}$. See [2] for the full definition.

3 Example

To introduce our tactics, we use a library of binary search trees. The specification and the code in this example are transliterations of the sorted_bintree example that comes with the VeriFast tool [9]. By stepping through a method of the library command by command, we demonstrate how our tactics modify the goal of the proof assistant while allowing the user to reason strictly at the level of the assertion logic – in this section there are no explicit stacks, there are no explicit heaps, and there are no explicit substitutions. They are all present in the background, but they are hidden from the user.

The library has methods $\mathsf{init}(\mathsf{x})$ for creating a singleton tree, $\mathsf{contains}(\mathsf{t},\mathsf{x})$ for membership query and $\mathsf{add}(\mathsf{t},\mathsf{x})$ for adding a single element. We specify the methods in terms of the representation predicate $\mathit{tree}(t, b)$, which describes the memory footprint of a tree with root pointer t and contents b, with $b : \mathit{bintree}$ defined as $b \triangleq \mathit{empty} \mid \mathit{node}\ n\ b\ b$, where n is an integer. We also create a predicate *indorder* of type $\mathit{bintree} \to \mathit{Prop}$ that holds if b is a proper search tree, and a function t_cont of type $\mathit{bintree} \to \mathbb{Z} \to \mathit{bool}$ that assumes that b is a search three and checks for membership of a value in the standard way. We also have a *TreeRec*-predicate that describes the footprint of a binary search tree

$$
\begin{array}{l}
\mathit{TreeRec}\ t\ b \triangleq \\
\quad \mathsf{match}\ b\ \mathsf{with} \\
\quad\quad \mid \mathit{empty} \qquad\quad \Rightarrow t = \mathit{null} \\
\quad\quad \mid \mathit{node}\ v\ \mathit{bl}\ \mathit{br} \Rightarrow t.\mathsf{value} \mapsto v * \exists l.\ \exists r.\ t.\mathsf{left} \mapsto l * t.\mathsf{right} \mapsto r * \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{TreeRec}\ l\ \mathit{bl} * \mathit{TreeRec}\ r\ \mathit{br} \\
\quad \mathsf{end}
\end{array}
$$

where t is a pointer to the heap, and b is of type *bintree*. The connective $*$ is the standard separating conjunction where $p * q$ reads that p is true for one part of the heap, and q is true for another disjoint part of the heap. Finally, we use this definition to define $\mathit{tree}\ t\ b \triangleq \mathit{TreeRec}\ t\ b \wedge \mathit{inorder}\ b$. The library is specified as follows, where $\dot{f}\ x_1\ \cdots\ x_n$ means $(\mathit{lift}_n\ f)\ x_1\ \cdots\ x_n$, and $\mathsf{C}::\mathsf{m}(\vec{\mathsf{x}}) \mapsto \{p\}\_\{q\}$ means that the class C contains a method m, with arguments $\vec{\mathsf{x}}$, that has a specification with the precondition p and the postcondition q.

$$
\begin{array}{l}
\mathit{Tree\_spec} \triangleq \forall b. \\
\qquad \mathsf{Tree}::\mathsf{init}(\mathsf{x}) \mapsto \{\dot{\top}\}\_\{\dot{tree}\ \mathsf{ret}\ (\dot{node}\ \mathsf{x}\ \dot{empty}\ \dot{empty})\} \wedge \\
\qquad \mathsf{Tree}::\mathsf{contains}(\mathsf{t},\mathsf{x}) \mapsto \{\dot{tree}(\mathsf{t}, \dot{b})\}\_\{\dot{tree}(\mathsf{t}, \dot{b}) \dot{\wedge} \mathsf{ret} \doteq t\_\dot{cont}(\dot{b}, \mathsf{x})\} \wedge \\
\qquad \mathsf{Tree}::\mathsf{add}(\mathsf{t},\mathsf{x}) \mapsto \begin{array}{l} \{\dot{tree}(\mathsf{t}, \dot{b}) \dot{\wedge} b \neq \dot{empty} \dot{\wedge} t\_\dot{cont}(\dot{b}, \mathsf{x}) \doteq \dot{false}\}\_ \\ \{\dot{tree}(\mathsf{t}, \dot{tree\_add}(\dot{b}, \mathsf{x}))\} \end{array}
\end{array}
$$

This is a predicate in our specification logic; for a full disclosure, see [2]. It lists the methods of a specification as well as their pre and their postconditions. The program variable ret in the postconditions store the return value of the method. In this example, we focus on verifying the contains-method. We need

the following lemmas

$$\mathsf{TreeRec\_null} \triangleq \forall b.\ (\mathit{TreeRec}\ \mathit{null}\ b \vdash \mathit{TreeRec}\ \mathit{null}\ b \wedge b = \mathit{empty})$$
$$\mathsf{TreeRec\_not\_null} \triangleq \forall t\ b.\ t \neq \mathit{null} \rightarrow (\mathit{TreeRec}\ t\ b \vdash \mathit{TreeRec}\ t\ b \wedge b \neq \mathit{empty})$$

Both lemmas follow directly from the definition of *TreeRec*. The connectives in the predicates are not lifted. The contains-method is defined as follows.

```
contains(t, x) =
  if t ≐ null then ret := false else
    v := t.value; if x ≐ v then ret := true
                  else if x ≺̇ v then l := t.left; ret := t.contains(l, x)
                  else r := t.right; ret := t.contains(r, x)
```

The operators in the conditional statements are lifted from Coq's standard library. To demonstrate how our tactics operate on this method, we step through the proof one command at a time. After some initial boiler-plate setup, our state looks as follows. For the exposition, we omit lifting constants such as $\dot{b}$.

$$\frac{b : \mathit{bintree}}{\{\dot{\mathit{tree}}\ \mathtt{t}\ b\}\mathbf{if}\ \mathtt{t} \doteq \mathtt{null}\ \mathbf{then}\ \mathtt{ret} := \mathit{false}\ \mathbf{else}\ \ldots\{\dot{\mathit{tree}}\ \mathtt{t}\ b \mathbin{\dot{\wedge}} \mathtt{ret} \doteq \dot{t\_cont}\ b\ \mathtt{x}\}}$$

We denote the postcondition of this triple with $\mathcal{P}$. Using the forward-tactic generates two sub-goals – one for each branch of the if-statement.

$$1\ \frac{b : \mathit{bintree}}{\left\{\begin{array}{l}\dot{\mathit{tree}}\ \mathtt{t}\ b \mathbin{\dot{\wedge}} \\ \mathtt{t} \doteq \mathit{null}\end{array}\right\}\mathtt{ret} := \mathit{false}\{\mathcal{P}\}} \qquad 2\ \frac{b : \mathit{bintree}}{\left\{\begin{array}{l}\dot{\mathit{tree}}\ \mathtt{t}\ b \mathbin{\dot{\wedge}} \\ \mathtt{t} \dot{\neq} \mathit{null}\end{array}\right\}\mathtt{v} := \mathtt{t.value};\ \ldots\{\mathcal{P}\}}$$

We start by proving subgoal 1. There is only one command, and applying the forward-tactic provides the following proof obligation.

$$\frac{b : \mathit{bintree} \qquad n : \mathbb{Z} \qquad H : \mathit{inorder}\ b}{\mathit{TreeRec}\ \mathtt{null}\ b \vdash \mathit{false} = \mathit{t\_cont}\ b\ n}$$

A few things have happened here. First of all, since the only command in the triple has been evaluated, the user is left with an assertion logic entailment to prove, the stack has been applied, and all liftings have been evaluated. Secondly, the program variable `x` has been replaced with an integer n, representing its value on the stack. Thirdly, the equivalence $\mathtt{t} = \mathit{null}$ has been applied. Finally, the *tree*-predicate has been evaluated and its non-spatial components (that do not depend on the heap) have been placed in the context. To prove the goal, we must infer that b is empty. The command `sl_apply TreeRec_null` applies the lemma *TreeRec_null* using forward-reasoning and places the non-spatial parts of the consequent in the context.

$$\frac{b : \mathit{bintree} \qquad n : \mathbb{Z} \qquad H : \mathit{inorder}\ b \qquad H_1 : b = \mathit{empty}}{\mathit{TreeRec}\ \mathtt{null}\ b \vdash \mathit{false} = \mathit{t\_cont}\ b\ n}$$

The command `sl_auto` then solves the goal. We now proceed to prove goal 2. We cannot immediately apply the forward-tactic as the precondition does not assert

what t.value is on the heap. The command `unfold tree; triple_nf` unfolds the *tree*-predicate and places the non-spatial predicate *inorder b* in the context.

$$\frac{b : \mathit{bintree} \qquad H : \mathit{inorder}\ b}{\{\mathit{Tree\dot{R}ec}\ \mathtt{t}\ b \mathbin{\dot\wedge} \mathtt{t} \mathbin{\dot\neq} \mathit{null}\}\mathtt{v} := \mathtt{t.value};\ \ldots\{\mathcal{P}\}}$$

More precisely, `triple_nf` places the goal in a normal form; this is discussed in more detail in sections 4 and 5.3. The command `sl_apply TreeRec_not_null` applies the lemma *TreeRec_not_null* in a forward-reasoning style to the precondition of the triple, again moving the non-spatial components of the consequent of the lemma to the context. Remember that the lemma is defined as an assertion logic formula, and its connectives are not lifted.

$$\frac{b : \mathit{bintree} \qquad H : \mathit{inorder}\ b \qquad H_1 : b \neq \mathit{empty}}{\{\mathit{Tree\dot{R}ec}\ \mathtt{t}\ b \mathbin{\dot\wedge} \mathtt{t} \mathbin{\dot\neq} \mathit{null}\}\mathtt{v} := \mathtt{t.value};\ \ldots\{\mathcal{P}\}}$$

The command `destruct b; [congruence| clear` H_1`]` does case-analysis on b. The proof for the case where b is empty is trivial as there is a contradiction in H_1. For the remaining case, H_1 is not needed and is cleared from the context.

$$\frac{v : \mathbb{Z} \qquad b_1 : \mathit{bintree} \qquad b_2 : \mathit{bintree} \qquad H : \mathit{inorder}\ (\dot{\mathit{node}}\ \ v\ b_1\ b_2)}{\left\{\begin{array}{l}\mathit{Tree\dot{R}ec}\ \mathtt{t}\ (\dot{\mathit{node}}\ \ v\ b_1\ b_2)\\ \dot\wedge\ \mathtt{t} \mathbin{\dot\neq} \mathit{null}\end{array}\right\}\ \mathtt{v} := \mathtt{t.value};\ \mathcal{B}\ \left\{\begin{array}{l}\dot{\mathit{tree}}\ \mathtt{t}\ (\dot{\mathit{node}}\ \ v\ b_1\ b_2) \mathbin{\dot\wedge}\\ \mathtt{ret} \doteq \dot{t\_\mathit{cont}}\ (\dot{\mathit{node}}\ \ v\ b_1\ b_2)\ \mathtt{x}\end{array}\right\}}$$

We denote the postcondition of this triple with $\mathcal{P}'$, and the rest of the program $\mathcal{B}$. The forward-tactic can now be applied as reducing *TreeRec* provides the content of t.value.

$$\frac{\begin{array}{l}v : \mathbb{Z} \qquad b_1 : \mathit{bintree} \qquad b_2 : \mathit{bintree} \qquad x_1 : \mathit{val} \qquad x_2 : \mathit{val}\\ H : \mathit{inorder}\ (\mathit{node}\ v\ b_1\ b_2)\end{array}}{\left\{\begin{array}{l}\mathtt{t.value} \dot\mapsto v \mathbin{\dot*} \mathtt{t.left} \dot\mapsto x_1 \mathbin{\dot*} \mathtt{t.right} \dot\mapsto x_2 \mathbin{\dot*} \mathit{Tree\dot{R}ec}\ x_1\ b_1 \mathbin{\dot*}\\ \mathit{Tree\dot{R}ec}\ x_2\ b_2 \mathbin{\dot\wedge} (\mathtt{v} \doteq v \mathbin{\dot\wedge} \mathtt{t} \mathbin{\dot\neq} \mathit{null})\end{array}\right\}\mathcal{B}\{\mathcal{P}'\}}$$

Here the *TreeRec*-predicate has been unfolded and its existentially quantified variables have been extracted to the context. We denote the context of this goal with $\mathcal{C}$ and the spatial component of the precondition with $\mathcal{S}$. The forward-tactic, following the structure of the code, again splits the conditional into two cases. For space reasons, we only cover the first case.

$$\frac{\mathcal{C}}{\{\mathcal{S} \mathbin{\dot\wedge} (\mathtt{x} \doteq \mathtt{v} \mathbin{\dot\wedge} \mathtt{v} \doteq v \mathbin{\dot\wedge} \mathtt{t} \mathbin{\dot\neq} \mathit{null})\}\mathtt{ret} := \mathit{true}\{\mathcal{P}'\}}$$

Since there is only one command in the triple, applying the forward-tactic leaves the user to prove the following entailment.

$$\frac{\mathcal{C} \qquad k : \mathit{val} \qquad H_2 : k \neq \mathit{null}}{\left(\begin{array}{l}k.\mathtt{value} \mapsto v * k.\mathtt{left} \mapsto x_1 * k.\mathtt{right} \mapsto x_2\\ * \ \mathit{TreeRec}\ x_1\ b_1 * \mathit{TreeRec}\ x_2\ b_2\end{array}\right) \vdash \left(\begin{array}{l}\mathit{tree}\ k\ (\mathit{node}\ v\ b_1\ b_2) \wedge\\ \mathit{true} = t\_\mathit{cont}\ (\mathit{node}\ v\ b_1\ b_2)\ v\end{array}\right)}$$

Here, the program variable `t` is evaluated to k, and `x` is replaced by v as they are equivalent. To prove the entailment we must prove that $t\_cont\ (node\ v\ b_1\ b_2)\ v$ holds, which it does by definition. Moreover, to prove that $tree\ k\ (node\ v\ b_1\ b_2)$ holds we must prove that $inorder\ (node\ v\ b_1\ b_2)$ holds, which we have from the context, and that $TreeRec\ k\ (node\ v\ b_1\ b_2)$ holds, which assuming that we instantiate the existential quantifiers of *TreeRec* correctly, is provable directly from the hypothesis of the entailment. The tactic `sl_simpl` solves the goal.

The rest of the proof follows the same pattern and is not more complicated. For the recursive method call, the precondition is proven as a separate assertion logic entailment, but also that follows the same pattern.

One of the main points of the trace above is to demonstrate what is not there as much as what is there. There are three notable things that are not in the trace: there are no visible stacks, there are no visible heaps, and there are no visible substitutions. They are all present, and they all play important roles, but they are never exposed to the user.

4 Tactics

For the rest of the paper, we will split assertions into three different categories: spatial assertions that depend on both the heap and the stack, denoted by t, u, or v, pure assertions that depend only on the stack, denoted by p, q, or r, and propositional assertions that depend on neither the heap nor the stack, denoted by P, Q, or R. We will denote assertions that can be either spatial, pure, or propositional with a, b, or c. Propositional assertions can be viewed as the standard *Prop*-sort in Coq. There are injections from propositional assertions to pure assertions to spatial assertions, but we leave these implicit in the presentation.

Like most custom-made tactics, we make use of existential variables in Coq. An existential variable is a variable in Coq's meta-logic. It has a type, but it has not yet been assigned a value. It can be thought of as a hole in the proof waiting to be filled. Existential variables will be preceded by a ? (for instance $?x$, $?y$, or $?z$). A valid proof can have no uninstantiated existential variables.

Tactics are split into two sub-categories – those that operate on the assertion logic, and those that operate on Hoare-triples. Both types of tactics require, and enforce, that the goal is in a normal form. Neither the premise of an entailment nor the precondition of a triple may contain existential quantifiers or propositional assertions; if they do, they are extracted to the Coq context. Moreover, in the case of triples, pure and spatial assertions are kept separate. More formally, the following goals are in normal form

$$\frac{\overrightarrow{H : P}}{\{(t_1 * \cdots * t_n) \wedge (p_1 \wedge \cdots \wedge p_m)\}c\{a\}} \qquad \frac{\overrightarrow{H : P}}{t_1 * \cdots * t_n \vdash a}$$

where $\overrightarrow{H : P}$ are the premises (zero or more) in the Coq-context. In both cases, t_1 to t_n and p_1 to p_m are atomic, i.e. they contain no further occurrences of $*$ and $\wedge$ respectively. We say that $t_1 * \cdots * t_n$ is a linear assertion. The reason that

pure assertions are kept in the precondition for triples and not in entailments is that pure and propositional assertions are indistinguishable in the assertion logic – the stack has already been fully applied and all liftings have been computed. In Section 5.3 we cover how to rewrite a goal to normal form.

4.1 Tactics on the assertion logic

All of the tactics for the assertion logic are language and memory-model independent. We achieve this by using the notion of separation algebras by Calcagno et al. [5]. For a full exposition, see [2], but in a nutshell, as long as the user provides a memory model that satisfies the axioms of separation algebras, all of the following tactics can be applied.

sl_simpl This tactic attempts to simplify an entailment. All modifications to the goal are safe in the sense that the tactic will not make a goal unprovable. It assumes that the goal is in normal form, and given an entailment $t \vdash a$ does the following simplifications:

- Split a into a spatial component u and a propositional component P. Split the goal and simplify $t \vdash u$ and $t \vdash P$ independently.
- For each sub-formula of u, step through t to see if it is present there as well. If so, remove it from both assertions.
- Remove every sub-formula of P that is present in the Coq-context.
- If u contains an existential quantifier, replace it with an existential variable and rerun the simplification. However, this step rolls back unless the simplifier manages to solve the assertion under the binder completely; an incorrect guess of the existential variable can otherwise lead to an unprovable goal.

The reason that the goal is split in the first step, and before the spatial components are simplified, is that the spatial components are often needed to prove propositional assertions. If the spatial simplification is done before the split, the tactic can make the goal unprovable.

The sl_simpl tactic is parametric on another tactic that guides the simplifier when instantiating existential variables. This tactic dictates what safe instantiations are, i.e., which instantiations are allowed even if the entire assertion under the quantifier cannot be discharged by the tactic. As a default, this tactic is the fail-tactic; it will never succeed, and no instantiation is considered safe. However, for our Java-fragment, we allow the simplifier to instantiate existential variables if either they are checked for equality under the binder, or if they appear in the range of a pointsto-predicate. For instance, the entailment $o.f \mapsto v \vdash \exists x\ y.\ o.f \mapsto x * i.g \mapsto w \wedge x = y$ is simplified to $\mathit{true} \vdash i.g \mapsto w$ even though the goal is not solved completely.

sl_auto This tactic is a more aggressive version of sl_simpl. It unfolds the definition of the available representation predicates, and simplifies commonly occurring sub-expressions using rewriting tactics. Finally it runs sl_simpl. This heuristic can put the goal in an unprovable state, and the tactic will never be applied automatically by any of the other tactics.

sl_apply Standard tactics in Coq like apply or rewrite do not work in the desired way when reasoning with entailments. In Section 3, we use lemmas TreeRec_null and TreeRec_not_null to modify the proof goal, but neither the goal nor the lemmas are in a form that apply or rewrite can use in the intended way. The tactic sl_apply is designed to allow for forward reasoning in the following manner: Assume that we have a goal with an entailment in normal form $\overrightarrow{H : P} \to t \vdash a$ and a lemma $\mathcal{L}$ that we wish to apply that has the form $\forall \vec{x}.\ S_1\ \vec{x} \to \cdots S_n\ \vec{x} \to (b\ \vec{x} \vdash c\ \vec{x})$ where the variables $\vec{x}$ can be of any Coq type. The first step of the tactic is to replace all universally-quantified variables in $\mathcal{L}$ with existential variables and to split b and c into its spatial and propositional components, leaving the lemma in the form $S_1 \to \cdots S_n \to (u \wedge Q \vdash v \wedge R)$ where $b \dashv\vdash u \wedge Q$, $c \dashv\vdash v \wedge R$, and all quantifiers $\vec{x}$ have been replaced by existential variables that are free in u, v, Q, R, and S_1 to S_n. The next step is to frame u out of t, i.e., find a t' such that $t \dashv\vdash u * t'$. If this is successful, the tactic will leave the user to prove the following goals

$$\frac{\overrightarrow{H : P} \qquad H_1 : R}{v * t' \vdash a} \qquad \frac{\overrightarrow{H : P}}{t \vdash Q} \quad \frac{\overrightarrow{H : P}}{t \vdash S_1} \cdots \frac{\overrightarrow{H : P}}{t \vdash S_n}$$

where the first goal is the result of the original goal state after the application of $\mathcal{L}$, and the rest are the proofs of the propositional premises of $\mathcal{L}$. If the tactic is unable to find t', it will fail. The existential variables that are introduced in place of the quantifiers $\vec{x}$ are typically unified by Coq when t' is obtained or when Q or S_1 to S_n are proven. Uninstantiated variables are left in the goal. This behaviour is similar to the eapply-tactic in Coq.

4.2 Tactics on triples

Unlike entailments, the predicates in triples contain program variables. A typical triple can have the form $\{t \dot{*}\ u \dot{\wedge}\ p\}c\{b\}$ where $\dot{*}$ and $\dot{\wedge}$ are the lifted versions of $*$ and $\wedge$ respectively, as described in Section 2. One of the more common rules in separation logic is the rule of consequence, which allows the pre and post-conditions of a triple to be rewritten. Since triples operate on lifted assertions, and entailments operate on standard ones, our rule of consequence has a slightly different form than the standard one.

$$\frac{\forall s.\ (a\ s \vdash a'\ s) \qquad \{a'\}c\{b'\} \qquad \forall s.\ (b'\ s \vdash b\ s)}{\{a\}c\{b\}}\ \textsc{RoC}$$

By applying a stack s to the lifted assertions, we obtain standard assertions. Even though this rule exposes the stack s, it is only used in intermediate steps of the tactic and the user will never see an explicit stack.

sl_apply We extend the sl_apply-tactic from Section 4.1 to work on triples as well as entailments. The general idea is to allow forward-reasoning by rewriting the precondition of a triple using the rule of consequence and the sl_apply-tactic for entailment. However, the tactics described so far are not sufficient. To demonstrate, we attempt to rewrite the triple $\{(a \mathrel{\dot{-\!\!*}} b) \mathbin{\dot{*}} a \mathbin{\dot{*}} d\}c\{e\}$ to $\{b \mathbin{\dot{*}} d\}c\{e\}$ using a modus ponens rule for $-\!\!*$ that states that $\forall a\ b.\ (a -\!\!* b) * a \vdash b$. Remember that the connectives are not lifted in the lemmas we apply.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{b\ s * d\ s \vdash ?x\ s}\ ???}{(a\ s -\!\!* b\ s) * a\ s * d\ s \vdash ?x\ s}\ -\!\!*\text{–MP}}{\forall s.\ (((a \mathrel{\dot{-\!\!*}} b) \mathbin{\dot{*}} a \mathbin{\dot{*}} d)\ s \vdash ?x\ s)}\ \forall\text{–I} \qquad \dfrac{\vdots}{\{?x\}c\{?y\}} \qquad \dfrac{\dfrac{}{?y\ s \vdash e\ s}\ \textit{Refl.}}{\forall s.\ (?y\ s) \vdash (e\ s)}\ \forall\text{–I}}{\{(a \mathrel{\dot{-\!\!*}} b) \mathbin{\dot{*}} a \mathbin{\dot{*}} d\}c\{e\}}\ \textsc{RoC}$$

Applying the rule of consequence generates the existential variables $?x$ and $?y$ for the pre and the postcondition respectively. Instantiating $?y$ is straightforward, and follows immediately by reflexivity of $\vdash$. Instantiating $?x$ is more problematic. First, we introduce the stack s; the stack then propagates over the lifted connectives resulting in assertions with corresponding un-lifted connectives. Coq does this automatically. We then apply the modus-ponens lemma. To conclude, we need to unify $?x$ with $b \mathbin{\dot{*}} d$, in effect reversing the computation that un-lifted the connectives. This, however, Coq is not able to do automatically – the proof does go through if the user manually instantiates $?x$ but for large proofs this quickly becomes tedious. We require a tactic that will transform the assertion $b\ s * d\ s$ to $(b \mathbin{\dot{*}} d)\ s$. How we solve this is described in Section 5.2.

forward The forward-tactic is the work horse tactic of Charge!. Given a triple $\{p\}c_1; \cdots ; c_n\{q\}$ the tactic symbolically executes the command c_1, given that its requisites are met by the precondition p, rewriting p to a new predicate p'. The user is left to prove either the triple $\{p'\}c_2; \cdots ; c_n\{q\}$, or, if the triple initially had only one command, the entailment $p' \vdash q$. The tactic only works for goals in normal form.

charge The charge-tactic is the tactic that gives our framework its name. The tactic repeatedly applies the forward-tactic until either forward fails or provides the user with more than one subgoal to prove.

5 Tactic building blocks

We have several automatic heuristics that solve frequently occurring sub-goals of our tactics. The tactics sl_simpl and sl_apply use a framing tactic that attempts to find one occurrence of a spatial formula in another and remove that instance; the sl_apply-tactic for triples require a tactic that lifts all connectives of an assertion; finally, most tactics require that triples and entailments are in normal form, hence we have a tactic that transforms a goal to normal form. These tactics work using a combination of hint-databases and reflective tactics.

5.1 Framing

In order to frame the spatial assertion u out of t, we have to find a t' such that $t \dashv\vdash t' * u$. This is achieved by rewriting t modulo commutativity and associativity of $*$. Since we know that t is spatial, we do not have to cover the cases of there being any pure assertions or occurrences of standard conjunction in t. The first step is to define a predicate $\mathit{Frame}\ t\ u\ t' \triangleq t \dashv\vdash t' * u$ that holds if framing u out of t results in t'. This predicate is then inserted into the proof wherever framing is required. In the derivation

$$\dfrac{\mathit{Frame}\ t\ u\ ?x \qquad \dfrac{\vdots}{?x * u \vdash a}}{t \vdash a}$$

an existential variable $?x$ is introduced, and the job of the framing tactic is to find a solution for the predicate $\mathit{Frame}\ t\ u\ ?x$, instantiating $?x$ in the process. The following inference rules accomplish this, assuming that t is linear, which the normal form guarantees.

$$\dfrac{}{\mathit{Frame}\ t\ \mathit{true}\ t} \qquad \dfrac{\mathit{Frame}\ u\ t\ u'' \qquad \mathit{Frame}\ u''\ t'\ u'}{\mathit{Frame}\ u\ (t * t')\ u'} \qquad \dfrac{t = u}{\mathit{Frame}\ (t * t')\ u\ t'}$$

$$\dfrac{\mathit{Frame}\ t'\ u\ t''}{\mathit{Frame}\ (t * t')\ u\ (t * t'')} \qquad \dfrac{t = u}{\mathit{Frame}\ t\ u\ \mathit{true}}$$

We add these rules in a left-to-right priority order to a hint database. The Coq `auto`-tactic is then used to solve the predicate.

5.2 Lifting connectives

Coq will reduce any term in the form $(\dot{f}\ a_1\ \cdots\ a_n)\ s$ to $f\ (a_1\ s)\ \cdots\ (a_n\ s)$, but as is demonstrated in Section 4.2, we need a tactic to reverse this computation. Similarly to framing, we have a predicate $\mathit{Lift}\ a\ b \triangleq a = b$ that in effect is a wrapper for standard Leibniz-equality. This predicate is then inserted into the proof derivations where required. For instance, in Section 4.2 we need to lift the term $b\ s * d\ s$ to $(b \mathbin{\dot{*}} d)\ s$ when using the `sl_apply`-tactic in a triple. Inserting the *Lift*-predicate in the derivation accomplishes this

$$\dfrac{\mathit{Lift}\ (b\ s * d\ s)\ (?x\ s)}{\dfrac{b\ s * d\ s \vdash ?x\ s}{\cdots}}$$

and similarly to the *Frame*-predicate, the tactics instantiate the existential variable $?x$ when proving the predicate. We add the following inference rules to a hint database in order to prove occurrences of the *Lift*-predicate.

$$\dfrac{}{\mathit{Lift}\ (s\ x)\ ((ve\ x)\ s)} \qquad \dfrac{\mathit{Lift}\ a_1\ (x_1\ s) \quad \cdots \quad \mathit{Lift}\ a_n\ (x_n\ s)}{\mathit{Lift}\ (f\ a_1\ \cdots\ a_n)\ ((\dot{f}\ x_1\ \cdots\ x_n)\ s)} \qquad \dfrac{}{\mathit{Lift}\ a\ a}$$

The first rule reverse a variable lookup on the stack; the second lifts any n-ary function; the final one is the base case that will fire when everything is lifted as far as possible. In our formalisation, we have one hint for every arity of function. We also have hints for a few other cases, like un-applied substitutions, quantifiers, and if-then-else-statements.

5.3 Normal form

Most tactics require that the goal is in normal form. Recall that the normal form ensures that, whether the goal is a triple or an entailment, that all existential variables and propositional assertions in the precondition have been extracted to the Coq-context. We create a tactic that given an assertion a, obtains a spatial assertion t, a pure assertion p, and a propositional assertion P, such that $a \dashv\vdash \exists \vec{x}.\ (t\ \vec{x} \wedge p\ \vec{x}) \wedge P\ \vec{x}$ where neither t, p, or P, contain any existential quantifiers. When such an assertion is in the precondition of an entailment or a triple, extracting the existentially quantified variables and propositional assertions to the Coq context is straightforward. The tactic that converts an assertion to normal form is a mix of reflective tactics, and hint-databases.

A deep embedding of assertions We create a Galina term that represents an assertion in normal form. The normal form requires that all existentially quantified variables are at the top level – this means that we must reason about open terms in order to describe the spatial, pure, and propositional predicates that appear under the binders. In effect, these assertions will be n-ary functions where n is the number of free logical variables in the assertion. We parametrise the deep embedding with a list of types corresponding to the types that have been existentially quantified so far. That list of types is converted to a tuple and the n-ary assertions are converted to unary ones that take one member of this tuple type, rather than a sequence of members of each binding type.

The type $\mathit{exs}\ \mathit{Ts}$ takes a list of types Ts and returns a tupled version of that list ($\mathit{exs}\ [] = ()$, $\mathit{exs}\ [\mathbb{Z},\ \mathit{val},\ \mathit{bool}] = (\mathbb{Z},\ (\mathit{val},\ (\mathit{bool},\ ())))$, et c.). An open term is a tuple of three lists: ts, of type $\mathit{list}\ (\mathit{exs}\ \mathit{Ts} \to (\mathit{heap} \to \mathit{Prop}))$, and ps and Ps of type $\mathit{list}\ (\mathit{exs}\ \mathit{Ts} \to \mathit{Prop})$. Intuitively, ts is a list of spatial terms separated by the $*$-operator, and ps and Ps are lists of pure and propositional terms respectively, separated by the $\wedge$-operator. We will use the notation $\langle \mathit{ts},\ \mathit{ps},\ \mathit{Ps} \rangle$ for such a tuple and give it the type $\mathit{deep\_asn}\ \mathit{Ts}$, where Ts is the list of types that the components of the tuple are parametrised on. Note that the type of the lists for pure and propositional assertions are of the same type since we cannot distinguish between these types of assertions in the assertion logic. Finally, we create a function $[T]d$ that given a term d of type $\mathit{deep\_asn}\ (T :: \mathit{Ts})$ closes the term and returns a term of the type $\mathit{deep\_asn}\ \mathit{Ts}$. Moreover, we will use π_1 and π_2 to denote projection of the first and the second element out of tuples respectively. To demonstrate, the deep embedding of the assertion $\exists x\ y.\ o.f \mapsto x * i.g \mapsto w \wedge x = y$ is $[\mathit{val}][\mathit{val}]\langle [\lambda t.\ o.f \mapsto (\pi_1\ t),\ \lambda t.\ i.g \mapsto w]_*,\ [],\ [\lambda t.\ \pi_1\ t = \pi_1(\pi_2\ t)] \rangle$

Transforming an assertion to normal form The transformation of assertions to normal form is done using hint databases. The first step is to create an evaluation function *eval* of type $\mathit{deep\_asn}\ \mathit{Ts} \to \mathit{exs}\ \mathit{Ts} \to (\mathit{heap} \to \mathit{Prop})$ that given an open term in normal form and a tuple instantiating the free variables, returns an equivalent assertion. We will write $[\![d]\!]_x$ for $\mathit{eval}\ d\ x$. Next, we define the following assertion $\mathit{NF}\ d\ a\ x \triangleq [\![d]\!]_x \dashv\vdash a\ x$ that holds if $[\![d]\!]_x$ evaluates to $a\ x$, given two open assertions, one deeply embedded d and one shallowly embedded a, and a tuple x that instantiates the free variables of d and a. This assertion is then inserted into proof trees when an assertion needs to be transformed to normal form. For instance, the following derivation puts the precondition of a triple in normal form.

$$\dfrac{\dfrac{\dfrac{\dfrac{\mathit{NF}\ ?y\ (a\ s)\ ()}{a\ s \vdash [\![?y]\!]_()} \quad \dfrac{\mathit{Lift}\ ([\![?y]\!]_())\ (?x\ s)}{[\![?y]\!]_() \vdash ?x\ s}}{\dfrac{a\ s \vdash ?x\ s}{\forall s.\ (a\ s \vdash ?x\ s)}\ \forall\text{–I}}\ \mathit{Trans.} \quad \dfrac{\vdots}{\{?x\}c\{?y\}} \quad \dfrac{\dfrac{}{?z\ s \vdash b\ s}\ \mathit{Refl.}}{\forall s.\ (?z\ s \vdash b\ s)}\ \forall\text{–I}}{\{a\}c\{b\}}\ \textsc{RoC}$$

The evaluation order of the predicates is important. Proving the predicate $\mathit{NF}\ ?y\ (a\ s)\ ()$ instantiates $?y$, obtaining an assertion in normal form. Proving the predicate $\mathit{Lift}\ ([\![?y]\!]_())\ (?x\ s)$ in turn instantiates $?x$ and lifts all connectives, allowing us to prove the triple, but with the precondition in normal form.

The next step is to create the hint-database that proves occurrences of the *NF*-predicate. We create two merge functions *merge_nf_sc* and *merge_nf_and*, both of type $\mathit{deep\_asn}\ \mathit{Ts} \to \mathit{deep\_asn}\ \mathit{Ts} \to \mathit{deep\_asn}\ \mathit{Ts}$ and written with the infix operators $\circledast$ and $\mathbin{⩓}$ respectively. The merger functions and the evaluation function are designed such that the following inferences hold.

$$\frac{\mathit{NF}\ d_a\ a\ x \qquad \mathit{NF}\ d_b\ b\ x}{\mathit{NF}\ (d_a \circledast d_b)\ (\lambda y.\ a\ y * b\ y)\ x} \qquad \frac{\mathit{NF}\ d_a\ a\ x \qquad \mathit{NF}\ d_b\ b\ x}{\mathit{NF}\ (d_a \mathbin{⩓} d_b)\ (\lambda y.\ a\ y \wedge b\ y)\ x}$$

$$\frac{\forall y : T.\ \mathit{NF}\ d\ (\lambda z.\ a\ (\pi_1\ z)\ (\pi_2\ x))\ (y,\ x)}{\mathit{NF}\ ([T]d)\ (\lambda z.\ \exists y : T.\ a\ y\ z)\ x} \qquad \frac{}{\mathit{NF}\ \langle [t],\ [],\ [] \rangle\ t\ x}$$

$$\frac{}{\mathit{NF}\ \langle [],\ [p],\ [] \rangle\ p\ x} \qquad \frac{}{\mathit{NF}\ \langle [],\ [],\ [P] \rangle\ P\ x}$$

The design of the merging functions is a bit intricate. When merging the terms d_a and d_b, their top level existential quantifiers are traversed and added in sequence to the resulting term. The difficulty comes when merging the two open terms – this requires a bit of work, and is left out for space reasons.

Recall that we cannot distinguish between pure and propositional assertions in the assertion logic, i.e. $[\![\langle [],\ [p],\ [] \rangle]\!]_x \dashv\vdash [\![\langle [],\ [],\ [p] \rangle]\!]_x$. When turning a propositional assertion to normal form, the tactics will check if there syntactically exists a stack in the assertion – if so, it is classified as pure, otherwise as propositional. It is important that this classification is correct or a pure assertion can end up in the Coq-context, rather than in the precondition of a triple.

6 Custom Hoare triples

The core of Charge! is designed to be language independent – the majority of the tactics are usable regardless of language or memory model. Once a language is defined, and all meta-theoretical properties have been proven, adapting the language to Charge! is relatively straightforward. In this section we demonstrate how to incorporate a standard read-rule from separation logic into Charge!. In separation logic, the Hoare-triple for the read rule often has the form

$$\frac{x \neq y \qquad x \notin fv\ e}{\{y.f \mapsto e\} x := y.f \{y.f \mapsto e \wedge x = e\}}\ \textsc{Read}$$

and stores the value of the expression e found at the memory location $y.f$ in the program variable x. There is also a side condition stating that x must not be a free variable in e and disjoint from y. This rule provides a minimal footprint of the read-command, but is often not directly usable as it requires the goal to be in a very specific form.

In [2] we provided an alternative read-rule that does not require the precondition to be of a certain shape, or impose any freshness conditions on x.

$$\frac{a \vdash y.f \mapsto e}{\{a\} x := y.f \{\exists v.\ a\{^{v}/_{x}\} \wedge x = e\{^{v}/_{x}\}\}}\ \textsc{ReadEnt}$$

When this rule is used, we need to prove that $y.f \mapsto e$ can be inferred from a. This is typically done by framing $y.f \mapsto e$ out of a. The substitutions perform the alpha-renamings required to enforce the side-conditions of the Read-rule. We use the tactics from Section 5 to create a new version of the rule that assumes that the goal is in normal form before it is applied, and ensures that the goal stays in normal form.

$$\frac{\begin{array}{l}\forall s.\ \exists u\ v\ ps.\ (p\ s \to \mathit{Frame}\ (a\ s)\ ((s\ y).f \mapsto v)\ u) \wedge \\ \qquad \mathit{Lift}\ v\ (e\ s) \wedge \mathit{PureBase}\ (\lambda t.\ (p\{^{\pi_1\ t}/_{x}\})\ s)\ ps \wedge \\ \qquad \left[\!\!\left[[val]\langle [(\lambda t.\ a\{^{\pi_1\ t}/_{x}\}\ s)]_*, \\ \qquad\quad [(\lambda t.\ s\ x = e\{^{\pi_1\ t}/_{x}\}\ s) :: ps,\ []\rangle \right]\!\!\right]_{()} = ?b\ s\end{array}}{\{a \wedge p\} x := y.f \{?b\}}\ \textsc{ReadNF}$$

This rule is a bit intimidating, but solvable by the tactics described so far. When applied, the quantifiers are introduced and existential variables created for the existentially quantified variables. We assume that the postcondition of the triple is an existential variable. If this is not the case, the rule of consequence can be used to obtain an existential variable for the postcondition. The *Frame*-predicate corresponds to the premise of the ReadEnt-rule, and the pure facts p can be used when proving the predicate. The range of the pointsto-predicate v, which is instantiated when *Frame* is proven, is then lifted using the *Lift*-predicate instantiating the expression e in the process. The predicate *PureBase p ps* instantiates *ps* to the empty list if p reduces to *true*, and $[p]$ otherwise – this is to avoid cluttering up the precondition with *true*-predicates. The evaluation of the normal form evaluates to the postcondition of ReadEnt. All of our triple-rules are in a similar form.

7 Related Work

Adapting proof assistants to reason in separation logic has been proposed before. Some of the earliest work is an unpublished article by Appel [1] where he creates a family of tactics that reasons about a small imperative language. The core philosophy is the same as ours – the user should be able to reason in the separation logic, not in its model, and there should never be an explicit stack or a heap in the proof context.

In later work, McCreight expanded on Appel's ideas and created a comprehensive set of tactics for verifying Cminor programs using separation logic in Coq [11]. This seminal piece of work drastically cut down proof script sizes, yet their approach differs from ours. In McCreight's work, the user will find the heap in the proof context. Where we have an entailment of the form $a * b \vdash c * d$, McCreight unfolds the definition of entailment exposing the heap $(a * b)\ m \rightarrow (c * d)\ m$, and the antecedent of the implication is then moved to the Coq context. The reason this approach works fine is that the definition of $*$ is never unfolded and even though the heap is exposed, the user never has to reason about sub-heaps or their disjointness-properties. One of the main motivations of our work was that we wanted to see whether or not it is possible to retain the simplicity of McCreight's tactics while keeping with the overall philosophy of Appel's ideas and strictly reason inside the separation logic. We claim that we have achieved this. One point of comparison is that all three formalisations have verified the standard in-place list reversal algorithm. In [1], Appel uses 200 lines and 795 words to verify this program by count of `wc`; McCreight uses 68 lines and less than 400 words [11]. We use 25 lines and 105 words. These numbers do not include the definition of the program, just the proofs themselves.

Other work includes the Bedrock framework by Chlipala [6]. Similar to Charge!, Bedrock strives to automate the tedium of program verification using separation logic in Coq. The focus on Bedrock lies on low level languages, including support to work with hardware registers.

Another interactive approach is Holfoot, by Tuerk [16], which verifies Smallfoot specifications inside HOL4. This approach is similar to ours in that the core of Holfoot also builds on the theories of abstract separation algebras by Calcagno et al. [5]. Holfoot also has impressive automation results, but to the best of our knowledge does not handle object-oriented programs or nested triples [14].

Another Coq-framework for separation logic by Dockins et al. is the MSL-library [7]. It provides extensive meta-theoretical results of separation algebras of different flavours, however it currently has very few tactics.

One very prominent tool for program verification is VeriFast [9]. VeriFast allows the user to write, specify, and compile C and Java-programs and prove their correctness. The approach is mostly interactive and, as the name suggests, fast. However, the tool provides no formal proof of program correctness. The binary tree example presented in Section 3 is taken from the VeriFast web-site.

8 Conclusion

We have developed Charge! – a comprehensive framework for verifying the correctness of Java-like programs using higher-order separation logic in Coq. Our tactics allow the user to focus on the actual program verification, as opposed to manually proving all of the tedious and repetitive steps that proofs of full functional program correctness typically require. Moreover, the work-flow is very close to the style of reasoning used for pen-and-paper proof outlines in separation logic, allowing users to freely exchange logical variables and program variables in the assertion logic predicates. Charge! is memory-model independent, and the modular design of the tactics makes adding new language features and commands simple and straightforward.

References

1. A. W. Appel. Tactics for separation logic, Draft of January 2006. http://www.cs.princeton.edu/∼appel/papers/septacs.pdf.
2. J. Bengtson, J. Jensen, F. Sieckowski, and L. Birkedal. Verifiying object-oriented programs with higher-order separation logic in Coq. In *Proceedings of ITP*, 2011.
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. C. Calcagno, P. W. O'Hearn, and H. Yang. Local action and abstract separation logic. In *Proceedings of LICS*, pages 366–378, 2007.
6. A. Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *PLDI*, pages 234–245, 2011.
7. R. Dockins, A. Hobor, and A. W. Appel. A fresh look at separation algebras and share accounting. In *The 7th Asian Symposium on Programming Languages and Systems*, pages 161–177. Springer ENTCS, 2009.
8. M. Dodds, S. Jagannathan, and M. J. Parkinson. Modular reasoning for deterministic parallelism. In *Proceedings of POPL*, 2011.
9. B. Jacobs and F. Piessens. The verifast program verifier. CW Reports CW520, Department of Computer Science, K.U.Leuven, August 2008.
10. N. R. Krishnaswami, J. Aldrich, L. Birkedal, K. Svendsen, and A. Buisse. Design patterns in separation logic. In *Proceedings of TLDI*, pages 105–116, 2009.
11. A. McCreight. Practical tactics for separation logic. In *Proceedings of TPHOLs*, pages 343–358, 2009.
12. 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.
13. M. J. Parkinson, R. Bornat, and C. Calcagno. Variables as resource in Hoare logic. In *Proceedings of LICS*, pages 137–146. IEEE, 2006.
14. J. Schwinghammer, L. Birkedal, B. Reus, and H. Yang. Nested Hoare triples and frame rules for higher-order store. In *Proceedings of CSL*, 2009.
15. K. Svendsen, L. Birkedal, and M. Parkinson. Verifying generics and delegates. In *Proceedings of ECOOP*, pages 175–199, 2010.
16. T. Tuerk. A Formalisation of Smallfoot in HOL. In *In proceedings of TPHOLs*, LNCS, pages 469–484, 2009.
17. C. Varming and L. Birkedal. Higher-order separation logic in Isabelle/HOLCF. *Electr. Notes Theor. Comput. Sci.*, 218:371–389, 2008.