Separation Logic in an Imperative Language

Sune Alkærsig and Thomas Hallier Didriksen
{sual, thdi}@itu.dk

IT University of Copenhagen, Rued Langgaards Vej 7, 2300 Copenhagen S, Denmark

Abstract. We present an extension of a simple imperative programming language called Imp, for which Hoare rules for basic operations have already been defined. The extension is based on an addition of an addressable and mutable state, realized as a heap, which currently does not exist for Imp. We define syntax, semantics, and Hoare rules for commands operating on such a heap, as well as redefine the logic in which Imp programs are verified. Verification using the defined Hoare rules is demonstrated through the verification of an example program. All the formalizations presented are verified with the Coq proof assistant.

Keywords: Coq, Separation Logic, Imp, Hoare Logic

1 Introduction

We investigate how to add shared mutable state to an imperative programming language, Imp, and how to verify Imp programs using this type of state using separation logic.

Separation logic [13] is an extension of Hoare logic, which is devised for local reasoning about shared, mutable data structures. Reasoning about shared data structures in classic Hoare logic is complicated by the fact that when developing programs, we often assume that data structures do not share memory, but when proving the same programs correct, we assume that they do and have to prove otherwise [14]. Separation logic alleviates this situation by allowing us to specify a program in terms of the data storage it modifies, abstracting from the parts of the storage that are unaffected by the program.

The Imp language is a simple imperative language, in which only the core constructs of fully-fledged languages, such and C and Java, are defined. These include variable assignment, sequencing of commands, conditional statements, and while loops. It does not, however, have a mutable state, making most data structures unnecessarily complex to define and cumbersome to work with. Extending the language with a shared mutable state adds to the expressiveness of the language, but also requires an extension of the logic used for verifying Imp programs.

Problem. The problem is twofold. It is 1) extending the intuitionistic Hoare logic in which Imp programs are verified with a separation logic and 2) defining new language constructs for the Imp language for manipulating the shared mutable state, defined as a heap, together with Hoare rules for these.

Contributions. Our contribution is a solution for the second part of the problem. We define syntax and semantics for the four essential contructs enabling utilization of a mutable state: read, write, allocation, and deallocation[1]. To be able to reason about and verify programs written in the Imp language involving these constructs, we define Hoare rules for each of them. The soundness of these rules have been proven with Coq, an interactive theorem prover. Furthermore, we define and prove the frame rule and rules of consequence for use in correctness proofs of Imp programs.

Dependencies. For the first part of the problem, we rely on the work on constructing separation logics from separation algebras done by Jesper Bengtson et al. [2], which will be introduced in Section 3.3. For the second part we use a reimplementation of maps from the standard Coq library Stéphane Lescuyer [9] to model the heap. The language we are extending, Imp, is defined in the book "Software Foundations" by Benjamin Pierce et al. [12], which is also our source for certain simple, but useful theorems about booleans and arithmetics [11].

Coq. Coq is a formal proof assistant which is designed for developing formal specifications and formal proofs in higher-order logics [6]. We use Coq for extending the Imp language with additional constructs, and for writing and verifying formal specifications for these constructs.

Outline. In Section 2 we present syntax and semantics for the Imp language, as well as the extended program state. We introduce Hoare logic as a formal system for program verification in Imp in Section 3.1 and extend this system with separation logic in Section 3.3. In this section we also introduce the Hoare rules for the heap-manipulating operations. In Sections 3.5 and 3.6 we discuss auxiliary rules. An example program verified using our constructs is presented in Section 4. We reflect upon related work in Section 5 and conclude on our findings in Section 6.

2 The Imp Language

In this section we describe the Imp language and its features. The parts of Imp originating from its initial implementation will be presented [12] alongside our additions. We will refer to the unmodified version of Imp as *the basic version of Imp*, and the version including our modifications as *the extended version of Imp*.

[1] The source code can be found at https://github.com/tdidriksen/SASP-Project

2.1 The Program State

Values at a point of execution Imp are represented by a *state*. In both the basic and the extended version of Imp, this state is a partially applied function $id \to value$, where id identifies a variable.

Definition 1 (State) *The state is a function from identifiers to values, which is defined as* $state \triangleq id \to value$.

We extend the storage in Imp with a mutable state in the form of a heap. For modeling this heap, we use maps from the Coq Containers library [9], which allow us to map addresses to values. For the sake of simplicity, we will consider the address space of the heap to be infinite, which means that a program cannot run out of memory.

Definition 2 (Heap) *The heap is a partial function mapping addresses to values, which is specified* $heap \triangleq a \to v$, *where* $a,\ v \in \mathbb{N}$.

To avoid terminological confusion, Definition 1 will henceforth be referred to as *the state*, Definition 2 as *the heap*, and the combination of the two as *the program state*.

Functions operating on the heap. For lookup on the heap we will be using the existing $find$ function from the implementation of maps [9]. Find returns a value wrapped in Coq's standard Option type [5], meaning that if the key exists in the map it will return $Some(value)$, otherwise it will return $None$.

Definition 3 (Find) *Find is a function which given a key k and a map m, returns the value at key k wrapped in an option type. Find is defined as* $find \triangleq key \to map \to option\ value$.

In our case, the key and value are natural numbers, because of our definition of the heap.

The following heap-manipulating functions underlying the implementation of the heap operations in Imp are used for specifying the semantics discussed in Section 2.3.

Definition 4 (write) *write is a function which given a value v, an address a, and a heap h, returns h, extended with a mapping from a to v, such that lookup on a results in v. Write is defined as* $write \triangleq a \to v \to heap \to heap$.

Definition 5 (dealloc) *dealloc is a function which given an address a and a heap h, returns h, with no mapping from a to v, such that lookup on a fails. Deallocate is defined as* $dealloc \triangleq a \to heap \to heap$.

Definition 6 (alloc) *alloc is a function which given an address a, a natural number of cells to allocate n, and a heap h, returns h extended with a mapping from a to 0, a+1 to 0 $\cdots$ a+n-1 to 0, such that lookup on a, a+1 $\cdots$ a+n-1 all result in 0. allocate is defined as* $alloc \triangleq a \to n \to heap \to heap$.

2.2 Grammar

```
program = com ;
com =   "SKIP"
      | id "::=" aexp            (* ASSIGNMENT *)
      | com ";" com
      | "IFB" bexp "THEN" com "ELSE" com "FI"
      | "WHILE" bexp "DO" com END
      | id ⇜ "[" aexp "]"        (* READ *)
      | "[" aexp "]" ⇜ id        (* WRITE *)
      | "DEALLOC" aexp           (* DEALLOCATE *)
      | id "&=" "ALLOC" nat ;    (* ALLOCATE *)
aexp =  "ANum" nat
      | "AId" id
      | "APlus" aexp aexp
      | "AMinus" aexp aexp
      | "AMult" aexp aexp ;
bexp =  "BTrue"
      | "BFalse"
      | "BEq" aexp aexp
      | "BLe" aexp aexp
      | "BNot" bexp
      | "BAnd" bexp bexp ;
id = "Id" nat ;
nat = natural number, 0 included ;
```

Fig. 1. Full grammar for the extended Imp language, specified in EBNF.

The full grammar for the extended Imp language, including the added operations for manipulating the heap, namely read, write, deallocate, and allocate, is shown in Figure 1. Besides these, we have skip, assignment, sequencing, conditional statements, and while loops from the basic version of Imp. Basically, any Imp program consists of a command, arithmetic expressions on natural numbers, and boolean expressions. All variables, implemented as identifiers on natural numbers (*id*), are evaluated with the *state*. Since the domain of the heap is the natural numbers, an address on the heap is accessed by evaluating any *aexp* representing a memory address to a natural number with the state.

2.3 Semantics

We now define operational semantics for the operations of the Imp language. Four operations have been added for manipulating the heap: Read, Write, Allocate, and Deallocate.

Notation. The operational semantics are specified using the following notation:

$$\frac{Preconditions}{\langle (c),\ (s,\ h)\rangle \Downarrow Some\ (s',\ h')}\ \textsc{Title}$$

Read this notation as follows: Given that the preconditions hold (if any), then evaluating the command c in a program state with a state s and heap h will result in a program state with a state s' and a heap h'. Due to the error semantics explained later in this section, the resulting program state is wrapped in an Option type [5]. Thus, any faulty program state is represented by $None$, while any normal program state is represented by $Some\ (state,\ heap)$.

Updating a state is described using this notation:

$$[s \mid X : n]$$

This means that the state s is updated with a variable X with the value n. If X already exists in the state, the new value for X will shadow the existing value.

Lastly, evaluating a boolean is described using this notation:

$$b \Downarrow BTrue \qquad b \Downarrow BFalse$$

The left one means that b evaluates to $true$, and the right one that b evaluates to $false$.

Semantics For completeness, the semantics for all the operations of the extended Imp language are given in Figure 2. We will, however, not discuss the semantics of the operations already existing in the basic version of Imp, as these have not been altered. A discussion of these can be found in "Software Foundations" [12].

Regarding the operations on the heap, there are a few interesting points to note. Firstly, the semantics of allocation and deallocation are asymmetric. This asymmetry becomes evident when we examine the semantics of the two operations more closely. The parameter a of the Deallocate operation represents an arithmetic expression denoting the address of the cell we wish to deallocate. In contrast, the parameter n of the Allocate operation denotes the number of cells we wish to allocate on the heap. Hence, we can allocate n cells with one invocation of Allocate, but deallocating n cells requires n invocations of Deallocate. This asymmetry makes deallocation easier to reason about, but places a heavier burden of memory management on the user of the language.

Secondly, locating free addresses in the address space for the allocation operation is not a part of the allocation semantics, although the semantics do state that these addresses must be consecutive and not already allocated. Here, we

$$\frac{}{\langle \mathbf{SKIP},\ (s,\ h)\rangle \Downarrow \mathit{Some}\ (s,\ h)}\ \textsc{Skip}$$

$$\frac{}{\langle (X\ ::=\ n),\ (s,\ h)\rangle \Downarrow \mathit{Some}\ ([s \mid X : n],\ h)}\ \textsc{Assignment}$$

$$\frac{\langle c1,\ (s,\ h)\rangle \Downarrow \mathit{Some}\ (s',\ h') \quad \langle c2,\ (s',\ h')\rangle \Downarrow \mathit{Some}\ (s'',\ h'')}{\langle c1; c2,\ (s,\ h)\rangle \Downarrow \mathit{Some}\ (s'',\ h'')}\ \textsc{Sequence}$$

$$\frac{b \Downarrow \mathit{BTrue} \qquad \langle c1,\ (s,\ h)\rangle \Downarrow \mathit{Some}\ (s',\ h')}{\langle (\mathbf{IFB}\ b\ \mathbf{THEN}\ c1\ \mathbf{ELSE}\ c2\ \mathbf{FI}),\ (s,\ h)\rangle \Downarrow \mathit{Some}\ (s',\ h')}\ \textsc{IfTrue}$$

$$\frac{b \Downarrow \mathit{BFalse} \qquad \langle c2,\ (s,\ h)\rangle \Downarrow \mathit{Some}\ (s',\ h')}{\langle (\mathbf{IFB}\ b\ \mathbf{THEN}\ c1\ \mathbf{ELSE}\ c2\ \mathbf{FI}),\ (s,\ h)\rangle \Downarrow \mathit{Some}\ (s',\ h')}\ \textsc{IfFalse}$$

$$\frac{\begin{array}{c} b \Downarrow \mathit{BTrue} \qquad \langle c,\ (s,\ h)\rangle \Downarrow \mathit{Some}\ (s',\ h') \\ \langle (\mathbf{WHILE}\ b\ \mathbf{DO}\ c\ \mathbf{END}),\ (s',\ h')\rangle \Downarrow \mathit{Some}\ (s'',\ h'') \end{array}}{\langle (\mathbf{WHILE}\ b\ \mathbf{DO}\ c\ \mathbf{END}),\ (s,\ h)\rangle \Downarrow \mathit{Some}\ (s'',\ h'')}\ \textsc{WhileLoop}$$

$$\frac{b \Downarrow \mathit{BFalse}}{\langle (\mathbf{WHILE}\ b\ \mathbf{DO}\ c\ \mathbf{END}),\ (s,\ h)\rangle \Downarrow \mathit{Some}\ (s,\ h)}\ \textsc{WhileEnd}$$

$$\frac{\mathit{heap}(a) = n}{\langle (X \leftsquigarrow [a]),\ (s,\ h)\rangle \Downarrow \mathit{Some}\ ([s \mid X : n],\ h)}\ \textsc{Read}$$

$$\frac{\exists e.\mathit{heap}(a) = e}{\langle ([a] \rightsquigarrow b),\ (s,\ h)\rangle \Downarrow \mathit{Some}\ ([s \mid X : a],\ \mathit{write}\ a\ b\ h)}\ \textsc{Write}$$

$$\frac{\exists e.\mathit{heap}(a) = e}{\langle (\mathbf{DEALLOC}\ a),\ (s,\ h)\rangle \Downarrow \mathit{Some}\ (s,\ \mathit{dealloc}\ a\ h)}\ \textsc{Deallocate}$$

$$\frac{\neg\exists e.\mathit{heap}(a) = e \qquad \forall n'.\ (n' > a \wedge n' \leq a + n) \Rightarrow \neg\exists e'.\mathit{heap}(n') = e'}{\langle (X\ \&\!\!= \mathbf{ALLOC}\ n),\ (s,\ h)\rangle \Downarrow \mathit{Some}\ ([s \mid X : a],\ \mathit{alloc}\ a\ n\ h)}\ \textsc{Allocate}$$

Fig. 2. Semantics for the operations in Imp.

simply assume that such consecutive and free addresses exist on the heap. The problem of finding such addresses is addressed by the soundness proof of the corresponding Hoare rule for allocation, presented in Section 3.4.

Lastly, it should be noted that we have no notion of void types and do not provide any measures for ignoring evaluated values. Consequently, providing a variable in which to store the result of a Read or Allocate operation is required. For Allocate, this also has the benefit of safeguarding against allocating unreachable memory, which would be unfortunate in an environment without garbage collection.

Error Semantics An interesting consequence of adding an addressable heap to a programming language is the possibility of writing programs that evaluate to a faulty program state. Contrary to the basic version of the Imp language, an addressable heap enables the user to write programs that type check, but evaluate to an erroneous program state: When reading, writing, or deallocating, the program might end up in a faulty program state if attempting to access an inactive address on the heap. Note that because we assume that the address space of the heap is infinite, allocation cannot fail. The error semantics are presented in Figure 3.

$$\frac{\neg\exists e.\mathit{heap}(a) = e}{\langle (X \leftsquigarrow [a]),\ \mathit{Some}\ (s,\ h)\rangle \Downarrow \mathit{None}}\ \textsc{ReadError}$$

$$\frac{\neg\exists e.\mathit{heap}(a) = e}{\langle ([a] \leftsquigarrow b),\ \mathit{Some}\ (s,\ h)\rangle \Downarrow \mathit{None}}\ \textsc{WriteError}$$

$$\frac{\neg\exists e.\mathit{heap}(a) = e}{\langle (\textbf{DEALLOC}\ a),\ \mathit{Some}\ (s,\ h)\rangle \Downarrow \mathit{None}}\ \textsc{DeallocateError}$$

Fig. 3. Error semantics for the Read, Write, and Deallocate operations.

3 Verifying Imp Programs

An interesting property of the Imp language is that any Imp program can be formally verified directly within the Coq environment. This section introduces Hoare logic, the formal system within which programs of the basic version of Imp are verified, together with its main concepts. Additionally, we introduce separation logic, an extension to Hoare logic, and define the rules required for reasoning about the operations of the extended Imp language.

3.1 Hoare Logic

Hoare logic [7] is a formal system for program verification. It formalizes the evaluation of a program using two types of assertions: preconditions and postcondtions. A precondition is an assertion that holds prior to program execution, and a postcondition is an assertion that holds after program execution.

In Hoare logic, only partial correctness of a program is proven. This means that we only prove the correctness of a program in the event that the program terminates. Proving that a program terminates is a separate activity, which we will not consider here.

The Hoare Triple The Hoare triple is one of the central concepts of Hoare logic.

Definition 7 (Original Hoare Triple) *If a command c is started in a program state satisfying an assertion P, and if c eventually terminates, then the resulting program state is guaranteed to satisfy the assertion Q* [12].

Note that in the case that c does not terminate, we can not say anything about Q. Below is a formalization of Definition 7.

$$\forall st\ st'.\ \langle c,\ st\rangle\ \Downarrow Some\ st' \Rightarrow P\ st \Rightarrow Q\ st'$$

The addition of a heap to the program state, however, renders the above formalization insufficient. As discussed in Section 2.3, some of the heap-manipulating operations can evaluate to a faulty program state. This is an issue, since the Hoare triple claims that if c terminates, the resulting program state is guaranteed to satisfy the postcondition. With Definition 7, this can no longer be guaranteed if the resulting program state is faulty. As such, we need our Hoare triple to express that the command c does not evaluate to an erroneous program state.

Definition 8 (Redefined Hoare Triple) *If a command c is started in a program state st satisfying an assertion P, if c eventually terminates, and if c does not fail in st, then the resulting program state is guaranteed to satisfy the assertion Q. We use the following notation for the Hoare triple:* $\{P\}\ c\ \{Q\}$

To express the idea of a command not failing in a given program state, we introduce the notion of safety. A command is safe in a program state st if the command, starting in st, does not evaluate to $None$. We therefore define safety as follows:

$$safe\ c\ st\ :\ \neg\ (\langle c,\ st\rangle\ \Downarrow None)$$

Having formalized the safety predicate, we can now formalize Definition 8 as follows:

$$\forall st.\ P\ st \Rightarrow safe\ c\ st \wedge (\forall st'.\langle c,\ st\rangle \Downarrow Some\ st' \Rightarrow Q\ st')$$

Any reference to *the Hoare triple* in subsequent sections will refer to the Hoare triple as defined in Definition 8.

3.2 Hoare Rules — Basic Version of Imp

The Hoare rules for the operations found in the basic version of Imp are shown in Figure 4. Because these are all defined in "Software Foundations", we will not go through the justification for each of them. As an example, however, we will consider the justification for the Assignment rule.

Let us think of the Q in the Assignment rule as any property of natural numbers, such as "being equal to 1", and note the fact that a is an arithmetic expression on natural numbers. This means that if we have $Q(a)$, that Q holds for a, we can state that "a is equal to 1". Then the assignment command, $X ::= a$, transfers this property of a to X, so that we are now able to state that "X is equal to 1", or $Q(X)$. Rephrasing this intuition, we can say that if we substitute all occurrences of X in Q for a in the precondition, and we know that $Q(a)$ holds, then when we transfer the property of Q to X, Q must hold in the postcondition, since $Q(a)$ held in the precondition.

Taking a closer look at the Assignment rule, we realize that it only holds for assigning a variable to the state, and not the heap. In fact, we have no way of expressing concise propositions about the heap inside a Hoare rule. For stating propositions about the heap, we turn to separation logic.

$$\frac{}{\{Q\{^{a}/_{X}\}\}\ X ::= a\ \{Q\}}\ \textsc{Assignment} \qquad \frac{}{\{P\}\ \textbf{SKIP}\ \{P\}}\ \textsc{Skip}$$

$$\frac{\{P\}\ c1\ \{Q\}\ \ \{Q\}\ c2\ \{R\}}{\{P\}\ c1; c2\ \{R\}}\ \textsc{Sequence} \qquad \frac{\{P \wedge b\}\ c\ \{P\}}{\{P\}\ \textbf{WHILE}\ b\ \textbf{DO}\ c\ \textbf{END}\ \{P \wedge \neg b\}}\ \textsc{While}$$

$$\frac{\{P \wedge b\}\ c1\ \{Q\}\ \ \{P \wedge \neg b\}\ c2\ \{Q\}}{\{P\}\ \textbf{IFB}\ b\ \textbf{THEN}\ c1\ \textbf{ELSE}\ c2\ \textbf{FI}\ \{Q\}}\ \textsc{If}$$

Fig. 4. Hoare rules of the basic version of the Imp language.

3.3 Separation Logic

Separation logic is an extension of Hoare logic that enables local reasoning about shared resources. While its original purpose was to provide assertions for reasoning about shared mutable data structures, such as heaps, it has now been applied to a broader range of subject areas, such as shared-variable concurrency [14].

The Connectives For separation logic, three new connectives are added to those already known from Hoare logic.

The separating conjunction. The most characteristic connective of separation logic is the *separating conjunction*, denoted by $*$. A separating conjunction $p * q$ on a heap h is a spatial connective, which states that h can be split into two separate (or disjoint) parts, one for which p holds and one for which q holds. Note that $p * q$ holds for the entirety of h, meaning that there is no third part of h for which a third predicate r would hold.

emp. emp is the unit of the separating conjunction, and only holds for the empty heap. Consequently, since *emp* only holds for the empty heap, $p * emp$ is logically equivalent to p.

The magic wand. Despite its name, there is nothing magic about the magic wand. Also known as the *separating implication*, $p \mathrel{-\!\!*} q$ says that if a heap is extended with a separate part for which p holds, then q holds for that extended heap.

The points-to relation. Aside from the new connectives, separation logic uses the *points-to* relation. Denoted by $e \mapsto e'$, it states that e points to e' on the heap, and nothing else is on the heap. As such, the points-to relation both says something about the contents and the size of the heap. Given $x \mapsto x' * y \mapsto y'$, we know that in one part of the heap, x points to x', and in another part of the heap, y points to y', and nothing else is on the heap.

Local Reasoning and the Frame Rule One of the key points of separation logic is the ability to reason locally about a shared mutable resource, which in our case is a heap. Local reasoning is achieved by specifying a program in terms of the heap cells it accesses, and then using the frame rule as described by Yang and O'Hearn [15] to abstract away the rest of the heap.

$$\frac{\{P\}\ c\ \{Q\}}{\{P\ *\ R\}\ c\ \{Q\ *\ R\}}$$

where no variable occurring free in R is modified by c.

Fig. 5. The standard definition of the frame rule.

If we have a program c with the precondition P and the postcondition Q, we can abstract away the rest of the heap R, as we know that c does not modify any part of R.

Separation Algebra

Definition 9 (Separation Algebra) *A separation algebra is a partial, commutative monoid (Σ, $\circ$, u) where Σ is the carrier type, $\circ$ is the binary operator, and u is the unit element* [4].

We can define heaps in terms of a separation algebra by taking Σ to be a type of heaps, $\circ$ to be a composition operator for two disjoint heaps, and u to be *emp*, the unit of the separating conjunction [2].

$$\frac{}{A \vdash \top} \qquad \frac{}{\bot \vdash A} \qquad \frac{A\ x \vdash B}{\forall A, A \vdash B}$$

$$\frac{\forall x, B \vdash A\ x}{\forall A, B \vdash A} \qquad \frac{\forall x, A\ x \vdash B}{\exists A, A \vdash B} \qquad \frac{B \vdash A\ x}{\exists A, B \vdash A}$$

$$\frac{A \vdash C}{A \wedge B \vdash C} \qquad \frac{B \vdash C}{A \wedge B \vdash C} \qquad \frac{A \vdash B}{A \vdash B \vee C}$$

$$\frac{A \vdash C}{A \vdash B \vee C} \qquad \frac{A \vdash C \quad B \vdash C}{A \vee B \vdash C} \qquad \frac{A \vdash B \quad A \vdash C}{A \vdash B \wedge C}$$

$$\frac{A \vdash (B \Rightarrow C)}{A \wedge B \vdash C} \qquad \frac{A \wedge B \vdash C}{A \vdash (B \Rightarrow C)}$$

Fig. 6. The axioms of our intuitionistic logic.

$$\frac{}{P * Q \vdash Q * P} \qquad \frac{}{(P * Q) * R \vdash P * (Q * R)} \qquad \frac{P * Q \vdash R}{P \vdash Q \mathrel{-\!\!*} R}$$

$$\frac{P \vdash Q \mathrel{-\!\!*} R}{P * Q \vdash R} \qquad \frac{P \vdash Q}{P * R \vdash Q * R} \qquad \frac{}{P * emp \vdash P}$$

$$\frac{}{P \vdash P * emp}$$

Fig. 7. The axioms of our separation logic.

Building the Separation Logic This section describes how we have used prior work by Jesper Bengtson et al. [2] to build a separation logic. Here, an intuitionistic logic is defined as a logic for which the axioms presented in Figure 6 hold. Accordingly, a separation logic is defined as a logic for which the axioms presented in Figure 7 hold.

Purpose. The purpose of building the separation logic is to be able to create assertions $heap \to state \to Prop$ for use in program specifications that need to make claims about both a state and a heap.

Procedure. The basis of our separation logic is *Prop*, the sort for propositions in Coq [6]. It is established that Prop is an intuitionistic logic by proving that it satisfies all the axioms of an intuitionistic logic. To further develop the intuitionistic logic of *Prop*, we exploit the following properties of intuitionistic logic:

1) We can to create an intuitionstic logic from a function space $A \to B$, where B is the carrier of an intuitionistic logic and A is any type.
2) We can create a separation logic from a function space $A \to B$, where A is the carrier of a separation algebra and B is the carrier of an intuitionistic logic. [3]

By 1), we can extend the intuitionistic logic of *Prop* with a state to achieve an intermediate logic $state \to Prop$. This will allow us to create assertions about programs using a state. Note that all of the connectives defined for the logic of *Prop* now work on assertions of $state \to Prop$, as these are just pointwise liftings.

As we have already established, we can define heaps in terms of a separation algebra, we extend the logic of $state \to Prop$ with a type of heaps by 2). Thus we achieve a separation logic $heap \to state \to Prop$, with which we can create assertions about programs involving both a state *and* a heap. Furthermore, since we create the separation logic from an equivalence relation on heaps, we can create assertions that are closed under the equivalence of heaps. This means that any assertion that holds for a given heap h will also hold for any equivalent heap h'. Analogously to the intermediate logic, the connectives now work on assertions of $heap \to state \to Prop$.

Using this logic, we can now define Hoare rules in terms of assertions that make claims about heaps. In other words, we are now able to define Hoare rules for the heap-manipulating operations of the extended Imp language.

3.4 Hoare Rules — Extended Version of Imp

This section describes Hoare rules for each of the four heap-manipulating operations. Note that these rules are local [13], and thus rely on the frame rule for reasoning about non-local specifications.

Read

$$\frac{}{\{e \mapsto e'\}\ X \rightsquigarrow [\ e\ ]\ \{\exists v.\ (e\ \mapsto\ e')\{^{v}/_{X}\}\ \wedge\ (X = (e'\{^{v}/_{X}\}))\}}\ \textsc{Read}$$

The Read rule reads the value at address e, e', into the state. Because the value of X has now changed, all previous occurrences of X must be substituted for the old value of X, here denoted by the existential variable v, to avoid corrupting previous definitions.

Write

$$\frac{}{\{e \mapsto -\}\ [\ e\ ] \rightsquigarrow e'\ \{e \mapsto e'\}}\ \textsc{Write}$$

The Write rule destructively updates an address on the heap. Importantly, the precondition requires the updated address to be active, i.e. it must exist on the heap beforehand. If one wishes to write to a non-active address, the address must be allocated first.

Allocate

$$\frac{}{\{\ emp\ \}\ X \mathrel{\&=} ALLOC\ n\ \{\exists a.X = a\ \wedge\ \odot_{i=a}^{a+n-1} i \mapsto 0\}}\ \textsc{Allocate}$$

The Allocate rule allocates n memory cells as specified by the parameter to $ALLOC$, and lets the variable X point to the first of these cells. As it is the case in other programming languages, such as Java [10], we allocate memory cells with a default value of zero. We do not have a separate notion of a *null* pointer, so pointing to zero is equivalent to pointing to *null*. Interpreting a value of zero as either a concrete '0' or a *null* pointer is up to the program.

The existential variable a in the postcondition denotes the address at which allocation begins. Because of our assumption that the address space of the heap is infinite, we can always choose this a to be greater than any other active address on the heap. Hence, allocation never fails.

Deallocate

$$\frac{}{\{e \mapsto -\}\ DEALLOC\ e\ \{emp\}}\ \textsc{Deallocate}$$

The Deallocate rule removes an active address from the heap. As reflected in the semantics presented in Section 2.3, deallocation is asymmetric with respect to allocation.

3.5 The Frame Rule

To make claims about more complex programs with effects outside just a local scope, we need to widen our perspective. We need to prove that the behaviour of a given command is not changed by the fact that there might be an additional part of the heap which it does not modify.

The side condition of the frame rule states that it only holds in the event that c does not modify R. This means that in order to use this rule, it would have to be proven that the command does not modify R. To avoid this, we alter the frame rule slightly: If c does not modify R, then R is unchanged by the execution of c. In the case that c does modify R, there must exist a list of values that when substituted for the variables that have been modified by c, will restore the original program state of R before c was executed. We can use this to construct a postcondition for the frame rule that will preserve the side condition from the standard frame rule. We formalize the modified frame rule in Figure 8.

$$\frac{\{P\}\ c\ \{Q\}}{\{P\ *\ R\}\ c\ \{Q\ *\ (\exists vs.\ R\{^{vs}/_{modified\_by\ c}\})\}}$$

Fig. 8. The modified definition of the frame rule.

Safety Monotonicity In Section 3.1 we introduced the notion of safety to our Hoare triple, as a way of ensuring that no commands that evaluate to an erroneous program state can satisfy the triple. Thus far we have only reasoned about safety in a local scope, so to use the frame rule to globalize our local Hoare rules, we have to consider safety monotonicity. Safety monotonicity means that if executing a command c is safe in a program state involving a heap h' , and $h' \sqsubseteq h$, then c must also be safe in a program state involving h. To be able to prove the frame rule, we have to also prove safety monotonicity.

3.6 Rules of Consequence

The precondition or postcondition of a Hoare rule might at times differ from the ones needed when verifying a program. While the condition we are trying to prove might be logically weaker or stronger than a given assertion, their semantical or structural differences might prevent them from being unified. To accommodate this difference, Hoare logic introduces the rules of consequence [7]. They state that any precondition P can be substituted for a weaker precondition P', while any postcondition Q can be substituted for a stronger postcondition Q'. These rules are shown in Figure 9.

$$\frac{P \vdash P' \quad \{P'\}\ c\ \{Q\}}{\{P\}\ c\ \{Q\}} \qquad \frac{Q' \vdash Q \quad \{P\}\ c\ \{Q'\}}{\{P\}\ c\ \{Q\}}$$

Fig. 9. The rules of consequence.

4 An Example

Figure 10 shows a simple example program swapping two values on the heap. An informal correctness proof of this program using the rules we have developed in the above sections is given in Figure 11. The frame rule is applied prior to each application of a Hoare rule, and lines 13–14 and 20–12 gives an example of the use of the rule of consequence. For simplicity, some trivial intermediate derivations are not shown. The derivations on lines 12 and 19 utilizes the notion of pure assertions [14], which in this case means that any conjunction between two assertions can be substituted for a separating conjunction if one (or both) of the assertions do not make claims about the heap.

```
1  { a ↦ c  *  b ↦ d }
2      (AId X) ⇜ [ a ];
3      (AId Y) ⇜ [ b ];
4      [ a ] ⇜ (AId Y);
5      [ b ] ⇜ (AId X)
6  { a ↦ d  *  b ↦ c }
```

Fig. 10. Example program: swap

```
1  { a ↦ c  *  b ↦ d }
2     { a ↦ c }                          (* Frame rule applied *)
3         (AId X) ⇜ [ a ]
4     { ∃ v, (a ↦ c) {v/X} ∧ X = (c {v/X}) }
5  { (∃ v, (a ↦ c) {v/X} ∧ X = (c {v/X}))
      * (∃ vs, (b ↦ d){vs/modified_by((AId X)⇜[a])}) }
6  { (b ↦ d) * (∃ v, (a ↦ c) {v/X} ∧ X = (c {v/X})) }
7     { b ↦ d }                          (* Frame rule applied *)
8         (AId Y) ⇜ [ b ]
9     { ∃ v, (b ↦ d) {v/Y} ∧ Y = (d {v/Y}) }
10 { (∃ v, (b ↦ d) {v/Y} ∧ Y = (d {v/Y}))
      * (∃ vs, (∃ v, (a ↦ c) {v/X} ∧ X = (c {v/X})){vs/modified_by((AId Y)⇜[b])})) }
11 { ((∃ v, (a ↦ c) {v/X}) ∧ (∃ v', X = (c {v'/X})))
      * (∃ v, (b ↦ d) {v/Y} ∧ Y = (d {v/Y})) }
12 { ((∃ v, (a ↦ c) {v/X}) * (∃ v', X = (c {v'/X})))
      * (∃ v, (b ↦ d) {v/Y} ∧ Y = (d {v/Y})) }
13    { ∃ v, (a ↦ c) {v/X} }             (* Frame rule applied *)
14     { a ↦ c }
15        [ a ] ⇜ (AId Y)
16     { a ↦ (AId Y) }
17 { a ↦ (AId Y) * (∃ vs, (∃ v', X = (c {v'/X})){vs/modified_by([a]⇜(AId Y))})
      * (∃ vs, (∃ v, (b ↦ d){v/Y} ∧ Y = (d {v/Y}){vs/modified_by([a]⇜(AId Y))}) }
18 { ((∃ v, (b ↦ d){v/Y}) ∧ (∃ v', Y = (d{v'/Y}))) * (∃ v, X = (c{v/X}))
      * (a ↦ (AId Y)) }
19 { (∃ v, (b ↦ d){v/Y}) * (∃ v', Y = (d{v'/Y})) * (∃ v, X = (c{v/X}))
      * (a ↦ (AId Y)) }
20    { (∃ v, (b ↦ d){v/Y}) }            (* Frame rule applied *)
21    { (b ↦ −) }
22        [ b ] ⇜ (AId X)
23    { (b ↦ (AId X)) }
24 { (b ↦ (AId X)) * (∃ vs, (∃ v', Y = (d{v'/Y})){vs/modified_by([b]⇜(AId X))})
      * (∃ vs, (∃ v, X = (c{v/X})){vs/modified_by([b]⇜(AId X))})
      * (∃ vs, a ↦ (AId Y){vs/modified_by([b]⇜(AId X))}) }
25 { (b ↦ (AId X)) * (Y = d) * (X = c) * (a ↦ (AId Y)) }
26 { a ↦ d  *  b ↦ c }
```

Fig. 11. Example: Swapping two values on the heap.

5 Related Work

The problem of verifying imperative languages with mutable states is a common one. In this section we present some related projects as well as alternative approaches and resources.

Appel and Blazy define formal semantics for a mid-level imperative programming language Cminor [1]. Similar to us, they have redesigned an existing imperative language, but in contrast to Imp, Cminor already has the notion of a heap. Instead they have had to redesign Cminor to be suitable for Hoare logic.

Jacobs et al. use Hoare triples extended with separation logic to create a verification environment for C and Java [8] called VeriFast. While VeriFast offer many features that we do not, the overall approach of the two projects have many aspects in common.

Ynot is a library for Coq which turns Coq into an environment for writing and verifying imperative programs. Ynot also defines a separation logic [16]. Instead of the approach that we have presented, we could have used Ynot as a basis for designing our own separation logic.

6 Conclusion

We have extended the Imp programming language with an addressable and mutable heap. For manipulating this heap, we have defined syntax, semantics, Hoare rules, and auxiliary theorems for verifying programs written in Imp. To be able to reason about programs using the heap, we have extended the formal system for verifying Imp programs with a separation logic.

By adding a heap, we have allowed for working with abstract data structures in Imp, which provides a considerable increase in the expressiveness of the language.

Although Imp is mainly a proof-of-concept language, the approach and formalizations presented should be generally applicable to other imperative languages.

References

[1] Andrew W. Appel and Sandrine Blazy. Separation logic for small-step c minor. In *IN 20TH INTERNATIONAL CONFERENCE ON THEOREM PROVING IN HIGHER-ORDER LOGICS (TPHOLS)*, 2007.

[2] J. Bengtson, J.B. Jensen, F. Sieczkowski, and L. Birkedal. Verifying object-oriented programs with higher-order separation logic in Coq. In *Proceedings of ITP*, February 2011.

[3] Jesper Bengtson. Charge! a framework for higher-order separation logic in coq. Lecture slides, March 2013. Available from http://www.itu.dk/courses/SASP/F2013/AMP%2016.pdf.

[4] Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. Local action and abstract separation logic. In *Proceedings of LICS*, pages 366–378, 2007.

[5] coq.inria.fr. Library coq.init.datatypes. http://coq.inria.fr/stdlib/Coq.Init.Datatypes.html.

[6] coq.inria.fr. A short introduction to coq. http://coq.inria.fr/a-short-introduction-to-coq.

[7] C. A. R. Hoare. An axiomatic basis for computer programming. *COMMUNICATIONS OF THE ACM*, 12(10):576–580, 1969.

[8] Bart Jacobs, Jan Smans, and Frank Piessens. A quick tour of the verifast program verifier. In *Proceedings of the 8th Asian conference on Programming languages and systems*, APLAS'10, pages 304–311, Berlin, Heidelberg, 2010. Springer-Verlag.

[9] Stéphane Lescuyer. Containers: a typeclass-based library of finite sets/maps. http://coq.inria.fr/pylons/contribs/view/Containers/v8.4.

[10] Oracle. Java language basics — primitive data types.

[11] Benjamin C. Pierce, Chris Casinghino, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjoberg, and Brent Yorgey. Sflib. http://www.cis.upenn.edu/~bcpierce/sf/SfLib.html.

[12] Benjamin C. Pierce, Chris Casinghino, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjoberg, and Brent Yorgey. *Software Foundations*. Electronic textbook, 2012. http://www.cis.upenn.edu/~bcpierce/sf.

[13] John Reynolds. Separation logic: A logic for shared mutable data structures. pages 55–74. IEEE Computer Society, 2002.

[14] John C. Reynolds. An introduction to separation logic, 2008.

[15] Hongseok Yang and Peter O'Hearn. A semantic basis for local reasoning, 2002.

[16] Ynot. Chlipala et al. http://ynot.cs.harvard.edu.