Towards Mechanized Program Verification with Separation Logic*

Tjark Weber

Institut für Informatik, Technische Universität München
Boltzmannstr. 3, D-85748 Garching b. München, Germany
webertj@in.tum.de

Abstract. Using separation logic, this paper presents three Hoare logics (corresponding to different notions of correctness) for the simple While language extended with commands for heap access and modification. Properties of separating conjunction and separating implication are mechanically verified and used to prove soundness and relative completeness of all three Hoare logics. The whole development, including a formal proof of the Frame Rule, is carried out in the theorem prover Isabelle/HOL.

Keywords. Separation Logic, Formal Program Verification, Interactive Theorem Proving

1 Introduction

Since C. A. R. Hoare's seminal work in 1969 [9], extensions of his logic have been developed for a multitude of language constructs [1, 2], including recursive procedures, nondeterminism, and even object-oriented languages. Extending Hoare logic to pointer programs however is not without difficulties. Recently separation logic was proposed by O'Hearn, Reynolds et al. [15, 19, 16] to overcome the local reasoning problem that is raised by the treatment of record components as arrays [6, 5].

Machine support is indispensable for formal program verification. Manual proofs are error-prone, and the verification of medium-sized programs has become feasible only because systems like SVC [3] can automatically discharge many proof obligations. Separation logic, although its usability has been demonstrated in several case studies [18, 4], currently lacks such support. In this paper we show how separation logic can be embedded into the theorem prover Isabelle/HOL [14]. We thereby lay the foundations for the use of separation logic in a semi-automatic verification tool. Our work is based on a previous formalization of a simple imperative language [12] which however did not consider pointers or separation logic. The current focus is on fundamental semantic properties of the resulting Hoare logics.

This paper is organized as follows. In Section 2 we define the programming language, together with its operational and denotational semantics. Section 3

* Research supported by *Graduiertenkolleg Logik in der Informatik* (PhD Program Logic in Computer Science) of the Deutsche Forschungsgemeinschaft (DFG).

introduces separation logic. In Section 4 we present three Hoare logics for our language, all of which are proved to be sound and relative complete. Also the Frame Rule is adressed, and its soundness is proved for one of the Hoare logics. We discuss the mechanical verification of a simple pointer algorithm, in-place list reversal, in Section 5.

2 The Language

2.1 Semantic Domains

We use an unspecified type *var* of variables. Addresses are elements of a numerical type, namely naturals (*nat*), to permit address arithmetic. For simplicity, the same type is used for values. Thereby the value of a variable can immediately be used as an address, with no need for a conversion function (cf. [16]).

Stores map variables to values. Heaps are modelled as partial functions from addresses to values. Other possibilities would be to define heaps as subsets of $addr \times val$ (with functionality constraints), or as $(addr \times val)\ list$ (again with functionality constraints, and modulo order). However, our current definition is much easier to state and work with in Isabelle/HOL since it can make use of readily available function types and does not require subtyping. On the other hand it also permits infinite heaps. This seemingly minor difference will become important again in Section 4.4, when we consider the Frame Rule.

A program state is either a pair consisting of a store and a heap, or *None*. The latter value will be used in the semantics of the language to indicate that a memory error occurred during program execution. Arithmetic and boolean expressions are only modelled semantically: they are just functions on stores (and hence independent of the heap).

Most of Isabelle's syntax used in this paper is close to standard mathematical notation and should not require further explanation. Both $\Longrightarrow$ and $\longrightarrow$ mean implication. $[\![P_1 ; \ldots ; P_n]\!] \Longrightarrow Q$ is an abbreviation for $P_1 \Longrightarrow \ldots \Longrightarrow P_n \Longrightarrow Q$. We use $'a \Rightarrow 'b$ for the type of total functions from $'a$ to $'b$. Likewise, infix $\rightharpoonup$ is used to denote the type of partial functions. Other type constructors, e.g. *list*, are written postfix. Thus the abovementioned semantic domains can be formalized as follows:

types $addr = nat$
$val = nat$
$store = var \Rightarrow val$
$heap = addr \rightharpoonup val$
$state = (store \times heap)\ option$
$aexp = store \Rightarrow val$
$bexp = store \Rightarrow bool$

2.2 Syntax

We consider an extension of the simple While language [9, 12] with new commands for memory allocation (list, alloc), heap lookup, heap mutation, and memory deallocation (dispose).

Both list and alloc allocate memory on the heap. list can only be used when the number of addresses to be allocated is known beforehand, i.e. for allocation of fixed-size records. The list command takes a list of arithmetic expressions as its second argument. The number of consecutive addresses to be allocated is given by the length of the list; the allocated memory is then initialized with the values of the expressions in the list. alloc on the other hand is meant for dynamic allocation of arrays. Its second argument is a single arithmetic expression that specifies the number of consecutive addresses to be allocated. The allocated memory is initialized with arbitrary values.

The lookup command assigns the value of an (allocated) address to a variable, the heap mutation command modifies the value of the heap at a given address, and dispose finally deallocates a single address. The precise operational semantics is given in Section 2.4.

2.3 Basic Operations on Heaps

Before we can define the semantics of our language, we need to introduce some basic operations on heaps. We define four functions to retrieve the value of a heap at a specific address, remove an address from the domain of a heap, test whether a set of addresses is free in a heap, and update a set of consecutive addresses in a heap with specific values. To some extent these functions allow us to abstract from our particular implementation of heaps as partial functions.

$$
\begin{array}{l}
\mathit{heap\text{-}lookup} :: \mathit{heap} \Rightarrow \mathit{addr} \Rightarrow \mathit{val} \\
\mathit{heap\text{-}lookup}\ h\ a \ \equiv\ \mathit{the}\ (h\ a) \\
\mathit{heap\text{-}remove} :: \mathit{heap} \Rightarrow \mathit{addr} \Rightarrow \mathit{heap} \\
\mathit{heap\text{-}remove}\ h\ a \ \equiv\ h(a{:=}\mathit{None}) \\
\mathit{heap\text{-}isfree} :: \mathit{heap} \Rightarrow \mathit{addr} \Rightarrow \mathit{nat} \Rightarrow \mathit{bool} \\
\mathit{heap\text{-}isfree}\ h\ a\ n \ \equiv\ \mathit{set}\ [a..a{+}n(] \cap \mathit{dom}\ h = \{\} \\
\mathit{heap\text{-}update} :: \mathit{heap} \Rightarrow \mathit{addr} \Rightarrow (\mathit{val}\ \mathit{list}) \Rightarrow \mathit{heap} \\
\mathit{heap\text{-}update}\ h\ a\ \mathit{vs} \ \equiv\ h([a..a{+}\mathit{length}\ \mathit{vs}(][\mapsto])\mathit{vs})
\end{array}
$$

Later we will also need notions of disjointness and union for heaps in order to define separating conjunction and separating implication. We say two heaps (or more generally, two partial functions) are *disjoint*, $\bowtie$, iff their domains are disjoint.

$$f \bowtie g \equiv \mathit{dom}\ f \cap \mathit{dom}\ g = \{\}$$

The *union* of heaps, $++$, is defined as one would expect, with the second heap having precedence over the first.

$$f{+}{+}g \equiv \lambda x.\ \mathit{case}\ g\ x\ \mathit{of}\ \mathit{None} \Rightarrow f\ x \mid \mathit{Some}\ y \Rightarrow \mathit{Some}\ y$$

We will only take the union of disjoint heaps however, and for those, $++$ is commutative:

Lemma $f \bowtie g \Longrightarrow f\ ++\ g = g\ ++\ f$

2.4 Operational Semantics

The operational semantics of our language is defined via a (big-step) evaluation relation $\longrightarrow_c$. We write $\langle c,s\rangle \longrightarrow_c t$ for *execution of c, started in state s, may terminate in state t*. This evaluation relation is defined inductively.

$\langle c,None\rangle \longrightarrow_c None$
$\langle \mathsf{skip},Some\ (s,h)\rangle \longrightarrow_c Some\ (s,h)$
$\langle x :== a,Some\ (s,h)\rangle \longrightarrow_c Some\ (s[x\mapsto a\ s],h)$
$\langle c0,s\rangle \longrightarrow_c s'' \Longrightarrow \langle c1,s''\rangle \longrightarrow_c s' \Longrightarrow \langle c0;\ c1,\ s\rangle \longrightarrow_c s'$
$b\ s \Longrightarrow \langle c0,Some\ (s,h)\rangle \longrightarrow_c s' \Longrightarrow \langle \mathsf{if}\ b\ \mathsf{then}\ c0\ \mathsf{else}\ c1,\ Some\ (s,h)\rangle \longrightarrow_c s'$
$\neg b\ s \Longrightarrow \langle c1,Some\ (s,h)\rangle \longrightarrow_c s' \Longrightarrow \langle \mathsf{if}\ b\ \mathsf{then}\ c0\ \mathsf{else}\ c1,\ Some\ (s,h)\rangle \longrightarrow_c s'$
$b\ s \Longrightarrow \langle c,Some\ (s,h)\rangle \longrightarrow_c s'' \Longrightarrow \langle \mathsf{while}\ b\ \mathsf{do}\ c,\ s''\rangle \longrightarrow_c s'$
$\quad \Longrightarrow \langle \mathsf{while}\ b\ \mathsf{do}\ c,\ Some\ (s,h)\rangle \longrightarrow_c s'$
$\neg b\ s \Longrightarrow \langle \mathsf{while}\ b\ \mathsf{do}\ c,Some\ (s,h)\rangle \longrightarrow_c Some\ (s,h)$
$[\![\ heap\text{-}isfree\ h\ a\ (length\ as);\ vs = map\ (\lambda e.\ e\ s)\ as\ ]\!]$
$\quad \Longrightarrow \langle x :== \mathsf{list}\ as,\ Some\ (s,h)\rangle \longrightarrow_c Some\ (s[x\mapsto a],\ heap\text{-}update\ h\ a\ vs)$
$(\forall\ a.\ \neg\ heap\text{-}isfree\ h\ a\ (length\ as)) \Longrightarrow \langle x :== \mathsf{list}\ as,\ Some\ (s,h)\rangle \longrightarrow_c None$
$(heap\text{-}isfree\ h\ a\ (n\ s) \wedge (length\ vs = n\ s))$
$\quad \Longrightarrow \langle x :== \mathsf{alloc}\ n,\ Some\ (s,h)\rangle \longrightarrow_c Some\ (s[x\mapsto a],\ heap\text{-}update\ h\ a\ vs)$
$(\forall\ a.\ \neg\ heap\text{-}isfree\ h\ a\ (n\ s)) \Longrightarrow \langle x :== \mathsf{alloc}\ n,\ Some\ (s,h)\rangle \longrightarrow_c None$
$a\ s \in dom\ h \Longrightarrow \langle x :== @a,Some\ (s,h)\rangle \longrightarrow_c Some\ (s[x\mapsto heap\text{-}lookup\ h\ (a\ s)],h)$
$a\ s \notin dom\ h \Longrightarrow \langle x :== @a,Some\ (s,h)\rangle \longrightarrow_c None$
$a\ s \in dom\ h \Longrightarrow \langle @a :== v,Some\ (s,h)\rangle \longrightarrow_c Some\ (s,heap\text{-}update\ h\ (a\ s)\ [v\ s])$
$a\ s \notin dom\ h \Longrightarrow \langle @a :== v,Some\ (s,h)\rangle \longrightarrow_c None$
$a\ s \in dom\ h \Longrightarrow \langle \mathsf{dispose}\ a,Some\ (s,h)\rangle \longrightarrow_c Some\ (s,heap\text{-}remove\ h\ (a\ s))$
$a\ s \notin dom\ h \Longrightarrow \langle \mathsf{dispose}\ a,Some\ (s,h)\rangle \longrightarrow_c None$

The rules for skip, assignment, composition, if, and while are standard, and only shown for completeness. The rules for the pointer commands come in pairs, with one rule leading to a valid successor state, the other one to the error state *None*. Which rule can be applied depends on the current heap. Allocating memory in a heap that does not have enough free addresses will result in an error, as will the attempt to access, modify, or deallocate free addresses.

With the exception of the first rule, these rules are all syntax directed (i.e. applicable only to a specific command). The first rule is needed to ensure that programs "don't get stuck" when an error occurred. For the same reason it is important that we do not restrict the rule for sequential composition to valid states.

Nondeterminism is introduced by the rules for list and alloc. Both commands choose an arbitrary sequence of (consecutive) free addresses for the newly allocated memory. Furthermore, alloc initializes this memory with arbitrary values.

2.5 Denotational Semantics

In addition to the operational semantics, we also define the denotational semantics of commands. We will show that both semantics are equivalent, thus we could (in principle) do without a denotational semantics. However, we found that

the denotational semantics, and in particular its fixed point characterization of while, is often easier to work with than the operational semantics. It enables us to prove semantic properties by induction on commands, rather than by induction on the evaluation relation. The denotational semantics of a command is given by a set of pairs of states.

types *com-den* = (*state* × *state*) *set*

The following function Γ is used to define the semantics of the while command as a least fixed point. The O operator denotes relational composition.

$$
\begin{array}{l}
\Gamma :: bexp \Rightarrow com\text{-}den \Rightarrow (com\text{-}den \Rightarrow com\text{-}den) \\
\Gamma\ b\ cd \equiv (\lambda\varphi. \\
\qquad \{\ (Some(s,h),t) \mid s\ h\ t.\ (Some(s,h),t) \in (\varphi\ O\ cd) \wedge b\ s\ \} \cup \\
\qquad \{\ (Some(s,h),Some(s,h)) \mid s\ h.\ \neg b\ s\ \} \cup \\
\qquad \{\ (None,None)\ \})
\end{array}
$$

The meaning function C, which maps each command to its denotational semantics, is now defined by primitive recursion.

$$
\begin{array}{l}
C\ \mathsf{skip} = Id \\
C\ (x :== a) = \{\ (Some(s,h),Some(s[x\mapsto a\ s],h)) \mid s\ h.\ True\ \} \cup \\
\qquad \{\ (None,None)\ \} \\
C\ (c0;c1) = C(c1)\ O\ C(c0) \\
C\ (\mathsf{if}\ b\ \mathsf{then}\ c1\ \mathsf{else}\ c2) = \{\ (Some(s,h),t) \mid s\ h\ t.\ (Some(s,h),t) \in C\ c1 \wedge b\ s\ \} \cup \\
\qquad \{\ (Some(s,h),t) \mid s\ h\ t.\ (Some(s,h),t) \in C\ c2 \wedge \neg b\ s\ \} \cup \\
\qquad \{\ (None,None)\ \} \\
C\ (\mathsf{while}\ b\ \mathsf{do}\ c) = lfp\ (\Gamma\ b\ (C\ c)) \\
C\ (x :== \mathsf{list}\ as) = \{\ (Some(s,h),Some(s[x\mapsto a],heap\text{-}update\ h\ a\ (map\ (\lambda e.\ e\ s)\ as))) \\
\qquad\qquad \mid s\ h\ a.\ heap\text{-}isfree\ h\ a\ (length\ as)\ \} \cup \\
\qquad \{\ (Some(s,h),None) \mid s\ h.\ \forall\ a.\ \neg\ heap\text{-}isfree\ h\ a\ (length\ as)\ \} \cup \\
\qquad \{\ (None,None)\ \} \\
C\ (x :== \mathsf{alloc}\ n) = \{\ (Some(s,h),Some(s[x\mapsto a],heap\text{-}update\ h\ a\ vs)) \\
\qquad\qquad \mid s\ h\ a\ vs.\ heap\text{-}isfree\ h\ a\ (n\ s) \wedge (length\ vs = n\ s)\ \} \cup \\
\qquad \{\ (Some(s,h),None) \mid s\ h.\ \forall\ a.\ \neg\ heap\text{-}isfree\ h\ a\ (n\ s)\ \} \cup \\
\qquad \{\ (None,None)\ \} \\
C\ (x :== @a) = \{\ (Some(s,h),Some(s[x\mapsto heap\text{-}lookup\ h\ (a\ s)],h)) \\
\qquad\qquad \mid s\ h.\ a\ s \in dom\ h\ \} \cup \\
\qquad \{\ (Some(s,h),None) \mid s\ h.\ a\ s \notin dom\ h\ \} \cup \\
\qquad \{\ (None,None)\ \} \\
C\ (@a :== v) = \{\ (Some(s,h),Some(s,heap\text{-}update\ h\ (a\ s)\ [v\ s])) \\
\qquad\qquad \mid s\ h.\ a\ s \in dom\ h\ \} \cup \\
\qquad \{\ (Some(s,h),None) \mid s\ h.\ a\ s \notin dom\ h\ \} \cup \\
\qquad \{\ (None,None)\ \} \\
C\ (\mathsf{dispose}\ a) = \{\ (Some(s,h),Some(s,heap\text{-}remove\ h\ (a\ s))) \mid s\ h.\ a\ s \in dom\ h\ \} \cup \\
\qquad \{\ (Some(s,h),None) \mid s\ h.\ a\ s \notin dom\ h\ \} \cup \\
\qquad \{\ (None,None)\ \}
\end{array}
$$

By induction on $\longrightarrow_c$, one can show that $\langle c,s\rangle \longrightarrow_c t$ implies $(s,\ t) \in C\ c$. The other direction, i.e. $(s,\ t) \in C\ c \Longrightarrow \langle c,s\rangle \longrightarrow_c t$, is shown by induction on c. For both directions, only the while case is not automatic (but still fairly

simple). Taking these two results together, we obtain equivalence of denotational and operational semantics:

Theorem $(s,t) \in C(c) \;=\; (\langle c,s\rangle \longrightarrow_c t)$

We will freely use this result in the following proofs whenever it is more convenient to reason using a particular semantics.

3 Assertions of Separation Logic

We only model the semantics of assertions, not their syntax. Assertions are predicates on stores and heaps:

types *assn* = *store* ⇒ *heap* ⇒ *bool*

This semantic approach (or *shallow embedding*) entails that any HOL term of the correct type can be used as an assertion, not just formulae of separation logic. If we had modelled assertions syntactically, we would have had to redefine most of HOL's logical connectives (including classical conjunction, implication, and first-order quantification), and the explicit definition of a formula's semantics would have introduced another layer of abstraction between separation logic and the lemmata and proof automation available in HOL. Our current definition on the other hand allows us to consider separation logic as an extension of higher-order logic, thereby giving us the features of HOL (almost) for free. The main drawback for our purposes is perhaps an esthetic one: when mixing classical and separating connectives, we have to use λ-abstractions to make their types compatible (cf. Section 3.1). A more detailed discussion of the respective strengths and weaknesses of shallow vs. *deep* embeddings is forthcoming [17].

Let us now introduce some abbreviations. *emp* asserts that the heap is empty (i.e. that no address is allocated), and $a \mapsto v$ is true of a heap iff a is the only allocated address, and it points to the value v.

$emp\ h \equiv dom\ h = \{\}$
$(a \mapsto v)\ h \equiv dom\ h = \{a\} \wedge heap\text{-}lookup\ h\ a = v$

Separation logic has two special connectives, separating conjunction ($\wedge*$) and separating implication ($-\!*$). $P \wedge* Q$ states that the heap can be split into disjoint parts satisfying P and Q, respectively. $P -\!* Q$ is true of a heap h iff Q holds for every extension of h with a disjoint part that satisfies P. These connectives are defined using quantification over heaps. The definitional approach allows us to *prove* their properties, rather than to introduce them as new axioms.

$sep\text{-}conj :: (heap \Rightarrow bool) \Rightarrow (heap \Rightarrow bool) \Rightarrow heap \Rightarrow bool$ **(infixl** $\wedge*$**)**
$(P \wedge* Q)\ h \equiv \exists h'\ h''.\ (h' \bowtie h'') \wedge (h' +\!\!+ h'' = h) \wedge P\ h' \wedge Q\ h''$

$sep\text{-}imp :: (heap \Rightarrow bool) \Rightarrow (heap \Rightarrow bool) \Rightarrow heap \Rightarrow bool$ **(infixr** $-\!*$**)**
$(P -\!* Q)\ h \equiv \forall h'.\ ((h' \bowtie h) \wedge P\ h') \longrightarrow Q\ (h +\!\!+ h')$

Although assertions of separation logic may depend on the store, they usually do so only in a completely homomorphic fashion (cf. [16]). Therefore this dependency can easily be eliminated from compound formulae, and it is sufficient to define separating conjunction and implication for predicates of type *heap* ⇒ *bool*. Further assertions denote that a heap contains exactly one allocated address a (written $a{\mapsto}-$), and that an address a points to a value v, where other addresses in the heap may be allocated as well ($a{\hookrightarrow}v$). Using address arithmetic (*Suc* is the successor function on naturals), we extend these notions to lists of values.

$(a{\mapsto}-)\ h \equiv \exists\, v.\ (a{\mapsto}v)\ h$
$(a{\hookrightarrow}v) \equiv (a{\mapsto}v) \mathbin{\wedge\!*}\ \mathit{true}$
$(a[{\mapsto}][]) \quad = \mathit{emp}$
$(a[{\mapsto}](v\#vs)) = ((a{\mapsto}v) \mathbin{\wedge\!*}\ ((\mathit{Suc}\ a)[{\mapsto}]vs))$
$(a[{\hookrightarrow}][]) \quad = \mathit{true}$
$(a[{\hookrightarrow}](v\#vs)) = ((a{\hookrightarrow}v) \mathbin{\wedge\!*}\ ((\mathit{Suc}\ a)[{\hookrightarrow}]vs))$

3.1 Properties of Separating Conjunction and Separating Implication

We can relatively easily prove associativity and commutativity of $\wedge*$, identity of *emp* under $\wedge*$, and various distributive and semidistributive laws. Most of the proofs are automatic; sometimes however we need to manually instantiate the existential quantifiers obtained by unfolding the definition of $\wedge*$.

Lemma $P \mathbin{\wedge\!*} (Q \mathbin{\wedge\!*} R) = (P \mathbin{\wedge\!*} Q) \mathbin{\wedge\!*} R$
Lemma $P \mathbin{\wedge\!*} Q = Q \mathbin{\wedge\!*} P$
Lemma $\mathit{emp} \mathbin{\wedge\!*} P = P$
Lemma $P \mathbin{\wedge\!*} \mathit{emp} = P$
Lemma $((\lambda h.\ P\ h \vee Q\ h) \mathbin{\wedge\!*} R)\ h = (P \mathbin{\wedge\!*} R)\ h \vee (Q \mathbin{\wedge\!*} R)\ h$
Lemma $((\lambda h.\ P\ h \wedge Q\ h) \mathbin{\wedge\!*} R)\ h \longrightarrow ((P \mathbin{\wedge\!*} R)\ h \wedge (Q \mathbin{\wedge\!*} R)\ h)$
Lemma $((\lambda h.\ \exists x.\ P\ x\ h) \mathbin{\wedge\!*} Q)\ h = (\exists x.\ (P\ x \mathbin{\wedge\!*} Q)\ h)$
Lemma $((\lambda h.\ \forall x.\ P\ x\ h) \mathbin{\wedge\!*} Q)\ h \longrightarrow (\forall x.\ (P\ x \mathbin{\wedge\!*} Q)\ h)$
Lemma $[\![\ \forall h.\ P\ h \longrightarrow P'\ h; \forall h.\ Q\ h \longrightarrow Q'\ h\ ]\!] \Longrightarrow (P \mathbin{\wedge\!*} Q)\ h \longrightarrow (P' \mathbin{\wedge\!*} Q')\ h$
Lemma $[\![\ \forall h.\ (P \mathbin{\wedge\!*} Q)\ h \longrightarrow R\ h\ ]\!] \Longrightarrow P\ h \longrightarrow (Q \mathbin{-\!\!*} R)\ h$
Lemma $[\![\ \forall h.\ P\ h \longrightarrow (Q \mathbin{-\!\!*} R)\ h\ ]\!] \Longrightarrow (P \mathbin{\wedge\!*} Q)\ h \longrightarrow R\ h$

Following Reynolds [16], we have also defined *pure*, *intuitionistic*, *strictly exact*, and *domain exact* assertions, and proved many of their properties. Our growing library of lemmata serves as a basis for verification proofs and increased proof automation.

4 Hoare Logics

4.1 Partial Correctness

In this subsection we present a Hoare logic for partial correctness. We say a Hoare triple $\{P\}c\{Q\}$ is *valid*, $\models_p$, iff every terminating execution of c that starts in

a valid state (i.e. in a state of the form *Some* $(s,\ h)$) satisfying the precondition P ends up in a state that satisfies Q, *unless* a memory error occurs.

$$\models_p \{P\}c\{Q\} \equiv \\ \quad \forall\, s\ h\ s'\ h'.\ (\mathit{Some}\ (s,h),\ \mathit{Some}\ (s',h')) \in C(c) \longrightarrow P\ s\ h \longrightarrow Q\ s'\ h'$$

Hence there are two ways in which a Hoare triple can be trivially valid: c, when executed in a state that satisfies the precondition, *i)* does not terminate at all, or *ii)* only terminates in the error state *None*.

Derivability, $\vdash_p$, of Hoare triples is defined inductively. The following set of Hoare rules is both sound and relative complete with respect to the notion of validity defined above.

$$\vdash_p \{P\}\ \mathsf{skip}\ \{P\}$$
$$\vdash_p \{\lambda s\ h.\ P\ (s[x\mapsto(a\ s)])\ h\}\ x{:==}a\ \{P\}$$
$$[\![\ \vdash_p \{P\}c\{Q\};\ \vdash_p \{Q\}d\{R\}\ ]\!] \Longrightarrow \vdash_p \{P\}\ c;d\ \{R\}$$
$$[\![\ \vdash_p \{\lambda s\ h.\ P\ s\ h \wedge b\ s\}c\{Q\};\ \vdash_p \{\lambda s\ h.\ P\ s\ h \wedge \neg b\ s\}d\{Q\}\ ]\!] \Longrightarrow \\ \quad \vdash_p \{P\}\ \mathsf{if}\ b\ \mathsf{then}\ c\ \mathsf{else}\ d\ \{Q\}$$
$$\vdash_p \{\lambda s\ h.\ P\ s\ h \wedge b\ s\}\ c\ \{P\} \Longrightarrow \\ \quad \vdash_p \{P\}\ \mathsf{while}\ b\ \mathsf{do}\ c\ \{\lambda s\ h.\ P\ s\ h \wedge \neg b\ s\}$$
$$\vdash_p \{\lambda s\ h.\ (\forall\, a.\ ((a[\mapsto](\mathit{map}\ (\lambda e.\ e\ s)\ \mathit{as})) -\!\!* (P\ (s[x\mapsto a]))))\ h)\} \\ \quad x :== \mathsf{list}\ \mathit{as}\ \{P\}$$
$$\vdash_p \{\lambda s\ h.\ (\forall\, a\ \mathit{vs}.\ (\mathit{length}\ \mathit{vs} = n\ s) \longrightarrow ((a[\mapsto]\mathit{vs}) -\!\!* (P\ (s[x\mapsto a]))))\ h)\} \\ \quad x :== \mathsf{alloc}\ n\ \{P\}$$
$$\vdash_p \{\lambda s\ h.\ (a\ s \in \mathit{dom}\ h) \longrightarrow P\ (s[x\mapsto \mathit{heap\text{-}lookup}\ h\ (a\ s)])\ h\}\ x :== @a\ \{P\}$$
$$\vdash_p \{\lambda s\ h.\ (a\ s \in \mathit{dom}\ h) \longrightarrow P\ s\ (\mathit{heap\text{-}update}\ h\ (a\ s)\ [v\ s])\}\ @a :== v\ \{P\}$$
$$\vdash_p \{\lambda s\ h.\ (a\ s \in \mathit{dom}\ h) \longrightarrow P\ s\ (\mathit{heap\text{-}remove}\ h\ (a\ s))\}\ \mathsf{dispose}\ a\ \{P\}$$
$$[\![\ \forall\, s\ h.\ P'\ s\ h \longrightarrow P\ s\ h;\ \vdash_p \{P\}c\{Q\};\ \forall\, s\ h.\ Q\ s\ h \longrightarrow Q'\ s\ h\ ]\!] \Longrightarrow \\ \quad \vdash_p \{P'\}c\{Q'\}$$

Soundness is proved by a straightforward induction on $\vdash_p$. The only nontrivial case is the while rule; it requires fixed point induction.

Theorem $\vdash_p \{P\}c\{Q\} \Longrightarrow \models_p \{P\}c\{Q\}$

To prove completeness, we employ the notion of *weakest (liberal) preconditions* [8].

$$\mathit{wp} :: \mathit{com} \Rightarrow \mathit{assn} \Rightarrow \mathit{assn}$$
$$\mathit{wp}\ c\ Q \equiv \lambda s\ h.\ (\forall\, s'\ h'.\ (\mathit{Some}\ (s,h),\ \mathit{Some}(s',h')) \in C(c) \longrightarrow Q\ s'\ h')$$

The key to the completeness proof is a lemma stating that Hoare triples of the form $\{\mathit{wp}\ c\ Q\}\ c\ \{Q\}$ are derivable. The lemma is proved by induction on c.

Lemma $\forall\, Q.\ \vdash_p \{\mathit{wp}\ c\ Q\}\ c\ \{Q\}$

From this, relative completeness of the Hoare rules follows easily with the rule of consequence.

Theorem $\models_p \{P\}c\{Q\} \Longrightarrow \vdash_p \{P\}c\{Q\}$

4.2 Tight Specifications

The Hoare logic from Section 4.1 does not guarantee the absence of memory errors. We now consider a slightly different Hoare logic for partial correctness, which perhaps better reflects the principle that "well-specified programs don't go wrong" [16]. In this logic, a Hoare triple $\{P\}c\{Q\}$ is *valid*, $\models_t$, iff every terminating execution of c that starts in a valid state satisfying P ends up in a valid state satisfying Q.

$$\begin{array}{l}\models_t \{P\}c\{Q\} \equiv \\ \quad \forall\, s\ h.\ ((P\ s\ h \longrightarrow (\mathit{Some}\ (s,h),\ \mathit{None}) \notin C(c)) \\ \qquad \wedge\ (\forall\, s'\ h'.\ (\mathit{Some}\ (s,h),\ \mathit{Some}\ (s',h')) \in C(c) \longrightarrow P\ s\ h \longrightarrow Q\ s'\ h'))\end{array}$$

Compared to the previous Hoare logic, we have added a safety constraint expressing that the error state *None* must be unreachable. Specifications are now "*tight*" in the sense that every address accessed by c must either be mentioned in the precondition, or allocated by c before it is used (in which case the precondition must ensure the existence of a free address).

Of course the preconditions in our Hoare rules must be modified to reflect this change in the definition of validity. The rules for skip, assignment, composition, if, and while, as well as the consequence rule, remain unchanged; therefore they are not shown below. The rules for list, alloc, lookup, mutate, and dispose however now have preconditions which consist of two parts: one guaranteeing the absence of an error, and the other one guaranteeing that the postcondition will hold in all reachable states.

$$\begin{array}{l}\vdash_t \{\lambda s\ h.\ (\exists\, a.\ \mathit{heap\text{-}isfree}\ h\ a\ (\mathit{length}\ as)) \wedge (\forall\, a.\ ((a[\mapsto](\mathit{map}\ (\lambda e.\ e\ s)\ as)) \\ \qquad -\!\!*\ (P\ (s[x\mapsto a]))))\ h)\}\ x :== \mathsf{list}\ as\ \{P\} \\ \vdash_t \{\lambda s\ h.\ (\exists\, a.\ \mathit{heap\text{-}isfree}\ h\ a\ (n\ s)) \wedge (\forall\, a\ vs.\ (\mathit{length}\ vs = n\ s) \\ \qquad \longrightarrow ((a[\mapsto]vs) -\!\!*\ (P\ (s[x\mapsto a]))))\ h)\}\ x :== \mathsf{alloc}\ n\ \{P\} \\ \vdash_t \{\lambda s\ h.\ (\exists\, v.\ ((a\ s)\hookrightarrow v)\ h \wedge P\ (s[x\mapsto v])\ h)\}\ x :== @a\ \{P\} \\ \vdash_t \{\lambda s\ h.\ ((a\ s)\mapsto - \wedge\!\!*\ (((a\ s)\mapsto(v\ s)) -\!\!*\ P\ s))\ h\}\ @a :== v\ \{P\} \\ \vdash_t \{\lambda s\ h.\ ((a\ s)\mapsto - \wedge\!\!*\ P\ s)\ h\}\ \mathsf{dispose}\ a\ \{P\}\end{array}$$

These rules are similar to the ones presented in [16], with the exception that for list and alloc, we need to assert the existence of available memory in the precondition. (In [16], free heap cells are guaranteed to exist because heaps are always finite.)

Using similar techniques as before – in particular, induction on $\vdash_t$ and a suitably modified notion of weakest liberal preconditions – we can prove soundness and relative completeness of this Hoare logic. Both properties are slightly more difficult to prove than for the logic in Section 4.1, since we do not just have to deal with the postcondition, but also with the safety constraint.

Theorem $(\models_t \{P\}c\{Q\}) = (\vdash_t \{P\}c\{Q\})$

4.3 Total Correctness

So far we have only considered partial correctness, where a Hoare triple is valid iff every reachable state satisfies the postcondition. If we also want to take termination into account, we need to define a judgment $c \downarrow s$ that expresses guaranteed

termination of c started in state s. The Hoare rules then differ from those for partial correctness only in the one place where nontermination can arise: the while rule. For the simple While language, the details have been carried out in [13]. Since the new pointer commands always terminate, the development would be almost identical for our extended language.

4.4 The Frame Rule

In Hoare logic for the simple While language, one can show that if $\models\{P\}c\{Q\}$, then $\models\{P\wedge R\}c\{Q\wedge R\}$, provided that no variables modified by c occur free in R. Under certain conditions (cf. the discussion in [19]), separation logic allows us to obtain a similar rule for our extended language:

$$\models\{P\}c\{Q\} \Longrightarrow \models\{P\wedge{*}R\}c\{Q\wedge{*}R\} \quad ,$$

with the same syntactic side condition on R. This *Frame Rule* is essential for modular verification, in particular in the presence of procedures. Unfortunately however, the Frame Rule does not hold in the two previously defined Hoare logics. As counterexamples consider

$$\models_p \{emp\}\ \mathsf{dispose}\ (\lambda s.\ 0)\ \{false\}$$
$$\neg(\models_p \{emp\wedge{*}true\}\ \mathsf{dispose}\ (\lambda s.\ 0)\ \{false\wedge{*}true\})$$

for the Hoare logic in Section 4.1, and

$$\models_t \{emp\}\ x{:=}{=}\mathsf{alloc}\ (\lambda s.\ 1)\ \{true\}$$
$$\neg(\models_t \{emp\wedge{*}true\}\ x{:=}{=}\mathsf{alloc}\ (\lambda s.\ 1)\ \{true\wedge{*}true\}$$

for the logic in Section 4.2. The reason why the Frame Rule does not hold in the second Hoare logic is that this logic, when used with potentially infinite heaps, does not validate *safety monotonicity* [19]. Safety monotonicity means that if executing c in a state with heap $h1$ is safe (i.e. cannot lead to *None*), then executing c in a state with an extended heap $h1 +\!\!+ h2$ (for $h1 \bowtie h2$) must be safe as well. This is in particular false for list and alloc, since there may not be enough free addresses left in the extended heap.

We could restore safety monotonicity by only considering finite heaps, as done in existing work on separation logic [16, 19]. Combined with an infinite contiguous address space, memory allocation will then always succeed. We note however that a slightly weaker property is sufficient to establish safety monotonicity: namely that heaps contain arbitrary long sequences of unallocated addresses. (Reynolds imposes an equivalent, but more complicated condition on the set of addresses in [16].) This motivates a Hoare logic where we only consider such *lacunary* heaps.

$$lacunary\ h \equiv \forall n.\ \exists a.\ heap\text{-}isfree\ h\ a\ n$$

Clearly every finite heap is lacunary, and every heap whose domain is contained in the domain of a lacunary heap is itself lacunary. Furthermore, lacunarity is invariant under execution of commands. This can be shown by induction

on the evaluation relation $\longrightarrow_c$, with the rules for list, alloc, and dispose being the more interesting cases. Unlike finiteness however, lacunarity is not preserved under union of heaps.

Lemma *finite* $(dom\ h) \Longrightarrow lacunary\ h$
Lemma $[\![\ dom\ h \subseteq dom\ h';\ lacunary\ h'\ ]\!] \Longrightarrow lacunary\ h$
Lemma $\langle c,Some\ (s,h)\rangle \longrightarrow_c Some\ (s',h') \Longrightarrow lacunary\ h' = lacunary\ h$

Based on the concept of lacunary heaps, we define yet another notion of validity, $\models_l$, for Hoare triples. The requirements are exactly the same as for $\models_t$ (i.e. the postcondition must hold in every reachable valid state, and the error state *None* must be unreachable), but for $\models_l$, they need to hold only if the initial heap is lacunary.

$$\begin{array}{l}\models_l \{P\}c\{Q\} \equiv \\ \quad \forall\, s\ h.\ lacunary\ h \longrightarrow ((P\ s\ h \longrightarrow (Some\ (s,h),\ None) \notin C(c)) \\ \qquad \wedge\ (\forall\, s'\ h'.\ (Some\ (s,h),\ Some\ (s',h')) \in C(c) \longrightarrow P\ s\ h \longrightarrow Q\ s'\ h'))\end{array}$$

A set of sound and relative complete Hoare rules is obtained by modifying the preconditions in the rules for skip, assignment, list, alloc, lookup, mutate, and dispose accordingly. The rules for list and alloc can then be simplified a little, since lacunarity already implies the existence of free addresses. The rules for composition, if, and while are the same as for $\vdash_t$. To prove completeness of the while rule, however, we need to strengthen the consequence rule.

$$\begin{array}{l}\vdash_l \{\lambda s\ h.\ lacunary\ h \longrightarrow P\ s\ h\}\ \mathsf{skip}\ \{P\} \\ \vdash_l \{\lambda s\ h.\ lacunary\ h \longrightarrow P\ (s[x\mapsto(a\ s)])\ h\}\ x{:=}{=}a\ \{P\} \\ \vdash_l \{\lambda s\ h.\ lacunary\ h \longrightarrow (\forall\, a.\ ((a[\mapsto](map\ (\lambda e.\ e\ s)\ as)) \\ \qquad -{*}\ (\lambda hh.\ (P\ (s[x\mapsto a])\ hh)))\ h)\}\ x :== \mathsf{list}\ as\ \{P\} \\ \vdash_l \{\lambda s\ h.\ lacunary\ h \longrightarrow (\forall\, a\ vs.\ (length\ vs = n\ s) \\ \qquad \longrightarrow ((a[\mapsto]vs)\ -{*}\ (\lambda hh.\ (P\ (s[x\mapsto a])\ hh)))\ h)\}\ x :== \mathsf{alloc}\ n\ \{P\} \\ \vdash_l \{\lambda s\ h.\ lacunary\ h \longrightarrow (\exists\, v.\ ((a\ s)\hookrightarrow v)\ h \wedge P\ (s[x\mapsto v])\ h)\}\ x :== @a\ \{P\} \\ \vdash_l \{\lambda s\ h.\ lacunary\ h \longrightarrow ((a\ s)\mapsto -\ \wedge{*}\ (((a\ s)\mapsto(v\ s))\ -{*}\ P\ s))\ h\} \\ \qquad @a :== v\ \{P\} \\ \vdash_l \{\lambda s\ h.\ lacunary\ h \longrightarrow ((a\ s)\mapsto -\ \wedge{*}\ P\ s)\ h\}\ \mathsf{dispose}\ a\ \{P\} \\ [\![\ \forall\, s\ h.\ lacunary\ h \longrightarrow P'\ s\ h \longrightarrow P\ s\ h;\ \vdash_l \{P\}c\{Q\}; \\ \qquad \forall\, s\ h.\ lacunary\ h \longrightarrow Q\ s\ h \longrightarrow Q'\ s\ h\ ]\!] \Longrightarrow \vdash_l \{P'\}c\{Q'\}\end{array}$$

As usual, soundness is proved by induction on $\vdash_l$, and relative completeness is proved using (an adapted notion of) weakest liberal preconditions. The abovementioned properties of lacunary heaps are used in both directions of the proof.

Theorem $(\models_l \{P\}c\{Q\}) = (\vdash_l \{P\}c\{Q\})$

4.5 Proving the Frame Rule

The proof of the Frame Rule presented in this subsection is largely based on [19]. Since we did not specify the syntax of assertions, our first step must be a semantic version of the Frame Rule's side condition. The set of variables that are *modified* by a command is defined as follows.

$$
\begin{array}{ll}
\mathit{ModifiedVars}\ \mathsf{skip} & = \{\} \\
\mathit{ModifiedVars}\ (x{:==}a) & = \{x\} \\
\mathit{ModifiedVars}\ (c1;c2) & = \mathit{ModifiedVars}\ c1 \cup \mathit{ModifiedVars}\ c2 \\
\mathit{ModifiedVars}\ (\mathsf{if}\ b\ \mathsf{then}\ c1\ \mathsf{else}\ c2) & = \mathit{ModifiedVars}\ c1 \cup \mathit{ModifiedVars}\ c2 \\
\mathit{ModifiedVars}\ (\mathsf{while}\ b\ \mathsf{do}\ c) & = \mathit{ModifiedVars}\ c \\
\mathit{ModifiedVars}\ (x :== \mathsf{list}\ as) & = \{x\} \\
\mathit{ModifiedVars}\ (x :== \mathsf{alloc}\ n) & = \{x\} \\
\mathit{ModifiedVars}\ (x :== @a) & = \{x\} \\
\mathit{ModifiedVars}\ (@a :== v) & = \{\} \\
\mathit{ModifiedVars}\ (\mathsf{dispose}\ a) & = \{\}
\end{array}
$$

By induction on c, one can show that for $x \notin \mathit{ModifiedVars}\ c$, the value of x is invariant under execution of c.

Lemma $\forall s\ h\ s'\ h'.\ (\mathit{Some}\ (s,h),\ \mathit{Some}\ (s',h')) \in C(c) \longrightarrow x \notin \mathit{ModifiedVars}(c) \longrightarrow (s\ x = s'\ x)$

We say an assertion P is *independent* of a set of variables S, written $S \natural P$, iff P does not depend on the value of variables in S.

$$S \natural P \equiv \forall s\ s'.\ (\forall x.\ x \notin S \longrightarrow s\ x = s'\ x) \longrightarrow (P\ s = P\ s')$$

The key lemma is now proved by induction on c. It states that a memory error occuring in a lacunary heap can also occur in every subheap, and a valid execution either has a corresponding "restricted" execution in the subheap, or it corresponds to a memory error.

Lemma $\forall s\ h1\ h2\ s'\ h'.\ h1 \bowtie h2$
$\longrightarrow (\mathit{lacunary}\ (h1{+}{+}h2) \longrightarrow (\mathit{Some}\ (s,h1{+}{+}h2),\ \mathit{None}) \in C(c)$
$\longrightarrow (\mathit{Some}\ (s,h1),\ \mathit{None}) \in C(c))$
$\wedge\ ((\mathit{Some}\ (s,h1{+}{+}h2),\ \mathit{Some}(s',h')) \in C(c)$
$\longrightarrow (\mathit{Some}\ (s,h1),\ \mathit{None}) \in C(c)$
$\vee\ (\exists h1'.\ h1' \bowtie h2 \wedge h1'{+}{+}h2 = h' \wedge (\mathit{Some}\ (s,h1),\ \mathit{Some}(s',h1')) \in C(c)))$

Both safety monotonicity and the *frame property* [19] follow immediately.

Lemma $[\![\ h1 \bowtie h2\ ;\ \mathit{lacunary}\ (h1{+}{+}h2)\ ;\ (\mathit{Some}\ (s,h1),\ \mathit{None}) \notin C(c)\ ]\!]$
$\Longrightarrow (\mathit{Some}\ (s,h1{+}{+}h2),\ \mathit{None}) \notin C(c)$

Lemma $[\![\ h1 \bowtie h2\ ;\ (\mathit{Some}\ (s,h1),\ \mathit{None}) \notin C(c)\ ;$
$(\mathit{Some}\ (s,h1{+}{+}h2),\ \mathit{Some}(s',h')) \in C(c)\ ]\!]$
$\Longrightarrow \exists h1'.\ h1' \bowtie h2 \wedge h1'{+}{+}h2 = h' \wedge (\mathit{Some}\ (s,h1),\ \mathit{Some}(s',h1')) \in C(c)$

Finally we can prove the Frame Rule. Safety monotonicity is used to show that the error state is unreachable, and the frame property proves that every reachable state satisfies the postcondition.

Theorem $[\![\models_l \{P\}c\{Q\};\ (\mathit{ModifiedVars}\ c) \natural R\ ]\!]$
$\Longrightarrow \models_l \{\lambda s\ h.\ (P\ s \wedge{*}\ R\ s)\ h\}c\{\lambda s\ h.\ (Q\ s \wedge{*}\ R\ s)\ h\}$

5 Example: In-place List Reversal

To evaluate the practical applicability of our framework, we verify an in-place list reversal algorithm. This relatively simple algorithm has been considered

before [6, 5], also by Reynolds [16], who gave an (informal) correctness proof using separation logic, and by Mehta and Nipkow [11], who formally verified the algorithm in Isabelle/HOL, but without separation logic. The actual algorithm is shown below. i, j and k are variables: i contains a pointer to the current (initially, the first) list cell, j contains a pointer to the previous list cell (initially *null*), and k, which is initialized at the beginning of the loop body, contains a pointer to the next list cell. *null* is just an abbreviation for *0*, rather than a distinguished address. This resembles the treatment of the *NULL* pointer in (e.g.) C [10].

$$
\begin{array}{l}
\mathit{reverse}\ i\ j\ k \equiv \\
\quad (j :== (\lambda s.\ \mathit{null})); \qquad (*\ \mathit{initially, there is no previous list cell}\ *) \\
\quad \mathsf{while}\ (\lambda s.\ s\ i \neq \mathit{null})\ \mathsf{do} \qquad (*\ \mathit{end of list reached?}\ *) \\
\quad\quad ((((k :== @(\lambda s.\ \mathit{Suc}\ (s\ i))); \qquad (*\ \mathit{the next list cell}\ *) \\
\quad\quad (@(\lambda s.\ \mathit{Suc}\ (s\ i)) :== (\lambda s.\ s\ j))); \qquad (*\ \mathit{update pointer to next cell}\ *) \\
\quad\quad (j :== (\lambda s.\ s\ i))); \qquad (*\ \mathit{previous} :== \mathit{current}\ *) \\
\quad\quad (i :== (\lambda s.\ s\ k))) \qquad (*\ \mathit{current} :== \mathit{next}\ *)
\end{array}
$$

The corresponding specification theorem states that if i, j, and k are distinct and i points to a list *vs*, then after execution of *reverse i j k*, j will point to the reversed list.

$$
\begin{array}{l}
\textbf{Theorem}\ \models_t \{\ \lambda s\ h.\ \mathit{heap\text{-}list\ vs}\ (s\ i)\ h \wedge \mathit{distinct}\ [i,j,k]\ \} \\
\qquad \mathit{reverse}\ i\ j\ k \\
\qquad \{\ \lambda s\ h.\ \mathit{heap\text{-}list}\ (\mathit{rev\ vs})\ (s\ j)\ h\ \}
\end{array}
$$

The predicate *heap-list* relates singly linked linear lists on the heap to Isabelle/HOL lists. *heap-list vs a h* is true iff the heap h contains a singly linked linear list whose cells contain the values *vs*, and whose first cell is at address *a*.

$$
\begin{array}{l}
\mathit{heap\text{-}list}\ []\ \ a\ h = ((a = \mathit{null}) \wedge \mathit{emp}\ h) \\
\mathit{heap\text{-}list}\ (v\#vs)\ a\ h = ((a \neq \mathit{null}) \wedge (\exists k.\ ((a[\mapsto][v,k]) \wedge\!* \mathit{heap\text{-}list\ vs}\ k)\ h))
\end{array}
$$

To prove the specification, we use soundness of $\vdash_t$ and apply appropriate Hoare rules until we are left with three verification conditions: namely that the precondition, after execution of $j :== (\lambda s.\ \mathit{null})$, implies the loop invariant

$$
\begin{array}{l}
(\exists xs\ ys.\ (\mathit{heap\text{-}list\ xs}\ (s\ i) \wedge\!* \mathit{heap\text{-}list\ ys}\ (s\ j))\ h \wedge (\mathit{rev\ vs}) = (\mathit{rev\ xs})@ys) \\
\wedge\ \mathit{distinct}\ [i,j,k]\ ,
\end{array}
$$

that the loop invariant is preserved during execution of the loop body, and finally that the loop invariant, together with $s\ i = \mathit{null}$, implies the postcondition.

$$
\begin{array}{l}
\textbf{Lemma}\ \mathit{heap\text{-}list\ vs\ a\ h} \Longrightarrow \exists xs\ ys.\ (\mathit{heap\text{-}list\ ys\ null} \wedge\!* \mathit{heap\text{-}list\ xs\ a})\ h \\
\qquad \wedge\ \mathit{rev\ vs} = \mathit{rev\ xs}\ @\ ys \\
\textbf{Lemma}\ (\mathit{heap\text{-}list\ ys\ j} \wedge\!* \mathit{heap\text{-}list}\ (x\ \#\ xs)\ i)\ h \Longrightarrow \\
\qquad (\mathit{heap\text{-}list\ xs}\ (\mathit{heap\text{-}lookup}\ h\ (\mathit{Suc}\ i)) \wedge\!* \mathit{heap\text{-}list}\ (x\ \#\ ys)\ i) \\
\qquad\qquad (\mathit{heap\text{-}update}\ h\ (\mathit{Suc}\ i)\ [j]) \\
\textbf{Lemma}\ (\mathit{heap\text{-}list\ xs\ null} \wedge\!* \mathit{heap\text{-}list\ ys\ a})\ h \Longrightarrow \mathit{heap\text{-}list}\ (\mathit{rev\ xs}\ @\ ys)\ a\ h
\end{array}
$$

The first and last lemma are easily proved with the help of simple properties of *heap-list*. The proof of the second lemma is more difficult. Using the definition

of separating conjunction, we obtain disjoint subheaps h' and h'' of h with *heap-list ys j* h' and *heap-list* $(x\#xs)$ i h''. The conclusion can then be shown by splitting *heap-update* h $(Suc\ i)$ $[j]$ into the two disjoint heaps $h''(i{:=}None, Suc\ i{:=}None)$ and $h'(i{\mapsto}x)(Suc\ i{\mapsto}j)$. Overall the separation logic proof is slightly less automatic than the proof in [11].

At the moment, the proof strategy employed here seems to be characteristic of formal program verification with separation logic. First Hoare rules are used to obtain a set of verification conditions; this step could easily be automated for programs with loop annotations. Some of the verification conditions can then be shown using simple algebraic properties (e.g. commutativity, associativity) of separating conjunction and implication and the involved predicates, while others presently require semantic arguments. Although it is known that separation logic is not finitely axiomatizable [7], we hope that further case studies will allow us to identify other useful laws of the separating connectives, so that the need for (usually involved) semantic arguments can be minimized.

6 Conclusions and Future Work

This work is a first step towards the use of separation logic in machine-assisted program verification. We have mechanically verified semantic properties of separation logic, and presented three different Hoare logics for pointer programs, all of which we proved sound and relative complete. The whole development, including a formal proof of the Frame Rule, was carried out in the semi-automatic theorem prover Isabelle/HOL.

From our experience, separation logic can be a useful tool to state program specifications in a short and elegant way. At this time, however, the advantage of concise specifications comes with a cost: verification proofs, when carried out at the level of detail that is required for mechanical verification, tend to become more intricate and less automatic. Further work is necessary to achieve a better integration of separation logic into the existing Isabelle/HOL framework, and to increase the degree of proof automation for the connectives of separation logic.

More immediate aims are a verification condition generator for an annotated version of the language, some syntactic sugar for the connectives of separation logic, and extensions to the programming language, e.g. recursive procedures and concurrency.

Acknowledgments The author would like to thank Tobias Nipkow, Farhad Mehta and the anonymous referees for their valuable comments.

References

[1] Krzysztof R. Apt. Ten years of Hoare's logic: A survey – part I. *ACM Transactions on Programming Languages and Systems*, 3(4):431–483, October 1981.

[2] Krzysztof R. Apt. Ten years of Hoare's logic: A survey – part II: Nondeterminism. *Theoretical Computer Science*, 28:83–109, 1984.

[3] Clark W. Barrett, David L. Dill, and Jeremy R. Levitt. Validity checking for combinations of theories with equality. In Mandayam K. Srivas and Albert J. Camilleri, editors, *Formal Methods in Computer-Aided Design*, volume 1166 of *Lecture Notes in Computer Science*, pages 187–201. Springer, November 1996.

[4] Lars Birkedal, Noah Torp-Smith, and John C. Reynolds. Local reasoning about a copying garbage collector. In *Proceedings of the 31-st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 220–231. ACM Press, January 2004.

[5] Richard Bornat. Proving pointer programs in Hoare logic. In *Mathematics of Program Construction*, pages 102–126, 2000.

[6] Rodney M. Burstall. Some techniques for proving correctness of programs which alter data structures. In Bernard Meltzer and Donald Michie, editors, *Machine Intelligence*, volume 7, pages 23–50. Edinburgh University Press, Edinburgh, Scotland, 1972.

[7] Cristiano Calcagno, Hongseok Yang, and Peter W. O'Hearn. Computability and complexity results for a spatial assertion language for data structures. In Ramesh Hariharan, Madhavan Mukund, and V. Vinay, editors, *FST TCS 2001: Foundations of Software Technology and Theoretical Computer Science*, volume 2245 of *Lecture Notes in Computer Science*, pages 108–119. Springer, 2001.

[8] E. Dijkstra. Guarded commands, non-determinacy and formal derivation of programs. *Communications of the ACM*, 18:453–457, 1975.

[9] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.

[10] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, Inc., second edition, 1988.

[11] Farhad Mehta and Tobias Nipkow. Proving pointer programs in higher-order logic. In Franz Baader, editor, *Automated Deduction – CADE-19*, volume 2741 of *Lecture Notes in Artificial Intelligence*, pages 121–135. Springer, 2003.

[12] Tobias Nipkow. Winskel is (almost) right: Towards a mechanized semantics textbook. *Formal Aspects of Computing*, 10(2):171–186, 1998.

[13] Tobias Nipkow. Hoare logics in Isabelle/HOL. In H. Schwichtenberg and R. Steinbrüggen, editors, *Proof and System-Reliability*, pages 341–367. Kluwer, 2002.

[14] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[15] Peter W. O'Hearn, John Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In Laurent Fribourg, editor, *Computer Science Logic*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2001.

[16] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS'02)*, pages 55–74, 2002.

[17] Martin Wildmoser and Tobias Nipkow. Certifying machine code safety: shallow versus deep embedding. Accepted to the 17th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2004).

[18] Hongseok Yang. *Local Reasoning for Stateful Programs*. PhD thesis, University of Illinois, Urbana-Champaign, 2001.

[19] Hongseok Yang and Peter W. O'Hearn. A semantic basis for local reasoning. In *Foundations of Software Science and Computation Structure*, pages 402–416, 2002.