

Lightweight Separation

Holger Gast

Wilhelm-Schickard-Institut für Informatik
University of Tübingen
gast@informatik.uni-tuebingen.de

Abstract. Lightweight separation is a novel approach to automatic reasoning about memory updates in pointer programs. It replaces the spatial formulae of separation logic, which complicate automation, by independent assertions about the memory content and the memory layout. As a result, assertions about the content can be treated by existing reasoners. The effect of memory updates is evaluated using specialized tactics that prove disjointness of memory regions from a given memory layout.

1 Introduction

Separation logic [20, 17] has proven a powerful tool for specifying and verifying algorithms that work on mutable heap data structures. Its spatial formulae capture the structure and content of the heap precisely: $E \mapsto F$ holds on heaps with domain $\{E\}$ where F is stored at address E ; spatial conjunction $P * Q$ holds on heaps that can be split such that P and Q hold for the disjoint parts. Specifications in separation logic are tight, i.e. a correct program never accesses memory outside of the heap specified in its precondition. A general frame rule then allows local specifications of procedures to be adapted at the call site.

Separation logic has a direct shallow embedding into higher-order logic [23, 1, 22, 2], but reasoning about programs requires a substantial amount of manual interaction, despite the sophisticated automation available in current proof assistants. Different reasons for this shortcoming have been identified in the literature. Appel [1] observes that separation logic is a linear logic, such that classical reasoners do not apply. Tuch [21] points out that the definition of spatial conjunction contains existential quantification, which is known to require human insight in proofs. Weber [23] argues that separation logic is not finitely axiomatizable [8], hence the goal can only be a comprehensive library of theorems.

We will investigate the pragmatic view that the spatial formulae of separation logic restrict access to those formulae that existing reasoners would handle well: the assertions about the content of memory are at least partially hidden below spatial conjunctions. Our goal is therefore to replace spatial formulae by assertions about the memory layout, such that the memory content can be specified independently in classical higher-order logic. This approach is lightweight in that it does not impose a structure on verification rules, assertions, and proofs in the way that separation logic does (e.g. [1, 2]).

The approach poses the obvious challenge that each assertion about the memory content is potentially affected by each memory update in the program. We use conditional rewrite rules to evaluate the effects of memory updates:

$$\frac{\text{is-valid (block a len || typed-block } \Gamma \text{ a' t)}}{\text{rd } \Gamma \text{ a' t (STORE a len M' M) = rd } \Gamma \text{ a' t M}}$$

In this rule, `rd` fetches a typed value from address `a` and `STORE` represents one update of `len` bytes at address `a`. `M` is the current, `M'` a previous memory state. The update can be discarded if the read and the modified memory regions do not overlap, which is expressed in the rule's premise. The disjointness in the premise will be derived from the given memory layout by specialized tactics.

This paper's contribution is a framework, developed in Isabelle/HOL, which uses memory layouts to simplify the effect of memory updates, to discharge side-conditions on the allocatedness of memory regions, and which maintains the layouts themselves through memory updates, allocation, and deallocation. We apply the approach to a low-level language with memory-allocated local variables, references to variables, and structured data types.

Organization of the Paper Section 2 surveys Isabelle's notation and simplifier. Section 3 describes the low-level language that serves as an application of our framework. Section 4 formalizes memory layouts and derives their properties. Section 5 describes the tactics for reasoning about the effects of memory updates. Section 6 applies the framework to the in-place list reversal algorithm. Section 7 surveys related work. Section 8 discusses future work and concludes.

2 Isabelle Syntax and Simplifier

The syntax of Isabelle/HOL follows standard mathematical conventions, with a few exceptions. In Isabelle's meta-logic, $\llbracket P_1; \dots; P_n \rrbracket \Longrightarrow Q$ denotes implication of Q by P_1, \dots, P_n . Meta-level forall quantification of P over x_1, \dots, x_n is written $\bigwedge x_1, \dots, x_n. P$. $s \equiv t$ is meta-level equality. The type of total functions from τ to τ' is written $\tau \Rightarrow \tau'$. $\tau \rightarrow \tau' = \tau \Rightarrow \tau'$ **option** is used for partial functions. Type variables are written $'a, 'b, \dots$. Type $'a$ **set** denotes the sets over $'a$. $\{a .. < b\}$ is the half-closed interval $[a, b)$.

Isabelle's simplifier [19, §10] is a flexible rewriting engine whose operation is controlled by *simpsets*. We use three advanced elements of simpsets: the *premises*, *simplification procedures*, and the *subgoaler*. The premises of a simpset are theorems representing the local assumptions valid at the *redex*. When the simplifier is invoked on a goal $\llbracket P_1; \dots; P_n \rrbracket \Longrightarrow Q$, it simplifies each P_i using the remaining P_j as premises, and simplifies Q with premises P_1, \dots, P_n . Simplification procedures [19, §10.2.5] are ML functions that compute rewrite rules on demand. Simpsets associate simplification procedures with term patterns. When a *redex* matches a pattern, the simplifier calls the corresponding procedure with the current simpset and the *redex*. The procedure may then prove a theorem that the simplifier will apply as a rewrite rule. The *subgoaler* [19, §10.2.7] of a simpset

is a specialized tactic for solving the premises of conditional rewrite rules. To apply a rule $\llbracket C_1; \dots; C_n \rrbracket \Longrightarrow A \equiv B$, the simplifier matches A against the redex and calls the subgoal with the instantiated premises $C_1\sigma, \dots, C_n\sigma$ and the current simpset.

3 A Low-Level Language

L_0 is a prototypical low-level language that shares many aspects with C, but ignores its semantic subtleties [16]. Its main features are memory-allocated local variables and an unrestricted address operator. Since our focus is on reasoning about memory updates, the presentation in this section is necessarily brief.

3.1 Types and Contexts

Types in L_0 are used to determine the size of data objects.

```
datatype ty = TVoid | TNat | TBool | TPtr ty | TStruct string
```

Contexts, denoted by Γ , store the definitions of structs, and the types and addresses of local variables. The representation of local variables as an association list simplifies reasoning about block-structured allocation and deallocation.

```
record ctx =
  ctx-structs :: "string  $\rightarrow$  (string  $\times$  ty) list"
  ctx-vars    :: "(string  $\times$  (addr  $\times$  ty)) list"
```

The size of types is given by inductively defined predicates $\mathit{tysz} \Gamma t l$ and $\mathit{tyszs} \Gamma F l$, where t is a type, F is a list of fields, and l is the length as a natural number. Type `TVoid` has size 0, the primitive types have size 1. `tyszs` gives the size of a struct type as the sum of the field sizes. The auxiliary predicate `field-data $\Gamma t f d l t'$` yields the offset d , the length l , and the type t' for field f of struct type t in context Γ . A type t is *well-formed* in context Γ , written $\mathit{wf-ty} \Gamma t$, if it has a size. This definition allows the use of C-style incomplete types. The definition of `tysz` and `tyszs` as a least fixpoint ensures that well-formed types have no cyclic references. The predicate `is-field $\Gamma t f$` asserts that struct type t is well-formed in Γ and has a field f . The function `sz-of-ty $\Gamma t = (\mathit{THE} s. \mathit{tysz} \Gamma t s)$` facilitates reasoning about type sizes using the definite description operator from Isabelle/HOL. Likewise, `addr-of Γx` and `type-of Γx` , and `sz-of Γx` yield the address, type, and size of local variable x in context Γ using `the`.

3.2 Memory Model

Memory is a partial mapping from addresses to numbers. Values are represented as lists of numbers [16, 22]. Memory operations are total functions `memory \Rightarrow memory`, but they mark memory invalid for accesses outside of the domain. A memory state is *well-formed* if it is valid and has a finite domain.

```
record memory =
  m-dom  :: addr set
  m-cnt  :: addr  $\Rightarrow$  nat
  m-valid :: bool

types
  addr = nat
  val  = nat list
  wf-mem M  $\equiv$  m-valid M  $\wedge$  mem-finite M
```

The field `m-valid` is a history variable [21]: it does not influence the behaviour of memory operations, but records illegal accesses. Side-conditions in the Hoare-logic will guarantee that correct programs do not invalidate memory.

The operations `fetch` and `store` follow `heap-list` and `heap-update` from [22].

```

fetch :: addr ⇒ nat ⇒ memory ⇒ (nat list × memory)
fetch a 0 M = ([],M)
fetch a (Suc l) M = (let (vs',M') = fetch (a + 1) l M
                      in if a ∈ m-dom M
                          then (m-cnt M a # vs', M')
                          else (arbitrary # vs', M' (| m-valid := False |)))

store :: addr ⇒ nat list ⇒ memory ⇒ memory
store a [] M = M
store a (v # vs) M = (let M' = store (Suc a) vs M
                      in if a ∈ m-dom M'
                          then M' (| m-cnt := (m-cnt M') (a := v) |)
                          else M' (| m-valid := False |))

```

The function `alloc` abstracts over the allocation strategy using Hilbert's choice operator. Deallocation restricts the domain of the memory state.

```

alloc :: nat ⇒ memory ⇒ (addr × memory)
alloc l M ≡ (let c = (ε a. { a ..< a + 1 } ∩ m-dom M = { })
              in (c, M (| m-dom := m-dom M ∪ { c..< c + 1 } |)))

dealloc :: addr ⇒ nat ⇒ memory ⇒ memory
dealloc a l M ≡ (if { a ..< a+1 } ⊆ m-dom M
                 then M (| m-dom := m-dom M - { a ..< a+1 } |)
                 else M (| m-valid := False |))

```

The terms `fetch-val a l M`, `fetch-M a l M`, `alloc-a a M`, and `alloc-M a M` abbreviate the first and second components, respectively, of the corresponding `fetch` and `alloc`.

3.3 State Updates for Forward-Reasoning

Separation logic is usually formulated with forward-style verification rules [17, 20, 5, 1]. Logics for forward reasoning have not been discussed extensively in the literature on mechanized Hoare logics, but Floyd's assignment axiom generalizes readily.

$$\{ P \} x := e \{ \exists x'. P[x'/x] \wedge x = e[x'/x] \}$$

The post-condition here is obtained by replacing references to x in both the pre-condition and the (side-effect-free) expression e by an existentially quantified variable x' denoting the old value of x . The generalization consists in inverse operators that cancel the effects of memory operators from Section 3.2. The definition of `ALLOC` is the definition of `dealloc`, and we omit it for brevity.

```

STORE :: addr ⇒ nat ⇒ memory ⇒ memory ⇒ memory
STORE a l M' M ≡ (if { a ..< a+1 } ⊆ m-dom M
                  then M (| m-cnt := λb. if b ∈ { a ..< a + 1 }
                                      then m-cnt M' b else m-cnt M b |)
                  else M (| m-valid := False |))

ALLOC :: addr ⇒ nat ⇒ memory ⇒ memory
DEALLOC :: addr ⇒ nat ⇒ memory ⇒ memory
DEALLOC a l M ≡ (if m-dom M ∩ { a ..< a+1 } = { }
                 then M (| m-dom := m-dom M ∪ { a ..< a+1 } |)
                 else M (| m-valid := False |))

```

3.4 Language, Semantics, and External Syntax

Expressions in L_0 imitate low-level intermediate languages. Type annotations at all operators and constants determine the size of the handled values. Variable types are kept in the context. Primitive operators are given explicitly by values of types `prim1` and `prim2`, which also contain the parameter and return types.

EVar string	EDeref ty exp	EPrim1 prim1 exp
ECnat nat	EAddr exp	EPrim2 prim2 exp exp
ENil ty	EAcc exp ty string	ENew ty
	EAssign ty exp exp	EFree ty exp

The statement language is standard and is omitted for brevity. It includes `if`, `while`, and block statements. Expressions can be executed as statements. Block statements contain local variable declarations as a list of names and types.

L_0 uses a standard big-step semantics for expressions and statements. Evaluation of expressions as l-values and r-values (in the sense of C, see [16]) is defined inductively by relations $\Gamma \vdash_l M, e \rightarrow a, M'$ and $\Gamma \vdash_r M, e \rightarrow v, M'$, where a is the address of the l-value, v is the r-value of type `val`, M is the old memory state and M' is the new memory state. The statement semantics is given by the relation $\Gamma \vdash_{sm} M, s \rightarrow M'$.

In the following, we highlight two properties that are crucial for the Hoare logic. Like any fetch operation, pointer dereferencing for an r-value may invalidate memory. `to_ptr` lifts a value of type `val` to type `addr` by taking its first element.

$$\begin{aligned} & \llbracket \Gamma \vdash_r M0, e \rightarrow vs', M1; \\ & \quad (vs, M2) = \text{fetch}(\text{to_ptr } vs') \text{ (sz-of-ty } \Gamma \text{ t) } M1 \\ & \rrbracket \implies \Gamma \vdash_r M0, \text{EDeref t } e \rightarrow vs, M2 \end{aligned}$$

Local variables are memory-allocated. The execution of block statements uses `alloc-var` and `dealloc-var`, defined in terms of `alloc` and `dealloc`. Declaration d is a pair of name and type; the type determines the size of the allocated memory.

$$\begin{aligned} & \llbracket (x, M1) = \text{alloc-var } \Gamma0 \text{ d } M0; \\ & \quad \Gamma1 = \Gamma0 \text{ (| ctx-vars := x \# ctx-vars } \Gamma0 \text{ |)}; \\ & \quad \Gamma1 \vdash_{sm} M1, \text{SBlock decls sm} \rightarrow M2; \\ & \quad M3 = \text{dealloc-var } \Gamma1 \text{ x } M2 \\ & \rrbracket \implies \Gamma0 \vdash_{sm} M0, \text{SBlock (d \# decls) sm} \rightarrow M3 \end{aligned}$$

L_0 expressions are very verbose. We therefore define an external syntax [19, §6.1] for statements, expressions, and types, and register a parse-translation function [19, §8.6] that creates the internal representation. It includes a type-checker that resolves overloading of primitive operators and computes type annotations by Hindley/Milner type inference. Assertions in Hoare triples are also pre-processed: names of declared local variables are replaced by a read access to the variables. A corresponding print-translation reverses these effects for display.

3.5 Hoare Logic

The following memory access functions are used in assertions. `rd` and `rdv` yield a value from address a and variable x , respectively. `tyval` asserts that a value has the correct length for a given type.

$$\begin{aligned} \text{rd } \Gamma \text{ a t} &\equiv \lambda M. \text{fetch-val a (sz-of-ty } \Gamma \text{ t) M} \\ \text{rdv } \Gamma \text{ x} &\equiv \lambda M. \text{rd } \Gamma \text{ (addr-of } \Gamma \text{ x) (type-of } \Gamma \text{ x) M} \\ \text{tyval } \Gamma \text{ vs t} &\equiv (\text{length vs} = \text{sz-of-ty } \Gamma \text{ t}) \end{aligned}$$

The inverse operator **STORE** is lifted to **STORE-TYPED** $\Gamma \text{ a t}$ and **STORE-VAR** $\Gamma \text{ x}$. Analogous lifted versions exist for **ALLOC** and **DEALLOC**. The inverse operator **ADD-VAR** $\text{x } \Gamma$ cancels the addition of x to Γ ; **DEL-VAR** $(\text{x,t}) \text{ a } \Gamma$ re-inserts the variable x with type t and address a .

The assertions in Hoare triples for statements are of type $\text{ctx} \Rightarrow \text{memory} \Rightarrow \text{bool}$. Post-conditions on expressions have access to the computed result [13]. Well-formedness of memory is an implicit invariant of all correct programs.

$$\begin{aligned} \models \{ P \} e \{ Q \} &\equiv (\forall \Gamma \text{ M M}' \text{ a. wf-mem M} \longrightarrow P \ \Gamma \text{ M} \\ &\quad \longrightarrow \Gamma \vdash \text{M}, e \rightarrow \text{a}, \text{M}' \longrightarrow \text{wf-mem M}' \wedge Q \ \Gamma \text{ M}' \text{ a}) \\ \models_r \{ P \} e \{ Q \} &\equiv (\forall \Gamma \text{ M M}' \text{ vs. wf-mem M} \longrightarrow P \ \Gamma \text{ M} \\ &\quad \longrightarrow \Gamma \vdash_r \text{M}, e \rightarrow \text{vs}, \text{M}' \longrightarrow \text{wf-mem M}' \wedge Q \ \Gamma \text{ M}' \text{ vs}) \\ \models_{\text{sm}} \{ P \} s \{ Q \} &\equiv (\forall \Gamma \text{ M M}'. \text{wf-mem M} \longrightarrow P \ \Gamma \text{ M} \\ &\quad \longrightarrow \Gamma \vdash_{\text{sm}} \text{M}, s \rightarrow \text{M}' \longrightarrow \text{wf-mem M}' \wedge Q \ \Gamma \text{ M}') \end{aligned}$$

Sound verification rules are obtained as lemmata about Hoare triples. We illustrate those aspects that concern memory reasoning. Pointer dereferencing for an r -value shows the expected side-condition on the allocatedness of the read region. The relation \triangleright is introduced in Section 4.1.

$$\begin{aligned} \llbracket \models_r \{ P_0 \} e \{ P_1 \} ; \\ \quad \forall \Gamma \text{ M vs. } P_1 \ \Gamma \text{ M vs} \longrightarrow \text{M} \triangleright \text{typed-block } \Gamma \text{ (to-ptr vs) t} \\ \rrbracket \implies \models_r \{ P_0 \} \text{EDeref t e } \{ \lambda \Gamma \text{ M vs. } \exists \text{vs}'. \text{vs} = \text{rd } \Gamma \text{ (to-ptr vs) t M} \wedge P_1 \ \Gamma \text{ M vs}' \} \end{aligned}$$

Assignment introduces the inverse operator **STORE** into the post-condition. The side-condition concerns the allocatedness of the modified memory region.

$$\begin{aligned} \llbracket \models \{ P_0 \} e_1 \{ P_1 \} ; \\ \quad \forall \text{a. } (\models_r \{ \lambda \Gamma \text{ M. } P_1 \ \Gamma \text{ M a} \} e_2 \{ P_2 \text{ a} \} \wedge \\ \quad \forall \Gamma \text{ M vs. } P_2 \text{ a } \Gamma \text{ M vs} \longrightarrow \text{M} \triangleright \text{typed-block } \Gamma \text{ a t} \wedge \text{tyval } \Gamma \text{ vs t}) \\ \rrbracket \implies \models \{ P_0 \} \text{EAssign t e}_1 e_2 \\ \quad \{ \lambda \Gamma \text{ M a. } \exists \text{M}' \text{ vs. } P_2 \text{ a } \Gamma \text{ (STORE a (sz-of-ty } \Gamma \text{ t) M}' \text{ M) vs} \wedge \text{vs} = \text{rd } \Gamma \text{ a t M} \} \end{aligned}$$

Block-statements allocate and deallocate local variables. The side-condition on the allocatedness of x could be replaced by the invariant that local variables remain allocated throughout their lifetime. For L_0 , we have opted for simplicity.

$$\begin{aligned} \llbracket \models_{\text{sm}} \{ \lambda \Gamma \text{ M. type-of } \Gamma \text{ x} = \text{t} \wedge P_0 \text{ (ADD-VAR } \text{x } \Gamma) \text{ (ALLOC-VAR } \Gamma \text{ x M)} \} \\ \quad \text{SBlock decls s } \{ P_2 \} ; \\ \quad \forall \Gamma \text{ M. } P_2 \ \Gamma \text{ M} \longrightarrow \text{M} \triangleright \text{var-block } \Gamma \text{ x} \\ \rrbracket \implies \models_{\text{sm}} \{ P_0 \} \text{SBlock } ((\text{x,t}) \# \text{decls}) \text{s} \\ \quad \{ \lambda \Gamma \text{ M. } \exists \text{a. } P_2 \text{ (DEL-VAR } (\text{x,t}) \text{ a } \Gamma) \text{ (DEALLOC-VAR (DEL-VAR } (\text{x,t}) \text{ a } \Gamma) \text{ x M)} \} \end{aligned}$$

4 Formalizing Memory Layouts

We split the presentation of lightweight separation into two parts. This section contains the definition and properties of covers and their use in formalizing memory layouts. The next Section 5 describes the tactics that use these properties to automate memory reasoning.

4.1 Covers

A cover is a predicate on address sets; it is *well-formed* if it yields true for a single set. We will call address sets *memory regions*, and speak of *the* memory region covered by a well-formed cover. A cover is *valid* if it covers some region.

```
types cover = addr set ⇒ bool
wf-cover A ≡ (∀ S S'. A S → A S' → S = S')
is-valid A ≡ (∃ S. A S)
```

Well-formed covers are interchangeable with the regions that they cover. It is this property which will enable many manipulations on memory layouts that would be invalid on spatial formulae of separation logic.

The relations $M \dot{=} A$, read “ M is covered by A ” and $M \triangleright A$, read “ A is allocated in M ” allow covers to be treated as descriptions of the memory layout. Both relations hold only on valid memory, because no assertion must be made about invalid, hence corrupt memory.

```
M ≐ A ≡ m-valid M ∧ A (m-dom M)
M ▷ A ≡ m-valid M ∧ (∃ S. S ⊆ m-dom M ∧ A S)
```

The *subcover* relation $A \preceq B$ asserts that A covers a part of the region covered by B and that validity of B implies validity of A . It is both reflexive and transitive for arbitrary covers and antisymmetric for well-formed covers.

```
A ≼ B ≡ ∀ S. B S → (∃ S'. S' ⊆ S ∧ A S')
```

Lemma (1) below reduces proving allocatedness of memory regions to proving a subcover relation if a memory layout is given. Lemma (2) uses antisymmetry to derive a complete layout B from a given layout A . The significance of the fact `is-valid A` in the third premise will be clarified in Section 4.2.

$$\llbracket M \dot{=} A; B \preceq A \rrbracket \Longrightarrow M \triangleright B \quad (1)$$

$$\llbracket M \dot{=} A; B \preceq A; \text{is-valid } A \rrbracket \Longrightarrow A \preceq B; \text{wf-cover } A \rrbracket \Longrightarrow M \dot{=} B \quad (2)$$

The *matchcover* relation $A \prec B$ is a weaker variant of subcover. It asserts that validity of B implies validity of A , but does not relate their covered regions. The tactics will employ it to reason about disjointness independently of allocatedness.

```
A ≺ B ≡ ∀ S. B S → (∃ S'. A S')
```

The following Lemma (3) reduces proving validity to proving a matchcover relation if a memory layout is given. Since valid layout expressions (Section 4.2) can capture disjointness, this lemma is central to lightweight separation.

$$\llbracket M \dot{=} A; B \prec A \rrbracket \Longrightarrow \text{is-valid } B \quad (3)$$

4.2 Layout Expressions

The memory layout is captured by nested covers, which we call *layout expressions*. The base of the layout expressions are blocks, which cover an interval of addresses. The empty cover covers the empty region.

```
block a l ≡ λS. S = {a ..< a + l}      Empty ≡ λA . A = {}
```

A typed-block $\Gamma \ t$ covers the block at a with the size of t ; a var-block $\Gamma \ x$ covers the memory region of variable x . Note that all blocks are well-formed and valid.

The following combinators of type $\text{cover} \Rightarrow \text{cover} \Rightarrow \text{cover}$ build nested layout expressions. A cover that is not constructed by one of them is a *layout block*.

$$\begin{aligned} "A \parallel B \equiv \lambda S. \exists S1 \ S2. A \ S1 \wedge B \ S2 \wedge S = S1 \cup S2 \wedge S1 \cap S2 = \{\}" \\ "A \setminus B \equiv \lambda S. \exists S1 \ S2. A \ S1 \wedge B \ S2 \wedge S2 \subseteq S1 \wedge S = S1 - S2" \\ "A \mid B \equiv \lambda S. \exists S2. A \ S \wedge B \ S2 \wedge S \cap S2 = \{\}" \end{aligned}$$

The cover $A \parallel B$, read *disjoint*, covers the union of the disjoint memory regions covered by A and B . Note that it parallels spatial conjunction of separation logic, but does not consider the memory content. As an operator, \parallel is both associative and commutative. $A \parallel B$ is well-formed if both A and B are well-formed.

The operators *clip* $A \setminus B$ and *weak disjointness* $A \mid B$ capture memory deallocation. The cover $A \setminus B$ asserts that the memory region covered by B has been deallocated, but that region is still covered by A . The cover $A \mid B$, on the other hand, asserts that the memory region covered by A does not contain the region covered by B .¹ Both $A \mid B$ and $A \setminus B$ are well-formed if A and B are well-formed.

The tactics of Section 5 use clipping as an intermediate step towards the layout after deallocation. The following lemmata provide the basis for removing clip. (4) shows that removing A from A yields an empty block, which is, of course, disjoint from A . The Lemmata (5) allow this replacement to occur recursively, on both side of \parallel and on the left, the allocated side, of \mid . Note that the deallocated part C is kept with the already deallocated D in this last case.

$$\llbracket \text{wf-cover } C; \text{is-valid } C; A = C \rrbracket \Longrightarrow A \setminus C = \text{Empty} \mid C \quad (4)$$

$$\begin{aligned} \llbracket \text{wf-cover } C; C \preceq A; (A \setminus C) = A' \rrbracket &\Longrightarrow ((A \parallel B) \setminus C) = (A' \parallel B) \mid C \\ \llbracket \text{wf-cover } C; C \preceq B; B \setminus C = B' \rrbracket &\Longrightarrow ((A \parallel B) \setminus C) = (A \parallel B') \mid C \\ \llbracket \text{wf-cover } C; C \preceq A; (A \setminus C) = A' \rrbracket &\Longrightarrow ((A \mid D) \setminus C) = (A' \mid (D \parallel C)) \mid C \end{aligned} \quad (5)$$

Several applications of weak disjointness in a row are redundant.

$$\begin{aligned} ((A \mid C) \parallel B) \mid C &= (A \parallel B) \mid C \\ (A \parallel (B \mid C)) \mid C &= (A \parallel B) \mid C \\ ((A \mid C) \mid (B \parallel C)) \mid C &= (A \mid (B \parallel C)) \mid C \end{aligned} \quad (6)$$

Proving allocatedness and disjointness is reduced to proving subcovers by Lemmata (1) and (3). The right-hand side then is a known memory layout, and the left-hand side must be proven to be a fragment of it. Lemmata (7) is used to focus on parts of the right-hand-side. Lemma (8) allows subcovers on disjoint regions to be proven independently. Lemma (9) shows that the covered region is immaterial for matchcover.

$$\begin{array}{cccc} A \preceq A \parallel B & A \preceq A \mid B & A \prec A \parallel B & A \prec A \mid B \\ B \preceq A \parallel B & & B \prec A \parallel B & B \prec A \mid B \end{array} \quad (7)$$

$$\llbracket A \preceq A'; B \preceq B' \rrbracket \Longrightarrow A \parallel B \preceq A' \parallel B' \quad (8)$$

$$A \parallel C \prec A \mid C \quad (9)$$

¹ Weak disjointness avoids a well-known incompleteness [5] of separation logic. The triple $\{E \mapsto F * E' \mapsto F'\} \text{dispose}(E) \{\text{emp} * E' \mapsto F'\}$ is provable, but loses information, since postcondition does not imply $E \neq E'$. Operator $A \mid B$ is defined to maintain just this lost information.

Lemmata (10) and (11) parallel (7) and (8) for the left-hand-side. Their repeated application removes weak disjointness from the third premise of (2). Note that cover B in (2) will not contain the combinator $|$ during forward reasoning.

$$\llbracket \text{is-valid } (A \mid B); \text{ is-valid } A \implies A \preceq A'; \text{ wf-cover } A \rrbracket \implies A \mid B \preceq A' \quad (10)$$

$$\llbracket \text{is-valid } (A \parallel B); \text{ is-valid } A \implies A \preceq A'; \text{ is-valid } B \implies B \preceq B' \rrbracket \implies A \parallel B \preceq A' \parallel B' \quad (11)$$

5 Simplification of Memory Updates

This section describes the specialized tactics that simplify memory updates in verification conditions. Sections 5.1 and 5.2 reduce this problem to proving sub-covers and eliminating clip. Sections 5.3– 5.5 describe the implemented tactics.

5.1 A Framework of Modifiers and Accessors

The Hoare logic introduces several *memory operators*, which take memory states to memory states, and several *memory accessors*, which read a particular value from memory. We define two types:

```
mem-op = memory  $\Rightarrow$  memory
'a mem-acc = memory  $\Rightarrow$  'a
```

The following two predicates characterize the behaviour of memory operators and memory accessors using covers: **modifies** asserts that **mop** does not change the memory state outside the region covered by A , **accesses** asserts that the result value does not depend on the memory state outside of region A .

```
modifies mop M A  $\equiv$  eqv-outside A M (mop M)
accesses ac M A  $\equiv$   $\forall M'. \text{eqv-inside } A M M' \longrightarrow \text{ac } M = \text{ac } M'$ 
```

The equivalence relation $\text{eqv-inside } A M M'$ checks that $\text{m-dom } M a = \text{m-dom } M' a$ and $\text{m-cnt } M a = \text{m-cnt } M' a$ for all addresses a covered by A . eqv-outside does the same for all addresses not covered by A . With these definitions, we have:

$$\begin{aligned} & \llbracket \text{modifies mop } M A; \text{accesses mac } M B; \\ & \quad \text{wf-cover } A; \text{wf-cover } B; \\ & \quad \text{is-valid } (A \parallel B) \\ & \rrbracket \implies \text{mac } (\text{mop } M) = \text{mac } M \end{aligned} \quad (12)$$

Lemma (12) serves as a generator for rewrite rules that simplify memory update operators in assertions. For each accessor and operator from Sections 3.3 and 3.5, we prove a lemma of the following form.

```
lemma STORE-TYPED-modifies[sepmop]:
  "modifies (STORE-TYPED  $\Gamma$  a t M') M (typed-block  $\Gamma$  a t)"
lemma rd-accesses[sepacc]:
  "accesses (rd  $\Gamma$  a t) M (typed-block  $\Gamma$  a t)"
```

The attributes `sepmop` and `sepacc` declare the lemmata as modifier and accessor theorems, respectively. The framework resolves each pair of modifier/accessor theorems against the first two premises of Lemma (12), solves the `wf-cover` premises

by theorems with attribute `sepwfcover`, and uses the result as a conditional rewrite rule like (13). The implemented tactics (Section 5.5) register a subgoal that proves the premise from a given memory layout.

$$\begin{aligned} & \text{is-valid (typed-block } \Gamma \text{ a t} \parallel \text{typed-block } \Gamma' \text{ a' ta)} \\ \implies & \text{rd } \Gamma' \text{ a' ta (STORE-TYPED } \Gamma \text{ a t M' M)} = \text{rd } \Gamma' \text{ a' ta M} \end{aligned} \quad (13)$$

5.2 Simplifying the Memory Layouts

The memory state M obviously appears in assertions $M \doteq A$, such that these terms need to be simplified as well. For the left-hand-side, we can derive rewrite rules for the inverse operators from Sections 3.3 and 3.5 based on these:

$$\begin{aligned} \text{block a l} \preceq A & \implies (\text{STORE a l M' M} \doteq A) = (M \doteq A) \\ (\text{ALLOC a l M} \doteq A) & = (M \doteq A \parallel \text{block a l}) \\ (\text{DEALLOC a l M} \doteq A) & = (M \doteq A \setminus \text{block a l}) \end{aligned}$$

Note how allocation and deallocation directly influence the layout expressions describing the memory layout, while `STORE` has the obvious proof obligation as a premise. The `clip` operator will be removed in Section 5.4.

The assertion $M \doteq A$ may contain, for example, `list-cover` (Section 6), and therefore M itself, within A . We can prove a `sepacc` lemma for `list-cover`, but no simplification can take place, since the memory layout $M \doteq A$ is not available as a premise during simplification of A . The solution is to register a simplification procedure (Section 2) that produces rewrite rules $M \doteq A \equiv M \doteq A'$, where the memory operators in A have been removed in A' . The procedure selects each layout block B of A in turn, and computes \hat{A} with $A = \hat{A} B$. It then proves:

$$M \doteq A = (\exists C. M \doteq \hat{A} C \wedge C = B) \stackrel{(\dagger)}{=} (\exists C. M \doteq \hat{A} C \wedge C = B') = M \doteq \hat{A} B'$$

The equality (\dagger) is resolved using variants of the Isabelle/HOL congruence rules for \exists and \wedge . The result is a goal of the form

$$M \doteq \hat{A} C \implies (C = B) = (C = B')$$

Now the rewrites from Section 5.1 can be applied to B . Any disjointness derivable from A that does not involve B is also derivable from $\hat{A} C$. Layout blocks like `list-cover`, however, access their own covered area. Lemma (14) provides the missing rewrite rules, using again pairs of modifier/accessor theorems.

$$\begin{aligned} & \llbracket \text{modifies mop M A; } \wedge M'. \text{ accesses mac M' (mac M')}; \\ & \quad \text{wf-cover A; } \wedge M'. \text{ wf-cover (mac M')}; \\ & \quad \text{is-valid (A} \parallel \text{C)} \\ & \rrbracket \implies (C = \text{mac (mop M)}) = (C = \text{mac M}) \end{aligned} \quad (14)$$

5.3 Proving Subcover Relations

The basis of automation of memory reasoning is the tactic `prove_subcover`, which solves goals of the form $A \preceq B$ and $A \prec B$. The procedure is motivated

by example goals that arise from applications of Lemmata (3), (1), and (2).

$$A \parallel B \preceq B \parallel C \parallel A \parallel F \quad (15)$$

$$A \parallel B \parallel E \preceq ((A \parallel B) \mid C) \parallel E \mid F \quad (16)$$

$$A \parallel B \prec B \parallel C \mid (A \parallel F) \quad (17)$$

$$A \parallel B \parallel E \parallel F \prec ((A \parallel B) \mid C) \parallel E \mid F \quad (18)$$

$$A \parallel B \parallel C \prec ((A \parallel B) \mid C) \parallel E \mid F \quad (19)$$

In the following, we leave applications of transitivity and reflexivity implicit. Goal (15) has only \parallel operators on both sides of the subcover relation. It suffices to rearrange the layout blocks using commutativity and associativity into $A \parallel B \parallel \text{Empty} \preceq A \parallel B \parallel C \parallel F$ and to apply (8) repeatedly to achieve trivial goals. Goal (16) is resolved like Goal (15), after the weak disjointness operators on the right have been removed by Lemmata (7) and (8). Goal (17) exploits the top-level weak disjointness; after an application of (9), the goal has the form of (15). Goal (18) is a combination of Goals (17) and Goal (16), in that the nested weak disjointness can be discarded while the top-level one must be exploited. Goal (19) is solved by bringing the inner weak disjointness to the top using Lemmata (7), and then proceeding in (17).

Note also that the following Goal (20) can *not* be derived, because block E may have been allocated in the region covered by the deallocated C .

$$C \parallel E \prec ((A \parallel B) \mid C) \parallel E \mid F \quad (20)$$

Applications of Lemma (2) during forward resolution can lead to subcover goals with weak disjointness on the left, as in Goal (21) below. However, these occurrences are removed by Lemmata (10) and (11).

$$\text{is-valid } (B \parallel C \mid (A \parallel F)) \implies B \parallel C \mid (A \parallel F) \preceq B \parallel C \quad (21)$$

These examples motivate the following procedure for proving $A \preceq B$ or $A \prec B$.

1. *Remove weak disjointness on the left-hand-side.* (Goal (21))
2. *Match the layout blocks of A with the layout blocks of B .*
Determine the layout blocks A_1, \dots, A_n and B_1, \dots, B_m of A and B . For each $i = 1 \dots n$, find a block B_{j_i} such that theorem $A_i \preceq B_{j_i}$ can be proven. Store also the paths p_i and q_{j_i} of A_i and B_{j_i} from the root of the layout expression. Fail if $j_i = j_{i'}$ for some i, i' .
3. *Drop common the prefix of q_{j_1}, \dots, q_{j_n} from B .* (Goal (19))
4. *Convert \prec to \preceq .* (Goal (17))
5. *Remove weak disjointness on the right-hand-side.* (Goals (16), (20))
6. *Align matching layout blocks on left- and right-hand-side.*
Use commutativity and associativity and the path information from Step 2.
7. *Solve the goal recursively.*
Use (Lemma 8) and the theorems from Step 2.

Step 2 contains the opportunity for automatic unfolding. Suppose that A_i and $A_{i'}$ are different fields in the same struct covered by B_j . By definition of the struct, we can prove $A_i \preceq B_j$ and $A_{i'} \preceq B_j$. The the resulting equality $j_i = j_{i'}$ then serves as a trigger for unfolding of B_j into its constituent fields.

5.4 Simplifying Clip

The clip operator $A \setminus B$ leaves implicit which part covered by A has been deallocated. The cover must be simplified into some $A' | B$ in which the part of A matching B does no longer occur in A' . Here are typical examples:

$$(A \| B \| C \| D) \setminus A = (B \| C \| D) | A \quad (22)$$

$$(A \| B | C) \setminus A = (B | (C \| A)) \quad (23)$$

$$(A \| B | (C \| D)) \setminus A = (B | (C \| D \| A)) \quad (24)$$

In Example (22), one of the allocated blocks is removed. Example (23) demonstrates that the term $(B | C | A)$ is not a solution, because it loses the information that A and C are disjoint. Example (24) applies this insight again.

A simplification procedure `clipproc` computes the above rewrite rules as follows. Given a cover $A \setminus B$, it recursively searches for the layout block B within A . When called on $C \setminus A$ and A is found in C , the search returns two theorems: $C \setminus B = C' | B$ and $B \preceq C$. In the base case, it uses Lemma (4) and reflexivity of subcover. In the recursion step, it resolves the two theorems obtained from the recursive call against the premises of one of the Lemmata (5).

5.5 Implemented Tactics

We have implemented the lightweight separation framework in the form of four tactics. Tactic `sep` invokes the Isabelle simplifier with the conditional rewrite rules from Section 5.1, the simplification procedure from Section 5.4, and a special subgoal. The subgoal resolves the `is-valid-premise` of a rewrite rule (13) by Lemma (3), resolves that lemma's first premise with a premise from the passed simpset, and proves the remaining matchcover relation by the procedure from Section 5.3. Tactic `ctx` registers only the simplification procedure from Section 5.2. Tactic `lift` rewrites the goal with rules declared as `lift`, among them the lifts from `STORE` to `STORE-TYPED` and `STORE-VAR` from Section 3.5.

Tactic `hoare` implements a verification condition generator for forward reasoning. It repeatedly applies one verification rule from Section 3.5 and then applies tactics `lift`, `ctx`, and `sep` to the resulting post-condition. It stops if some inverse operator cannot be removed, if a non-trivial side-condition is to be proven, and after the `if` and `while` rules have introduced the outcome of the test into the precondition. At these points, the user can apply arbitrary tactics before resuming the verification by another invocation of `hoare`.

6 Example: List Reversal

The algorithm for the reversal of a singly-linked list has been used frequently for comparing different approaches to the verification of pointer programs [6, 14, 1, 21]. Figure 1 states the specification and algorithm in L_0 (Section 3.4). The `VERIFY` keyword triggers the parse translation from the external syntax, the

```

lemma list-reversal:
  "∀XS. VERIFY VAR p : * struct node; VAR q : * struct node;
  |=sm { node-known Γ ∧ M ≐ p || q || list-cover Γ p M ∧ list-vals Γ p XS M }
  BEGIN
    q = nil;
    {INV node-known Γ ∧ M ≐ p || q || list-cover Γ p M || list-cover Γ q M ∧
      (∃PS QS. list-vals Γ p PS M ∧ list-vals Γ q QS M ∧ XS = (rev QS) @ PS) }
    WHILE p != nil DO
      BEGIN VAR t : * struct node;
        t = (*p). next;
        (*p). next = q;
        q = p;
        p = t
      END
    END
  { M ≐ p || q || list-cover Γ q M ∧ list-vals Γ q (rev XS) M }"

```

Fig. 1. List Reversal

subsequent VAR declarations are used to type-check the program. The outermost forall quantifier introduces the auxiliary variable XS [14].

The assertions in the example demonstrate the independent treatment of memory layout and memory content in lightweight separation. The content of a list is captured in the standard way (e.g. [14, 21]) by an inductively defined predicate $\text{list-vals } \Gamma \text{ p XS M}$, where the values of the HOL list XS are stored in the data fields of the singly-linked list starting at p. Note that the predicate can be used freely in HOL terms, e.g. below an existential quantifier in the invariant.

The assertions about the memory layout require a cover for the nodes of a linked list. The struct node with fields *data* (L_0 -type nat) and *next* (L_0 -type *struct node) represents list nodes. The predicate *node-known* states that struct node is defined in context Γ . The functions *node-data-rd* and *node-next-rd* read the fields' content. The cover for the list is given by the following inductive definition.

$$\begin{aligned}
 & \llbracket p = \text{nil}; S = \{\} \rrbracket \implies \text{list-cover } \Gamma \text{ p M S} \\
 & \llbracket p \neq \text{nil}; \quad (\text{typed-block } \Gamma \text{ (to-ptr p) (TStruct "node")}) \\
 & \quad \quad \quad \llbracket \text{list-cover } \Gamma \text{ (node-next-rd } \Gamma \text{ (to-ptr p) M) M} \rrbracket \text{ S} \\
 & \rrbracket \implies \text{list-cover } \Gamma \text{ p M S}
 \end{aligned}$$

To use the framework from Section 5.1, we show that *list-cover* is a well-formed cover and as a memory operator accesses the addresses covered by itself; *list-vals* accesses the corresponding *list-cover*.

$$\begin{aligned}
 \text{node-known } \Gamma & \implies \text{accesses (list-cover } \Gamma \text{ p) M (list-cover } \Gamma \text{ p M)} \\
 \text{node-known } \Gamma & \implies \text{accesses (list-vals } \Gamma \text{ p vs) M (list-cover } \Gamma \text{ p M)}
 \end{aligned}$$

The verification of the algorithm then consists of 17 steps, 7 of which are invocations of the **hoare** tactic. The main proof obligations are straightforward. After the existentially quantified PS has been instantiated to XS, Isabelle's **force** method proves that the loop invariant holds initially. Maintenance of the loop invariant is proven by **sep** for the memory layout and by **force** for the memory content. Note that the temporary variable t has been allocated and deallocated, and that the *list-cover* in the memory layout has been simplified.

Within the loop body, the side-conditions on allocatedness (Section 3.5) and the applications of `ctx` and `sep` (Section 5) require the memory layout in different levels of detail: For some, the individual `node` fields need to be exposed, others use the entire `list-cover`. The user must therefore fold or unfold these structures manually using equalities like the following.

```
node-known  $\Gamma \implies$ 
  typed-block  $\Gamma$  a (TStruct "node") =
    field-block  $\Gamma$  a (TStruct "node") "data" || field-block  $\Gamma$  a (TStruct "node") "next"
```

When the automatic unfolding mechanism sketched in Section 5.3 is implemented, the proof is reduced to 12 lines, 5 of which are applications of `hoare`, and the others solve the natural proof obligations described above.

7 Related Work

Appel [1] provides tactics to manipulate spatial conjunctions in forward verification. His approach is to lift pure assertions, which do not access the heap, to assertions about the empty heap. The tactics facilitate access to such pure spatial conjuncts: they extract equalities for rewriting, apply lemmata to specific conjuncts, and solve trivial implications between spatial formulae by rearranging conjuncts. The general problem remains that the application of Hoare rules places syntactic restrictions on the pre-condition, and these restrictions must be resolved by hand. The example verifications in [1, Section 5] therefore still require a number of technical insights.

Tuch et al. [22] use separation logic for reasoning about C programs by lifting an untyped memory to a typed view, where distinct types occupy disjoint memory regions [7]. They also lift proof obligations about raw memory updates to goals in separation logic [22, §5.2], but do not provide specific support for reasoning in separation logic. Tuch [21] extends the approach to structured types and provides rewrite rules that exploit disjointness of parts within a structured value. He reports using 67 lines for the verification of the list-reversal algorithm.

A different line of research aims at completely automatic verification [5, 4, 15] and shape analysis [9, 11, 3] using separation logic. These approaches focus on the shape of data structures and restrict formulae to the form $\Pi \wedge \Sigma$, where Π is a pure conjunction of pointer equalities and inequalities, and Σ is a spatial conjunction. Automation is achieved by a decidable entailment relation between the restricted formulae (e.g. [5, §4]). Unfortunately, the restrictions also preclude general assertions, such as about the values stored in a linked list.

While separation logic aims at describing the layout of heaps explicitly, many languages maintain the invariant that the data objects of different types and the different fields of the same struct do not overlap on the heap [7]. This insight has proven very successful for mechanized reasoning about pointer programs [10, 6, 14]. However, the assertions there need to contain explicit inequalities between pointers that must be reasoned about manually. Recently, this model has been complemented by a static analysis that discharges many of the remaining disjointness conditions [12]. Since the invariant cannot be circumvented even locally [22], these approaches do not apply to low-level programs.

8 Conclusion

Lightweight separation replaces the spatial formulae of separation logic by layout expressions that capture the disjointness of memory regions in classical higher-order logic. Assertions about the memory content likewise remain classical predicates, such that existing reasoners can be applied. A framework of specialized tactics removes memory updates from verification conditions and discharges side-conditions on allocatedness.

This paper has introduced the method of lightweight separation, focussing on the automation of memory reasoning. Two extensions will be described in companion papers. First, the formalization of (nested) structured datatypes with automatic unfolding (Sections 5.3 and 6). Second, the extension of L_0 by procedures, which require the following variant of the frame rule [20]: preconditions of procedures contain the memory layout $M \doteq R \parallel A$, where A is the region accessed by the procedure and R is an auxiliary variable [18] covering the remaining memory. Pre- and post-condition contain $\text{eqv-inside } R \ M_0 \ M$ with auxiliary variable M_0 . The framework generates additional simplification rules by instantiating the following lemma with modifier theorems:

$$\begin{array}{l} \llbracket \bigwedge M. \text{ modifies mop } M \ A; \\ \text{ is-valid } (A \parallel B); \\ \text{ wf-cover } A; \text{ wf-cover } B \\ \rrbracket \implies \text{eqv-inside } B \ M \ (\text{mop } M') = \text{eqv-inside } B \ M \ M' \end{array}$$

Since the procedure does not modify R , the post-condition $\text{eqv-inside } R \ M_0 \ M$ can be proven. By instantiating M_0 with the pre-state, and proving the arising tautology in the precondition, the caller gets the assertion that the post-state is unchanged from the pre-state within R , which can in turn be used to simplify the memory operators introduced by the procedure call.

References

1. A. W. Appel. Tactics for separation logic. <http://www.cs.princeton.edu/~appel/papers/septacs.pdf>, Jan. 2006.
2. A. W. Appel and S. Blazy. Separation logic for small-step C minor. In K. Schneider and J. Brandt, editors, *TPHOLs*, volume 4732 of *Lecture Notes in Computer Science*, pages 5–21. Springer, 2007.
3. J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. O’Hearn, T. Wies, and H. Yang. Shape analysis of composite data structures. In *CAV 2007*, volume 4590 of *LNCS*, Heidelberg, 2007. Springer.
4. J. Berdine, C. Calcagno, and P. W. O’Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, editors, *FMCO*, volume 4111 of *LNCS*. Springer, 2005.
5. J. Berdine, C. Calcagno, and P. W. O’Hearn. Symbolic execution with separation logic. In K. Yi, editor, *APLAS*, volume 3780 of *LNCS*, pages 52–68. Springer, 2005.
6. R. Bornat. Proving pointer programs in Hoare logic. In *Mathematics of Program Construction*, 2000.
7. R. Burstall. Some techniques for proving correctness of programs which alter data structures. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, number 7. Edinburgh University Press, 1972.

8. C. Calcagno, H. Yang, and P. W. O'Hearn. Computability and complexity results for a spatial assertion language for data structures. In *FST TCS '01: Proceedings of the 21st Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 108–119, London, UK, 2001. Springer.
9. D. Distefano, P. W. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In H. Hermanns and J. Palsberg, editors, *TACAS*, volume 3920 of *LNCS*, pages 287–302. Springer, 2006.
10. J.-C. Filliâtre and C. Marché. Multi-prover verification of C programs. In *Sixth International Conference on Formal Engineering Methods (ICFEM04)*, Nov. 2004.
11. A. Gotsman, J. Berdine, and B. Cook. Interprocedural shape analysis with separated heap abstractions. In K. Yi, editor, *SAS*, volume 4134 of *LNCS*, pages 240–260. Springer, 2006.
12. T. Hubert and C. Marché. Separation analysis for deductive verification. In *Heap Analysis and Verification (HAV'07)*, Braga, Portugal, Mar. 2007.
13. T. Kowaltowski. Axiomatic approach to side effects and general jumps. *Acta Informatica*, 7:357–360, 1977.
14. F. Mehta and T. Nipkow. Proving pointer programs in higher-order logic. *Inf. Comput.*, 199(1-2):200–227, 2005.
15. H. H. Nguyen, C. David, S. Qin, and W.-N. Chin. Automated verification of shape and size properties via separation logic. In B. Cook and A. Podelski, editors, *VMCAI*, volume 4349 of *LNCS*, pages 251–266. Springer, 2007.
16. M. Norrish. *C formalised in HOL*. PhD thesis, University of Cambridge, 1998. Technical Report UCAM-CL-TR-453.
17. P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Proceedings of the 15th International Workshop on Computer Science Logic*, number 2142 in *LNCS*, pages 1–19. Springer, 2001.
18. D. v. Oheimb and T. Nipkow. Hoare logic for NanoJava: Auxiliary variables, side effects and virtual methods revisited. In L.-H. Eriksson and P. A. Lindsay, editors, *Formal Methods Europe (FME'02)*, volume 2391 of *LNCS*. Springer, 2002.
19. L. C. Paulson. *Isabelle – A Generic Theorem Prover*. Number 828 in *LNCS*. Springer, Berlin Heidelberg, 1994.
20. J. 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)*, 2002.
21. H. Tuch. Structured types and separation logic. In *3rd International Workshop on Systems Software Verification (SSV 08)*, Feb. 2008.
22. H. Tuch, G. Klein, and M. Norrish. Types, bytes, and separation logic. In M. Hofmann and M. Felleisen, editors, *Proc. 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'07)*, Nice, France, Jan. 2007.
23. T. Weber. Towards mechanized program verification with separation logic. In J. Marcinkowski and A. Tarlecki, editors, *Computer Science Logic – 18th International Workshop, CSL 2004*, volume 3210 of *LNCS*. Springer, Sept. 2004.