

# Reasoning about Memory Layouts

Holger Gast

Received: 28.2.2010 / Accepted: 10.8.2010

**Abstract** Verification methods for memory-manipulating C programs need to address not only well-typed programs that respect invariants such as the split-heap memory model, but also programs that access through pointers arbitrary memory objects such as local variables, single struct fields, or array slices. We present a logic for memory layouts that covers these applications and show how proof obligations arising during the verification can be discharged automatically using the layouts. The framework developed in this way is also suitable for reasoning about data structures manipulated by algorithms, which we demonstrate by verifying the Schorr-Waite graph marking algorithm.

## 1 Introduction

Verification methods for programs that manipulate the heap necessarily formalize and reason about the memory layout: each access to the memory generates the proof obligation that the accessed region is allocated, and the influence of writes on the validity of assertions needs to be determined by considering the possible aliasing between pointers. The required reasoning has been automated successfully for Burstall's split-heap memory model [1], which in particular is expressive enough for object-oriented programming languages (e.g. [2,3]). Single objects as well as sets of objects are supported by current reasoning technology (e.g. [4]).

Burstall's memory model assumes that all pointers are object references and objects with different references do not overlap. It is therefore too imprecise for many C programs, and the employed reasoning techniques do not scale directly to more precise memory models [5]. Unfortunately, the excluded "low-level" usage is not confined to a few border cases, but is within the range of idiomatic C code. A few examples from a current Linux kernel, which are deliberately taken from different modules, will illustrate the point.

Data structures throughout the kernel are, for instance, protected by mutexes. The following functions (from `mutex.c`) acquire and release mutexes.

---

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

```
void mutex_lock(struct mutex *lock);
void mutex_unlock(struct mutex *lock);
```

The mutexes, i.e. memory objects of type `struct mutex`, are allocated in various ways. In `socket.c`, for instance, global mutexes protect the (global) `ioctl` settings. Calls like `mutex_lock(&br_ioctl_mutex)` are thus distributed throughout the module. But mutexes also protect `inodes` (defined in `fs.h`):

```
struct inode { ... struct mutex i_mutex; ... } (1)
```

Locking such an `inode` involves passing a pointer to a struct member:

```
mutex_lock(&inode->i_mutex); (2)
```

Furthermore, it is common to pass pointers to local variables and also to fields within local variables (from `hrtimer.c`, where `struct hrtimer_sleeper t`):

```
hrtimer_init_on_stack(&t.timer, /*... */); (3)
```

Also elements from local arrays are passed by reference (from `compat.h`, where `struct timespec tv[2]`):

```
if (get_compat_timespec(&tv[0], &t[0])) { ... } (4)
```

Indeed, these examples do not represent particularly “low-level” kernel code. Similar idioms are presented in manuals and textbooks as the established best practice.

The common challenge in these examples is that the specifications of the called functions do not foresee these particular uses, but are formulated with respect to the passed pointers alone — they are small specifications [6]. To verify the calls, it is necessary to reason about the layout of the data structures and their components, and to derive frame axioms for the remaining data structures.

This paper’s contribution is a method for automatic reasoning about memory layouts in the context of a Hoare logic. We provide a language for expressing layouts and a logic and proof method for refining and re-interpreting layouts. The approach is flexible in that it supports user-defined layout components and user-provided refinements and re-interpretations. This contribution is thus complementary to the work presented in [7], where unfoldings were left as future work. The treatment of layouts is largely independent of the specific Hoare logic used, but solves proof obligations that arise in Hoare logics generally.

While the earlier presentation [8] already showed the framework’s applicability to C language constructs, the developed method is not limited to these examples. To demonstrate how it supports high-level reasoning about manipulations of data structures, we verify the Schorr-Waite graph marking algorithm, which is considered a benchmark for verification methodologies [9]. As a novel aspect, our proof does not assume Burstall’s split heap memory model, as previous studies have done, but directly treats byte-addressed memory. Nevertheless, both the assertions and the proof steps remain very close to the informal understanding of the algorithm, as expressed in pointer diagrams.

The development presented in this paper is mechanized in Isabelle/HOL to ensure its soundness. We also use Isabelle/HOL as an example verification environment, and the presented proof strategies are implemented as ML tactics to establish their utility. From a different point of view, the theorems used during verification can be seen as a first-order axiomatization of the introduced layout constants and operators. In this perspective, HOL serves as a meta-logic in which these theorems are proven (see [10] for a similar discussion).

*Organization of the Paper* Section 2 analyzes the proof obligations about memory layouts arising in Hoare logics and summarizes the concepts from [7]. Section 3 contains our framework for reasoning about memory layouts. Section 4 shows that the framework can solve the introductory examples. Section 5 presents the verification of the Schorr-Waite graph marking algorithm. Section 6 surveys related work. Section 7 concludes.

## 2 Memory-Related Proof Obligations in the Hoare Logic

This section describes the considered programming language and Hoare logic. It summarizes the formalization of memory layouts in lightweight separation [7] and applies them to the memory-related proof obligations of the Hoare logic. The treatment from [7] is adapted to finite address spaces and extended by frames for procedure calls.

### 2.1 Language and Hoare Logic

We consider a C-like language that is inspired by Norrish’s detailed analysis [11]. Its expressions include the usual primitive arithmetic operations, pointer dereferencing, and side-effecting operators, as well as pointer arithmetic and an address operator applicable to arbitrary l-values (i.e. memory objects). As statements, we support if, while, return, blocks with local variable declarations, and the execution of expressions.

We use a standard big-step operational semantics. Compared with [11], we introduce mainly two simplifications with the purpose of focussing on memory-related aspects: first, expressions are executed left-to-right and side-effects are committed to memory immediately. Second, there is no distinction between allocated and initialized memory (cf. [11, Sec. 3.1.2] for both).

The memory model is captured by the following Isabelle/HOL type, where `addr` is a type isomorphic to 32-bit words [12]. (“ $\Rightarrow$ ” denotes total functions.)

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

A memory state’s *domain* and *content* together define a partial function from the allocated addresses to their content. The *history variable* `m-valid` designates whether any illegal accesses have occurred during the execution [13]. The operational semantics accesses memory only through the functions `fetch` and `store`, which transfer byte-representations of values from and to memory. These functions set `m-valid` to `false` if unallocated addresses are manipulated.

```
fetch :: "addr  $\Rightarrow$  nat  $\Rightarrow$  memory  $\Rightarrow$  byte list  $\times$  memory"
store :: "addr  $\Rightarrow$  byte list  $\Rightarrow$  memory  $\Rightarrow$  memory"
```

Execution is defined relative to a *context*, given by the following record type, which contains the definitions of struct types, functions, and local variables. We will subsequently use  $\Gamma$  to name contexts. (“ $\rightarrow$ ” denotes partial functions; `ty` is the datatype representing the language types.)

```
record ctx =
  ctx-structs :: "string  $\rightarrow$  struct-def"
  ctx-prog    :: "string  $\rightarrow$  func"
  ctx-vars    :: "string  $\rightarrow$  addr  $\times$  ty"
```

Note that this memory model does not make a structural distinction between local variables and the heap. In particular, it is possible to apply the address operator and pointer arithmetic for accessing local variables.

We use a Hoare logic for fault-avoiding partial correctness. The rules are forward-style and generalize Floyd's assignment axiom [7]. The treatment of recursive functions and auxiliary variables are based on Schirmer's presentation [14]. Side-effecting expressions are handled using Kowaltowski's approach [15]. For the present purposes, only the memory-related proof obligations are important. They are analyzed subsequently.

## 2.2 Formalizing Layouts

Memory layouts are usually perceived as recursively nested objects (e.g. [13,16]), which suggests a formalization by a grammar (or equivalently an algebraic data type). This approach has, however, the drawback that it fixes the set of possible memory layouts. The examples in Section 1, on the other hand, suggest that different views on a single memory state may be necessary. We therefore use a shallow embedding of memory layouts into HOL, i.e. we define HOL constants and functions that capture the memory region covered by a layout. The central notion is therefore that of a *cover*, which describes a region by comprehension:

```
cover = "addr set ⇒ bool"
```

All covers mentioned subsequently will be *well-formed* in the sense that they accept a single address set or none at all. It is then straightforward to define raw memory regions, and the regions occupied by some typed value, or a variable, and by arrays.<sup>1</sup>

```
block a n      ≡ λS. S = {a.. < a ⊕ n} ∧ a ≤ a ⊕ n
typed-block Γ a t ≡ λS. block a (of-nat (sz-of-ty Γ t)) S ∧ is-small-type Γ t
var-block Γ v   ≡ typed-block Γ (addr-of Γ v) (type-of Γ v)
array Γ t p i j ≡ λS. S = {p ⊕ [Γ,t] i .. < p ⊕ [Γ,t] j} ∧ 0 ≤s i ∧ i ≤s j ∧
                    p ⊕ [Γ,t] i ≤ p ⊕ [Γ,t] j ∧ unat j * (sz-of-ty Γ t) ≤ unat max-word
```

The covers `block a n` and `array Γ t p i j` thus describe continuous regions of addresses. The side-conditions exclude overflows in the address arithmetic. The remaining two constants introduce typed views on blocks. Composite structures are expressed using the following disjointness combinator for covers:

```
A || B ≡ λS. ∃S1 S2. A S1 ∧ B S2 ∧ S = S1 ∪ S2 ∧ S1 ∩ S2 = {}
```

Subsequently, a *layout block* is a cover given by a defined constant, as opposed to being constructed by the disjointness combinator.

As an example, a variable `p` (of type `int*`), and the region it refers to would be formalized as follows (double quotes surround strings; `to_ptr` converts the byte-representation of the pointer into an address; `rdv` reads the byte-representation of the value stored in a variable):

```
var-block Γ "p" || typed-block Γ (to_ptr (rdv Γ "p" M)) int
```

The intention in introducing covers, and in particular cover constants, is to reason symbolically and efficiently to solve the memory-related proof obligations discussed

<sup>1</sup> `sz-of-ty`, `addr-of`, and `type-of` look up information on types and variables; `of-nat` and `unat` convert between `nat` and `word` [12]; `is-small-type` asserts that the size of a type can be represented in 32 bits; `max-word` is the largest 32-bit word.  $\lambda$  denotes a function;  $\{a .. < b\}$  is the notation for the interval  $[a, b)$ ; relations  $\leq_s$  and  $<_s$  are signed comparison on words [12],  $\oplus$  is raw address arithmetic, pointer arithmetic  $\oplus[\Gamma, t]$  uses a type and the context.

subsequently. To make this reasoning uniform, we introduce the following *subcover* relation, which expresses that a region of cover A is part of the region of cover B.

$$A \preceq B \equiv \forall S. B S \longrightarrow (\exists S'. S' \subseteq S \wedge A S')$$

In the arising proof obligations, which will be examined subsequently, cover B will be given in the specification of algorithms and describe the overall memory layout. Cover A will describe the memory accessed by the program or an assertion at a particular point. Both A and B in general consist of several layout blocks, combined by the disjointness operator. The core of the symbolic computation, as formulated in [7], then consists in suitably rearranging A and B by associativity and commutativity of that operator, and applying one of the rules (5) to finish the proof.

$$C \preceq C \parallel D \quad D \preceq C \parallel D \quad (5)$$

Already this simple form of reasoning yields concise proofs in many cases, e.g. for the list reversal example [7]. This paper extends the computation to take into account structural unfoldings of the blocks in B, which makes the verification approach viable both for low-level programs (Section 4) and reasoning about data structures (Section 5).

### 2.3 Normal Form of Assertions

The essence of lightweight separation [7] is that the user simply specifies the memory layout in addition to a first-order (or higher-order) assertion about the memory content. The memory layout is captured by covers, using the following *covered* predicate (read “M is covered by A”):

$$M \blacktriangleright A \equiv \text{m-valid } M \wedge A (\text{m-dom } M)$$

For an assertion P about the content, the normal form of assertions is therefore:

$$\lambda \Gamma M. \exists x_1 \dots x_n. M \blacktriangleright A \wedge P \Gamma M x_1 \dots x_n \quad (6)$$

The variables  $x_1 \dots x_n$  name intermediate results encountered during expression evaluation as usual in forward-style Hoare logics. In post-conditions of expressions, the result  $v$  would be an additional parameter [15]. In a first-order setting,  $\Gamma$ ,  $M$ , and possibly  $v$  would be allowed to occur free in A and P.

### 2.4 Proof Obligations on Allocatedness

Since the Hoare logic is fault-avoiding, any memory access generates the proof obligation that the region is allocated. This condition is captured by the *allocated* predicate (read “A is allocated in M”, where A is a cover and M a memory state):

$$M \triangleright A \equiv \text{m-valid } M \wedge (\exists S. S \subseteq \text{m-dom } M \wedge A S)$$

In the rule for dereference expressions  $*e$ , for instance, let P be the post-condition of the evaluation of  $e$ . Following [15], it is a predicate on the current context  $\Gamma$ , the memory state  $M$  after the possibly side-effecting execution of  $e$ , and the computed result  $v$ . The necessary proof obligation is:

$$\forall \Gamma M v. P \Gamma M v \longrightarrow M \triangleright \text{typed-block } \Gamma (\text{to-ptr } v) t \quad (7)$$

By construction of the Hoare rules,  $P$  will be in normal form and the allocatedness can be determined from the layout given in  $P$ . The following theorem is then used to reduce (7) to a proof about layouts alone.

$$\frac{M \blacktriangleright A \quad B \preceq A}{M \triangleright B} \quad (8)$$

## 2.5 Side-conditions in Memory Layouts

Memory layouts usually include tacit assumptions about the involved addresses. The C standard prescribes, for instance, that an allocated block consists of a sequence of increasing addresses, which motivates the side-condition excluding overflows in the address arithmetic in the definitions of `block` and `array` (Section 2.2). For the purposes of verification, these assumptions constitute invariants. Including them into the definitions of cover constants facilitates automatic reasoning, since theorems about the constants have fewer premises. The definition of a cover constant  $c$  with parameters  $x_1 \dots x_n$  therefore usually has the following form (where the region  $S$  does not occur in  $P$ ).

$$c x_1 \dots x_n \equiv \lambda S. S = \dots \wedge P x_1 \dots x_n$$

To express that the side-conditions hold, the covered region itself is immaterial. We say that a cover is *valid* if it covers some memory region; from the validity, it can be deduced that the side-conditions are satisfied.

$$\text{is-valid } A \equiv \exists S. A \ S$$

It is obviously possible to derive validity from a given memory layout by means of the subcover relation; furthermore, the subcover relation itself preserves validity.

$$\frac{M \blacktriangleright A \quad B \preceq A}{\text{is-valid } B} \quad \frac{\text{is-valid } A \quad B \preceq A}{\text{is-valid } B} \quad (9)$$

With the interpretation of side-conditions as invariants, an `is-valid` statement should be read as “the side-conditions hold”. In Section 3, this reading explains how side-conditions are maintained through unfoldings.

## 2.6 Side-effects, Disjointness, and Aliasing

Side-effects in the semantics are formalized in Hoare rules by syntactic manipulations of assertions. Suppose, for instance, that a command  $c$  performs some side-effect  $f$  on the memory, i.e. for the pre-state  $M$ , the post-state is  $f M$ . In a higher-order setting, the Hoare rule can be given directly (cf. [14, Figure 3.1]):

$$\vdash \{ \lambda \Gamma M. Q \Gamma (f M) \} c \{ Q \}$$

In a first-order verification environment, the  $\beta$ -reduction is replaced by a syntactic substitution, as in Hoare’s assignment axiom.

We prefer to use a forward-style Hoare logic in order to emulate the reasoning possible in separation logic [6, 17, 10]. Here, Floyd’s assignment axiom can be generalized by introducing *inverse operators* [7]. An inverse operator for operator  $f$  is a function  $F$  such that  $F M (f M) = M$ , i.e.  $F$  undoes the effect of  $f$  by replacing the region modified

by  $f$  by the content from its first argument. The forward-style Hoare rule for command  $c$  is then obtained by existentially quantifying over the previous memory state  $M'$ :

$$\vdash \{ P \} c \{ \lambda \Gamma M. \exists M'. P \Gamma (F M' M) \wedge Q \Gamma M \}$$

The assertion  $Q$  expresses the result of  $f$ , such as a particular region now containing a particular value. Again, in a first-order system, the verification condition generator would apply a syntactic substitution.

In both cases, therefore, the “current” memory state  $M$  in some assertion  $P$  is replaced by a modified state  $(g M)$ , where  $g$  is either the effect itself or its inverse operator. In both cases, the goal must be to remove the operator  $g$  in order to retrieve an assertion about the “current” state  $M$  itself.

To illustrate the point, suppose  $f$  is the operator  $\text{store } a \ v \ M$  which writes the byte-representation of value  $v$  at address  $a$  in  $M$ . Suppose then that pre-condition  $P$  contains the assertion  $\text{to-int}(\text{rdv } \Gamma \ "x" \ M) > 0$ , which reads the content of variable “ $x$ ” and interprets the byte-representation as an integer. The post-condition would be  $\text{to-int}(\text{rdv } \Gamma \ "x" \ (\text{STORE } a \ (\text{length } v) \ M' \ M)) > 0$  instead. Intuitively, this assertion reads: “*before* the side-effect occurred,  $x$  contained a positive value”.

We thus wish to simplify terms of the form  $\text{mac}(\text{mop } M)$ , where  $\text{mac}$  is a *memory accessor* like  $\text{rdv } \Gamma \ x$  and  $\text{mop}$  is a *memory operator* like  $\text{STORE } a \ (\text{length } v) \ M'$ . We capture the behaviour of memory accessors and operators abstractly by the following constants (where the predicates  $\text{eqv-inside}$  and  $\text{eqv-outside}$  assert that their memory arguments are the same inside and outside, respectively, a region covered by  $A$ ).

$$\begin{aligned} \text{accesses } \text{mac } M \ A &\equiv \forall M'. \text{eqv-inside } A \ M \ M' \longrightarrow \text{mac } M = \text{mac } M' \\ \text{modifies } \text{mop } M \ A &\equiv \text{eqv-outside } A \ M \ (\text{mop } M) \end{aligned}$$

Theorem (10) then allows the desired simplification to take place. Premises 1 and 2 are properties of the involved accessor and modifier constants. Premises 3 and 4 are solved automatically, since all covers we use are well-formed.

$$\frac{\begin{array}{l} \text{is-valid } B \longrightarrow \text{accesses } \text{mac } M \ B \\ \text{is-valid } A \longrightarrow \text{modifies } \text{mop } M \ A \\ \text{wf-cover } A \quad \text{wf-cover } B \\ M \blacktriangleright C \quad A \parallel B \preceq C \end{array}}{\text{mac}(\text{mop } M) = \text{mac } M} \quad (10)$$

In a first-order setting, the verification condition generator would use (10) to pre-generate rewrite rules by solving the premises 1–4 with provided theorems.

## 2.7 Function Calls

The specification of functions introduces the frame inference problem (e.g. [10]): it must be possible to infer from the specification which parts of the memory remain unmodified. As the examples in Section 1 show, no reasonable restriction can be placed on the memory objects that can be passed by reference. Our solution is to introduce a constant  $\text{frame}$  which asserts that a memory region given by a cover  $R$  has not been modified between the states  $M$  and  $M'$ .

$$\text{frame } R \ M \ M' \equiv \text{wf-cover } R \wedge \text{eqv-inside } R \ M \ M'$$

A function specification then consists of the pre- and post-conditions of the form

$$\begin{aligned} M \blacktriangleright A \parallel R \wedge \text{frame } R \ M_0 \ M \ \wedge \ P \\ M \blacktriangleright B \parallel R \wedge \text{frame } R \ M_0 \ M \ \wedge \ Q \end{aligned}$$

where  $A$  and  $B$  capture the memory parts directly manipulated by the function and  $R$  and  $M0$  are auxiliary (or logical) variables [14, §3.1.1].

The pre-condition of the function call must then imply the function's pre-condition. For a pre-condition  $\exists x_1 \dots x_n. M \triangleright C \wedge P'$ , the proof obligation becomes:

$$\forall x_1 \dots x_n. M \triangleright C \wedge P' \longrightarrow M \triangleright A \parallel ?R \wedge \text{frame } ?R \text{ ?M0 } M \wedge P$$

Here the auxiliary variables have become unknowns  $?R$  and  $?M0$  that can be instantiated (cf. [14, Sec. 3.1.1]). The reasoning task is expressed by theorem (11): we need to rewrite the cover  $C$  suitably, assuming that its invariants hold.

$$\frac{M \triangleright C \quad \text{is-valid } C \longrightarrow C = A}{M \triangleright A} \quad (11)$$

### 3 A Framework for Reasoning about Memory Layouts

Section 2 has identified the memory-related proof obligations arising in Hoare logics and has reduced them to subcover and equality relations between (first-order) covers. However, the challenges from Section 1 remain: in each of these examples, the manipulated memory parts are not directly given in the specified layout; instead, the layout must first be refined and re-interpreted. This section presents a matching framework for general *unfoldings* of memory layouts.

#### 3.1 Unfoldings of Layouts

In the following examples, we use the syntax supported by our Isabelle/HOL implementation: a variable block for  $x$  is written  $\langle\langle x \rangle\rangle$  and a typed block at  $p$  with type  $t$  is rendered as  $\langle\langle p : t \rangle\rangle$ . Context arguments, `rdv`, `to_ptr`, etc. are inserted by translation functions [18, Sec. 8.6].

In the example (2), then, the local variable `inode` points to a record containing a mutex object, while the called function expects the mutex object alone. The required equality in (11) becomes:

$$\langle\langle \text{inode} \rangle\rangle \parallel \langle\langle \text{inode} : \text{struct inode} \rangle\rangle = \langle\langle \&\text{inode} \rightarrow \text{i\_mutex} : \text{struct mutex} \rangle\rangle \parallel ?R$$

The task is to unfold  $\langle\langle \text{inode} : \text{struct inode} \rangle\rangle$  into the constituent fields to expose the mutex. The examples (3) and (4) do, indeed, not introduce any new complications, because the Hoare logic and memory layouts treat heap- and stack-allocated memory objects in the same way. The required equality for (3) is

$$\langle\langle t \rangle\rangle = \langle\langle \&t.\text{timer} : \text{struct hrtimer} \rangle\rangle \parallel ?R$$

In the final example (4), the only difference is that the variable `tv` contains an array rather than a struct, which results in the proof obligation:

$$\langle\langle \text{tv} \rangle\rangle = \langle\langle \&\text{tv}[0] : \text{struct timespec} \rangle\rangle \parallel ?R$$

The automated reasoning support must thus be able to re-write memory layouts on the fly. The unfolding rules used in our framework are of the form:

$$\frac{P_1 \dots P_n}{\text{is-valid } A \longrightarrow A = B_1 \parallel \dots \parallel B_m} \quad (12)$$

If the premises  $P_1 \dots P_n$  hold, the layout  $A$  can be refined into layout  $B_1 \parallel \dots \parallel B_m$ . In the case  $m = 1$ , the unfolding is, in fact, a re-interpretation of layout block  $A$ . Note

also how the side-conditions associated with  $A$  (Section 2.5) are available during the unfolding — the framework will apply the rule only in corresponding situations. The invariants associated with  $A$  therefore need not be repeated in the premises  $P_1 \dots P_n$ . If a theorem does not depend on the validity of  $A$ , it can omit the implication in the conclusion.

As a first example, variables can be re-interpreted as typed blocks by (13). When the program takes the address of a variable, the automated reasoning will apply the theorem correspondingly.

$$\frac{a = \text{addr-of } \Gamma v \quad t = \text{type-of } \Gamma v}{\text{var-block } \Gamma v = \text{typed-block } \Gamma a t} \quad (13)$$

By (14), unfolding rules can be used to prove subcover relations as needed for allocation (8) and disjointness (10) proofs.

$$\frac{\text{is-valid } A \longrightarrow A = B}{B \preceq A} \quad (14)$$

By reflexivity of the equality and subcover relations and the following theorems, unfoldings can be applied anywhere within a nested layout:

$$\frac{\frac{\text{is-valid } A \longrightarrow A = A' \quad \text{is-valid } B \longrightarrow B = B'}{\text{is-valid } (A \parallel B) \longrightarrow A \parallel B = A' \parallel B'}}{\frac{A \preceq A' \quad B \preceq B'}{A \parallel B \preceq A' \parallel B'}} \quad (15)$$

In principle, the examples from Section 1 can be solved using the above theorems and the unfolding rules for special data structures from Section 4. For restricted assertion languages, a brute-force unfolding with all possible rules is viable [10]. For first-order (or higher-order) assertions, we develop a two-step proof search. For a given proof obligation  $B \preceq A$  or  $\text{is-valid } A \longrightarrow A = B$ , it first *locates* all elementary blocks from  $B$  in  $A$  and then *unfolds*  $A$  to expose the blocks  $B$ , using the information gathered in the first step during the second.

### 3.2 Locating Memory Blocks

The proof obligations to be treated have the form  $B_1 \parallel \dots \parallel B_m \preceq A_1 \parallel \dots \parallel A_n$  or  $\text{is-valid}(A_1 \parallel \dots \parallel A_n) \longrightarrow A_1 \parallel \dots \parallel A_n = B_1 \parallel \dots \parallel B_m$ . For the proof search, it is useful to consider the blocks  $\{B_1 \dots B_m\}$  as a multiset by associativity and commutativity of disjointness. By abuse of notation, we will therefore write  $\{B_1 \dots B_m\} \preceq A$  for layout blocks  $B_1 \dots B_m$  and cover  $A$ . The location phase enriches this notation further with a justification of the subcover relation. In our Isabelle/HOL implementation of the proof search, this information is kept in ML data structures; for using a first-order prover, it can be encoded in terms.

The justification for a subcover relation  $B_k \preceq A$  (for  $k \in [1, m]$ ) consists in unfolding  $A$  in particular positions. Positions are denoted by *paths* in  $(L|R)^*$  (for “left” and “right”; the empty path is  $\varepsilon$ , concatenation is written  $p \cdot q$ ; the sub-layout of  $A$  at  $p$  is  $A \downarrow_p$ ). Since unfoldings can occur recursively, the enriched notation is:

$$\{(B_k, p_{k0}, ((u_{k1}, p_{k1}) \dots (u_{km_k}, p_{km_k})))\}_{k=1}^m \preceq A \quad (16)$$

The reading, to be defined formally in Section 3.3, is: for each  $k$ , the block  $\mathbf{B}_k$  is found inside  $A$  by first following path  $p_{k0}$ , then applying unfolding rule  $u_{k1}$  (using (14)), then following  $p_{k1}$ , and proceeding in this manner until all unfoldings and paths have been exhausted. Note that for each application of an unfolding rule  $u_{kj}$ , its premises need to be proven (see (12)).

We now show that justifications can be computed efficiently. The central idea is to prepare in advance a set  $S$  for subcover justifications of all possible combinations of a given set of unfolding rules. The process starts with the trivial justification  $\{(A, \varepsilon, \varepsilon)\} \preceq A$ , which captures the reflexivity of the subcover relation. Then, for any  $\{(C, \varepsilon, J)\} \preceq D \in S$  and unfolding step  $u$  of the form (12), where  $\sigma C = \sigma A$  for some  $\sigma$ , i.e.  $C$  unifies with  $A$ , we insert for  $k = 1 \dots m$  into  $S$  a new justification (17).

$$\{(\sigma \mathbf{B}_k, \varepsilon, (\sigma J, (\sigma u, p_k)))\} \preceq \sigma D \quad (17)$$

The application of substitution  $\sigma$  here is defined by its application to all occurring terms and theorems. The path  $p_k$  is the path to  $\mathbf{B}_k$  on the right-hand side in (12). Note that these justifications are sound by lemmas (5).

If this saturation process terminates with a set  $S$ , then by construction  $B \preceq A$  can be proven by the given unfolding rules iff there is some path  $p_0$ , substitution  $\sigma$ , and justification  $\{(B', \varepsilon, J)\} \preceq A' \in S$  such that  $B = \sigma B'$  and  $A \downarrow_{p_0} = \sigma A'$ . The resulting justification is then  $\{(B, p_0, \sigma J)\} \preceq A$ .

To solve the proof obligations given at the beginning of this section efficiently, just apply this step to all pairs  $\mathbf{A}_i$  and  $\mathbf{B}_j$ . A term index is used to identify possible justifications. The desired result (16) is found.

*Remark 1 (Termination)* The generation phase might fail to terminate. This is, in particular, the case for recursive, linked data structures such as lists or trees, whose structural unfoldings can be applied several times in a row. The termination argument for the only approach handling such unfoldings [10] relies on a restricted form of assertions that is not sufficient for full functional verification. We therefore leave a general solution as future work.

In the meantime, the following approach has led to concise proofs. The saturation process is modified to never apply the same unfolding twice in a row, which enforces termination and is sufficient for non-recursive data types. For inductive data structures, the user introduces specific pointers, which resemble *iterators* [19], that designate places in the data structure. For instance, in [20] we define inductively a predicate  $\text{iter } \Gamma \mathbf{a} \mathbf{p} \mathbf{q} \mathbf{M}$  which expresses that  $\mathbf{a}$  is a node in the list fragment  $\mathbf{p} \dots \mathbf{q}$ . When the program reads a value through  $\mathbf{a}$ , the node  $\mathbf{a}$  is exposed automatically by the following unfolding rule (which saturation combines with an unfolding of the first list node):

$$\frac{\text{iter } \Gamma \mathbf{a} \mathbf{p} \mathbf{q} \mathbf{M}}{\text{list } \Gamma \mathbf{p} \mathbf{q} \mathbf{M} = \text{list } \Gamma \mathbf{p} \mathbf{a} \mathbf{M} \parallel \text{list } \Gamma \mathbf{a} \mathbf{q} \mathbf{M}}$$

We have found that the *iter* predicate not only enables these unfoldings, but beyond the use in this auxiliary capacity serves well to express specifications and invariants succinctly and close to the programmer's intention.

*Remark 2 (Multiple Unfoldings)* Low-level programs usually work with data in memory at different levels of abstraction, e.g. by copying the byte-representation of some data value. The unfoldings given to the framework will therefore contain several unfoldings for the same cover, one for each such view. In the location phase, a block  $\mathbf{B}_j$  might

then be located in  $A_i$  by several justifications. (However, by definition of disjointness, it cannot be located in a different  $A_{i'}$ .) In the subsequent presentation, we assume that a single justification has been computed; if ambiguities arise, they are resolved by iteration through the possible solutions. With respect to efficiency, we have found that most unfoldings can be discarded immediately and quickly by trying to prove their equality premises.

### 3.3 Computing Unfoldings

The result of the location phase is of the form  $\{(\mathbf{B}_k, p_{0k}, J_k)\}_{k=1}^m \preceq A$ : it captures precisely where the  $\mathbf{B}_1 \dots \mathbf{B}_m$  are located inside layout  $A$ . To compute the actual unfolding equality, it is sufficient to follow the justifications  $\{(p_{0k}, J_k)\}_{k=1}^m$  in a recursive process. We give the process in the form of inference rules for a judgement

$$\{(\mathbf{B}_k, p_{0k}, J_k) | P k\} \rightsquigarrow \text{is-valid } A \longrightarrow A = B$$

where  $P$  selects a subset of  $[1, m)$  and the result  $\text{is-valid } A \longrightarrow A = B$  is a theorem. The recursion base consists in a single  $\mathbf{B}_k$  being actually found:

$$\{(\mathbf{B}_k, \varepsilon, \varepsilon)\} \rightsquigarrow \text{is-valid } \mathbf{B}_k \longrightarrow \mathbf{B}_k = \mathbf{B}_k \quad (18)$$

The first recursion step computes the unfoldings of two disjoint sub-multisets of the  $\mathbf{B}_k$  using theorem (15).

$$\frac{\begin{array}{l} \{(\mathbf{B}_k, p'_{0k}, J_k) | P k \wedge p_{0k} = L \cdot p'_{0k}\} \rightsquigarrow \text{is-valid } A \longrightarrow A = B \\ \{(\mathbf{B}_k, p'_{0k}, J_k) | P k \wedge p_{0k} = R \cdot p'_{0k}\} \rightsquigarrow \text{is-valid } A' \longrightarrow A' = B' \\ \forall k. P k \longrightarrow p_{0k} \neq \varepsilon \end{array}}{\{(\mathbf{B}_k, p_{0k}, J_k) | P k\} \rightsquigarrow \text{is-valid } (A \parallel A') \longrightarrow A \parallel A' = B \parallel B'} \quad (19)$$

Finally, if the paths in the justification have been exhausted, an unfolding takes place. We denote by  $u(A)$  the application of the unfolding rule of the form (12) to a term  $A$ , which yields the unfolded layout. At this point the premises of step  $u$  need to be proven.<sup>2</sup> Note how the local paths  $p_k$  become the new paths in the justification.

$$\frac{\{(\mathbf{B}_k, p_k, J_k) | P k\} \rightsquigarrow \text{is-valid } u(A) \longrightarrow u(A) = B}{\{(\mathbf{B}_k, \varepsilon, (u, p_k) \cdot J_k) | P k\} \rightsquigarrow \text{is-valid } A \longrightarrow A = B} \quad (20)$$

Since (19) and (20) apply only to judgments of specific forms, the completeness of this procedure needs discussion. Consider a justification  $\{(\mathbf{B}_k, p_k, J_k) | P\} \preceq A$  where  $A$  is a layout block and the  $\mathbf{B}_k$  can be proven disjoint by the given unfolding rules. If the multiset is a singleton, rule (18) applies. Next, since the blocks  $\mathbf{B}_k$  can be proven disjoint, either all  $p_k$  must be non-empty, in which case (19) applies, or they are all empty. (Otherwise, there would be some  $k$  and  $k'$  with  $\mathbf{B}_{k'} \preceq \mathbf{B}_k$ , contradicting disjointness.) In this latter case, (20) applies with some common unfolding rule  $u$ , since we can assume that the justifications have been selected correspondingly by Remark 2.

The process of unfolding a memory layout is thus made deterministic by employing the information gathered in the location phase.

<sup>2</sup> Note that the same premises have been proven during the location phase. In the ML implementation, we keep the proven premises as Isabelle theorems.

### 3.4 Unfolding On-Demand

The above presentation of the proof search assumes that unfolding rules have the static form (12). This is, however, too restrictive in general: locating a set of array elements and array slices in a given array would require “guessing” a suitable split of the index range in advance (see Section 4.2). The problem is solved by lazy computation of unfoldings. The unfolding rules are used only in two places: to generate the subcover theorems for the location phase by (14) and to compute the unfolding in (20). The first use can be removed if the subcover theorems are given directly. When the unfolding rule is required in (20), more information is available in the form of the proven premises of (12). We therefore apply a standard strategy (e.g. [18, Sec. 10.2.5]): when (20) is applied, an ML function is called with the current judgment. The function returns both the unfolding rule  $u$  and the paths  $p_k$  for each of the blocks in the left-hand side. Unfoldings in this way are computed lazily, on-demand.

## 4 Application to Language Constructs

The framework from Section 3 supports automatic reasoning about a wide range of unfoldings of memory layouts. In this section, we apply it to examples derived from Section 1 by giving specific unfolding rules for C language constructs. In the next Section 5, we will show that the same mechanisms also apply to the verification of algorithms manipulating data structures.

### 4.1 Struct Types

The layout of a struct type is given with its type definition. It consists of the struct’s fields, possibly separated by padding to ensure alignment of the fields’ data. Since it can be added straightforwardly, we omit padding for the time being. Using auxiliary constants `field-off` and `field-ty`, which determine the offset and type of a field in a given struct type, we can define a new cover for a single field:

$$\text{field-block } \Gamma \text{ p t f} \equiv \text{typed-block } \Gamma (\text{p} \oplus (\text{of-nat } (\text{field-off } \Gamma \text{ t f}))) (\text{field-ty } \Gamma \text{ t f})$$

For a list of fields, their joined layout is then given by a recursive function:

$$\begin{aligned} \text{fields-cover } \Gamma \text{ t a } [] &= \text{Empty} \\ \text{fields-cover } \Gamma \text{ t a } ((\text{f,ty}) \# \text{fs}) &= \text{field-block } \Gamma \text{ a t f} \parallel \text{fields-cover } \Gamma \text{ t a fs} \end{aligned}$$

With these preliminary definitions, we can prove a general unfolding theorem for struct types. Its premise captures that type  $t$  is a defined struct in the context  $\Gamma$ .

$$\frac{\text{wf-struct } \Gamma \text{ t}}{\text{typed-block } \Gamma \text{ a t} = \text{fields-cover } \Gamma \text{ t a } (\text{struct-fields } \Gamma \text{ t})} \quad (21)$$

For a specific struct type such as `struct point { int x; int y; }`, the special rule (22) can be generated from (21) directly. Here `point-known`  $\Gamma$  captures, again, that the definition of `struct point` is present in context  $\Gamma$ .

$$\frac{\text{point-known } \Gamma}{\text{typed-block } \Gamma \text{ a } (\text{struct point}) = \text{field-block } \Gamma \text{ a } (\text{struct point}) \text{ "x" } \parallel \text{field-block } \Gamma \text{ a } (\text{struct point}) \text{ "y" }} \quad (22)$$

The re-interpretation of a local variable of struct type as a memory object is given by the unfolding step (13). Together with (22), the challenges of the introductory examples can be solved. Suppose, for instance, a function `void set(int *p, int i)` sets `*p` to `i`. Its pre-condition requires a layout  $M \blacktriangleright \langle\langle s \rangle\rangle \parallel \langle\langle i \rangle\rangle \wedge \text{point-known } \Gamma \wedge P \langle\langle s.x \rangle\rangle i$ . Its post-condition asserts that `*p = i` and `R` is framed, i.e. not modified. We can then verify the following triple automatically (where `struct point s`; and `int i`; are local variables;  $\langle\langle s.x \rangle\rangle$  is expanded to reading field `x` from struct variable `s`).

$$\begin{aligned} & \vdash \{ M \blacktriangleright \langle\langle s \rangle\rangle \parallel \langle\langle i \rangle\rangle \wedge \text{point-known } \Gamma \wedge P \langle\langle s.x \rangle\rangle i \} \\ & \quad \text{set}(\&s.y, 1); \\ & \{ M \blacktriangleright \langle\langle s \rangle\rangle \parallel \langle\langle i \rangle\rangle \wedge \text{point-known } \Gamma \wedge P \langle\langle s.x \rangle\rangle i \wedge \langle\langle s.y \rangle\rangle = 1 \} \end{aligned}$$

For the function call, the given layout is unfolded to reveal the field-block for `s.y` and to show that the remainder `R` of the memory consists of the field-block for `s.x` and the local variable `i`. Note that the higher-order predicate `P` represents an arbitrary further assertion about `s.x` and `i`, independently of its actual structure.

This example covers usage (3) directly. It also shows the simpler usage (2) to be supported, where the variable-block re-interpretation (13) can be omitted.

## 4.2 Arrays

The unfolding of arrays poses the problem that the necessary refinement of the layout cannot be determined in advance, because it depends on the actual slices and elements that need to be revealed. Unfoldings are therefore computed lazily (Section 3.4) and explicit subcover theorems are used in the location phase (see Remark 1 for termination): (23) and (24) allow the prover to locate slices and elements, and (28) re-interprets an array element as a typed block, thus enabling access by pointer arithmetic.

$$\frac{i \leq_s i' \quad i' <_s j' \quad j' \leq_s j}{\text{array } \Gamma \text{ t a } i' j' \preceq \text{array } \Gamma \text{ t a } i j} \quad (23)$$

$$\frac{i \leq_s j \quad j <_s k}{\text{array-elem } \Gamma \text{ t a } j \preceq \text{array } \Gamma \text{ t a } i k} \quad (24)$$

$$\frac{p = a \oplus [\Gamma, t] j}{\text{array-elem } \Gamma \text{ t a } j = \text{typed-block } \Gamma \text{ p t}} \quad (25)$$

With these rules, the location phase can associate layout blocks with a given array, proving the premises in the process. When the actual unfolding is required, an `ML` function is called to compute an unfolding rule tailored to the situation. Towards that end, it determines the relative order of the indices from the proven premises and applies Theorems (26) and (27).

$$\frac{i \leq_s j \quad j \leq_s k}{\text{array } \Gamma \text{ t a } i k = \text{array } \Gamma \text{ t a } i j \parallel \text{array } \Gamma \text{ t a } j k} \quad (26)$$

$$\text{is-valid}(\text{array } \Gamma \text{ t a } j (j+1)) \longrightarrow \text{array } \Gamma \text{ t a } j (j+1) = \text{array-elem } \Gamma \text{ t a } j \quad (27)$$

Theorem (27) is of particular interest, because it crucially uses the invariants associated with the single-element array. An array element is simply a typed-block (25), which does not contain all invariants of arrays (Section 2.2).

Arrays in local variables, which have type  $\text{Array } t \ n$  with constant size  $n$  and element type  $t$ , are found by the following rule (28), which is applied after (13).

$$\text{is-valid (typed-block } \Gamma a (\text{Array } t \ n)) \longrightarrow \text{typed-block } \Gamma a (\text{Array } t \ n) = \text{array } \Gamma t a \ 0 \ n \quad (28)$$

Again, this unfolding relies on the validity of the typed block to establish the side-conditions of the `array` from the well-formedness of the `Array` type.

With these unfoldings, the last introductory challenge (4) can be resolved. Here, a local variable `int a[16]`; is allocated and some assertion about its first  $i$  elements is given. It can then be proven automatically that setting `a[i]` does not influence that assertion, as stated in the following triple.

$$\begin{aligned} &\vdash \{ M \blacktriangleright \langle i \rangle \parallel \langle a \rangle \wedge 0 \leq_s i \wedge i <_s 16 \wedge (\forall k. 0 \leq_s k \wedge k <_s i \longrightarrow \langle a[k] \rangle = 0) \} \\ &\quad a[i] = 1; \\ &\{ M \blacktriangleright \langle i \rangle \parallel \langle a \rangle \wedge 0 \leq_s i \wedge i <_s 16 \wedge (\forall k. 0 \leq_s k \wedge k <_s i \longrightarrow \langle a[k] \rangle = 0) \} \end{aligned}$$

This example also shows that our method goes beyond automated fragments of separation logic [10, 21] in handling local assumptions on quantified variables.

A final example demonstrates the flexibility of the presented framework. Suppose the function `void memcpy(char *src, char *dst, int n)` copies memory byte-wise. Its pre-condition demands that the source and destination arrays are allocated:

$$M \blacktriangleright \text{array } \Gamma \langle \text{char} \rangle \text{ src } 0 \ n \parallel \text{array } \Gamma \langle \text{char} \rangle \text{ dst } 0 \ n \parallel R$$

Its post-condition asserts the same layout, that the elements of `src` have been copied to `dst`, and that only the `dst` array has been modified, such that any assertions about `src` and  $R$  continue to hold.

$$\begin{aligned} M \blacktriangleright &\text{array } \Gamma \langle \text{char} \rangle \text{ src } 0 \ n \parallel \text{array } \Gamma \langle \text{char} \rangle \text{ dst } 0 \ n \parallel R \wedge \\ &\text{array-elems } \Gamma \langle \text{char} \rangle \text{ src } 0 \ n \ M = \text{array-elems } \Gamma \langle \text{char} \rangle \text{ dst } 0 \ n \ M \wedge \\ &\text{frame } (R \parallel \text{array } \Gamma \langle \text{char} \rangle \text{ src } 0 \ n) \ M \ 0 \ M \end{aligned}$$

Then, we can implement a low-level string concatenation. We prove the following triple in a variable context `int m; int n; char *a; char *b; char *dst;`.

$$\begin{aligned} &\vdash \{ M \blacktriangleright \langle a \rangle \parallel \langle n \rangle \parallel \langle b \rangle \parallel \langle m \rangle \parallel \langle \text{dst} \rangle \parallel \\ &\quad \text{array } \Gamma \langle \text{char} \rangle a \ 0 \ n \parallel \text{array } \Gamma \langle \text{char} \rangle b \ 0 \ m \parallel \text{array } \Gamma \langle \text{char} \rangle \text{dst } 0 \ (n + m) \wedge \\ &\quad \text{array-elems } \Gamma \langle \text{char} \rangle a \ 0 \ n \ M = A \wedge \text{array-elems } \Gamma \langle \text{char} \rangle b \ 0 \ m \ M = B \} \\ &\quad \text{memcpy}(a, \text{dst}, n); \\ &\quad \text{memcpy}(b, \text{dst} + n, m); \\ &\{ \text{array-elems } \Gamma \langle \text{char} \rangle \text{dst } 0 \ (n + m) \ M = (A \ @ \ B) \} \end{aligned}$$

In the second call, the pre-condition of `memcpy` mentions an array starting at index 0, while the actual argument is a slice starting at  $n$ . The re-interpretation (29) shifts a given array slice to begin at index 0.

$$\frac{b = a \oplus [\Gamma, t] \ i \quad m = n - i}{\text{is-valid } (\text{array } \Gamma t \ a \ i \ n) \longrightarrow \text{array } \Gamma t \ a \ i \ n = \text{array } \Gamma t \ b \ 0 \ m} \quad (29)$$

## 5 Verification of the Schorr-Waite Algorithm

The framework introduced in Section 3 is motivated by the desire to support reasoning about the C language constructs: for defined record types and arrays, the obvious deductions about their content should be performed automatically in the presence of pointer-based memory modifications.

This section shows that the developed framework also supports higher-level reasoning heap data structures. Standard algorithms on singly linked lists have already been used for purposes of illustration in previous work [7,20]. We now verify the Schorr-Waite graph marking algorithm, which has been identified as a typical and non-trivial example [9] and has served since as a benchmark in several studies [22–26].

The proof is presented in two steps. Section 5.1 develops a theory of graphs of memory objects, with objects as nodes and references as edges. It is independent of the particular algorithm to be verified. The link to the framework is given by lazy unfolding procedures (Sections 5.1.2, 5.1.3) that prove disjointness of memory regions within object graphs. Section 5.2 then verifies the Schorr-Waite algorithm based on the theory of object graphs. The exposition highlights the close connection between an informal understanding based on pointer diagrams and the formal proofs, thus demonstrating a benefit of a forward reasoning style claimed in [7].

The generic theory of object graphs is developed using Isabelle’s *locales* [27,28]. They enable abstraction over constants under assumptions about these constants, such that the resulting theories can be instantiated in different specific situations, provided that the introduced assumptions can be proven.

## 5.1 Formalizing Object Graphs

Garbage collection algorithms have long been a focus of verification efforts [29–33, 9,22]. Their critical role in ensuring a system’s proper behaviour and the intricate and often low-level details of the algorithms motivate the desire for machine-assisted proofs. The Schorr-Waite algorithm can be used to implement the marking phase in a mark-and-sweep collector in this context.

A characteristic feature of garbage collection algorithms is that they deal with unstructured graphs of memory objects. Starting from a given root set of pointers, they repeatedly traverse any pointers stored in memory objects until all objects reachable from the root set have been identified. In this traversal, the application-specific data structures built from these memory objects are ignored — the algorithms only deal with the definition of the objects themselves.

We are now going to formalize graphs of objects in such a way that the framework developed in Section 3 can be used to automate reasoning about algorithms manipulating the graphs. Section 5.1.1 gives the basic definition of the node set of the graph, on which disjointness can be proven by two lazy unfolding mechanisms presented in Sections 5.1.2 and 5.1.3. Section 5.1.4 finally defines reachability in object graphs.

### 5.1.1 Node Sets

The basis for capturing object graphs consists in capturing the memory area occupied by a set of disjoint objects, which will later form the graph’s nodes. We therefore introduce a cover  $\text{nodes } \Gamma \text{ N}$  for a set of objects given by the pointers  $\text{N}$  to their start addresses. To keep the development independent of a specific type of objects, we assume that a constant  $\text{node}$  yielding the cover for a single object in the graph is given, where the  $\text{node}$  cover must be well-formed in the sense of Section 2.2.

$\text{node} :: \text{"ctx} \Rightarrow \text{addr} \Rightarrow \text{cover}"$

The definition of the cover  $\text{nodes}$  is then straightforward and directly captures the intention: any two objects in the set  $\text{N}$  are disjoint and the entire set occupies the union

of the single contained objects. Furthermore, the single objects are defined consistently, as indicated by the *is-valid* clause.

$$\begin{aligned} \text{nodes } \Gamma \text{ N S} \equiv & (\forall a \in \text{N}. \forall b \in \text{N}. a \neq b \longrightarrow \text{is-valid}(\text{node } \Gamma a \parallel \text{node } \Gamma b)) \wedge \\ & \text{S} = (\bigcup p \in \text{N}. (\text{THE S'. node } \Gamma p \text{ S'})) \wedge \\ & (\forall a \in \text{N}. \text{is-valid}(\text{node } \Gamma a)) \end{aligned}$$

### 5.1.2 Disjointness of Object Sets

Assertions in algorithms manipulating object graphs are concerned with single objects or sets of objects. In order to reason about them effectively, we construct a lazy unfolding procedure (Section 3.4) which is able to prove subsets of a set of objects disjoint.

In the location phase, the rules (30) and (31) identify single objects or sets of objects in a given set  $\text{N}$ . The completion procedure from Section 3.2 ensures that any structure of the single objects, for instance if they are of a defined record type, will also be unfolded automatically.

$$\frac{p \in \text{N}}{\text{node } \Gamma p \preceq \text{nodes } \Gamma \text{N}} \quad (30)$$

$$\frac{A \subseteq B}{\text{nodes } \Gamma A \preceq \text{nodes } \Gamma B} \quad (31)$$

The lazy computation of a suitable unfolding rule must then prove disjoint any number of objects located in this way. For this task, the premises of the applied rule instances are available. For a unified treatment, single nodes are first converted to singleton sets:

$$\text{node } \Gamma p = \text{nodes } \Gamma \{ p \}$$

The procedure is thus given a set of inclusions  $A_i \subseteq B$  and has to prove the objects at  $A_1 \dots A_n$  disjoint. It performs this proof iteratively by computing a sequence of sets  $C_i$ , for  $i = 1, \dots, n$ , such that  $C_i = \bigcup_{j=1}^i A_j$ . In each step, it derives two theorems: the disjointness of the node sets (32) and their embedding into the overall set (33).

$$\text{nodes } \Gamma C_i = \text{nodes } \Gamma A_1 \parallel \dots \parallel \text{nodes } \Gamma A_i \quad (32)$$

$$C_i \subseteq B \quad (33)$$

The final set  $C_n$  thus obtained may not be equal to  $B$ . In order to unfold  $B$  by an equality (Section 3.3), the procedure applies the following rule in a finishing step.

$$\frac{C \subseteq B}{\text{nodes } \Gamma B = \text{nodes } \Gamma C \parallel \text{nodes } \Gamma (B - C)} \quad (34)$$

The remaining task is then to prove  $C_i$  and  $A_{i+1}$  disjoint in each step. The central idea is that since the objects in the node set are not distinguishable by any underlying structure, they must be distinguishable by their properties. Rule (35) expresses this insight: if the assumption that the sets overlap leads to a contradiction, the corresponding memory objects must be disjoint.

$$\frac{\forall p. p \in A \wedge p \in B \longrightarrow \text{False}}{\text{nodes } \Gamma (A \cup B) = \text{nodes } \Gamma A \parallel \text{nodes } \Gamma B} \quad (35)$$

The contradiction in the premise is proven using Isabelle's `metis` resolution prover. It also receives any contextual hypotheses available at the point where the unfolding takes place, such that knowledge about variables occurring in the terms  $A$  and  $B$  can be taken into account.

### 5.1.3 Burstall-style Disjointness Proofs

Many verification methodologies assume Burstall’s split-heap memory model [1], also called the component-as-array model. That model exploits an invariant on heap representations that holds in many strongly typed, and in particular object-oriented languages: records never overlap partially and pointers are always references to the start of records. Consequently, different fields of any pair of records are known to be disjoint, independently of which records on the heap are actually concerned. The memory can therefore be modelled as a collection of arrays, where each array represents a field from the record types occurring in the program. The pointers to records are used as indexes to access their fields in these arrays.

With this modelling, interactive proofs [22,9] as well as fully automatic proofs [2] become feasible. The same method can also be applied to C programs that obey suitable syntactic restrictions [3,34] or that can be proven to respect the invariant in parts of the code [5]. Low-level garbage collection [32] has been verified by deriving the assembly code from a typed intermediate language.

The main incentive for using Burstall’s memory model is that its strong assumptions enable fast disjointness proofs: since different fields are modelled as distinct arrays, it is clear syntactically that updates to one field will not affect any other fields. While this efficiency is desirable, introducing the memory model directly precludes the usage of pointer arithmetic from Section 1. We therefore now give an integration into our framework that enables similarly efficient reasoning on top of the low-level linear memory model used in this paper; furthermore, the integration is automatic: where the special case of Burstall-style reasoning applies, that unfolding is chosen, otherwise the general reasoning from Section 5.1.2 is employed.

The central idea of the integration is to translate Burstall’s assumption that all fields in the program are known beforehand to the context of our framework. Whenever the `node` underlying a given set of `nodes` is found to be a record, that record’s definition provides an unfolding for its fields by (21). Using this unfolding, we will pre-compute rules for the location and unfolding phases and thus avoid the contradiction proofs necessary for (35).

Our implementation detects the special case of records being used as nodes using Isabelle’s locale infrastructure [27,28]. Whenever a new instance of the generic nodes theory from Section 5.1.1 is requested, we try to prove the following theorem by unfolding the definition of the `node`.

$$\forall r. \text{node } \Gamma r = \text{typed-block } \Gamma r t \quad (36)$$

Then, the instance type `t` is examined to retrieve the name of the record and its unfolding rule generated by (21). For each field of the record, we then instantiate the generic rule (37) to be used in the location phase. Its first premise is solved by (36), the second one is proven directly from the record’s unfolding rule. The remaining premise  $p \in \mathbf{N}$  expresses that the record in question is actually allocated within the node-set under consideration.

$$\frac{\begin{array}{l} \forall r. \text{node } \Gamma r = \text{typed-block } \Gamma r t \\ \forall r. \text{field-block } \Gamma r t f \preceq \text{typed-block } \Gamma r t \\ p \in \mathbf{N} \end{array}}{\text{field-block } \Gamma p t f \preceq \text{nodes } \Gamma \mathbf{N}} \quad (37)$$



Our own definition follows that of list fragments (e.g. [9,22]). List fragments generalize the notion of complete, `null`-terminated lists for use in stating loop invariants. The predicate `list p ps q` expresses that the nodes `ps` are found starting at `p` before finally reaching `q`. In consequence, `q` itself is not contained in the list `ps`. This can be exploited by restricting assertions to the nodes preceding some node `q` currently being modified or examined.

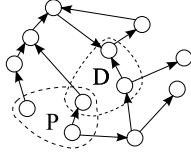
The transfer from list fragments to reachability consists in introducing a *boundary set* `Q` such that the set `R` in definition (41) is the set of nodes reachable from `P` without crossing or touching the boundary `Q`.

$$\begin{aligned} \text{reachable } \Gamma P Q R M \equiv \\ (R = \{q. \exists p \in P. \exists ps. \text{path } \Gamma p ps q M \wedge \text{set } ps \cap Q = \{\} \wedge q \notin Q\}) \end{aligned} \quad (41)$$

With this definition, the `reachable` predicate depends on the reachable nodes `R`, but never on the nodes in the boundary set `Q`. Lemma (42) therefore enables our framework to determine whether a memory update influences reachability in the object graph.

$$\text{accesses } (\text{reachable } \Gamma P Q R) M (\text{nodes } \Gamma (R - Q)) \quad (42)$$

The boundary set in reachability furthermore simplifies reasoning about algorithms working on the object graph. Whenever any reachable node is modified, the graph structure and thus reachability may be altered. To handle these modifications, one must be able to split the graph at a set of modified nodes. The intuition behind this step is depicted in Figure 1: any node that was reachable from `P` only by crossing through the set `D` will also be reachable from some successor of a node in `D`.



**Fig. 1** Splitting Reachability from `P` at Node Set `D`

Lemma (44) captures this intuition directly. It makes use of the auxiliary definition (43) which collects the successors of a set of nodes.

$$\text{Succs } \Gamma P M \equiv \bigcup_{p \in P} \text{set } (\text{succs } \Gamma p M) \quad (43)$$

$$\frac{\text{reachable } \Gamma P Q R M \quad D \subseteq R \quad D \cap Q = \{\}}{(\exists R1 R2. \text{reachable } \Gamma P (Q \cup D) R1 M \wedge \text{reachable } \Gamma (\text{Succs } \Gamma D M) (Q \cup D) R2 M \wedge R = R1 \cup R2 \cup D)} \quad (44)$$

The split parts of an object graph can then be recombined using the obvious fact (45).

$$\frac{\text{reachable } \Gamma P Q R M \quad \text{reachable } \Gamma P' Q R' M}{\text{reachable } \Gamma (P \cup P') Q (R \cup R') M} \quad (45)$$

It is worth noting the case distinction introduced by (42) and (44) for the manipulation of a single node `p` in the reachable set `R`. The node `p` can be either in `Q`, in which case (42) asserts that the reachability within the graph is not altered by the

modification. If it is not in  $Q$ , then (44) allows us to split the graph at  $p$  such that both assertions about reachability are independent of any modifications to  $p$ . After the modification, the two parts of the graph can be re-joined by (45), taking into account any local modifications to the successors of  $p$ .

We will see a further use of the boundary set in the loop invariant of the Schorr-Waite algorithm (Section 5.2), which must express that any unmarked node is reachable from a given stack of nodes without crossing already marked nodes. Setting the marked nodes as the boundary set expresses this intention very succinctly.

The following general lemmas, which directly follow from the definition of reachability, finish our development: no nodes will be reachable starting from nodes in the boundary set (46); the set of start nodes, except for the boundary set, is directly reachable (47); removing the boundary nodes from the start nodes does not change reachability (48); finally, reachability is closed under following references (49).

$$\frac{P \subseteq Q}{\text{reachable } \Gamma P Q R M = (R = \{\})} \quad (46)$$

$$\frac{\text{reachable } \Gamma P Q R M}{P - Q \subseteq R} \quad (47)$$

$$\text{reachable } \Gamma P Q R M = \text{reachable } \Gamma (P - Q) Q R M \quad (48)$$

$$\frac{\text{reachable } \Gamma P Q R M \quad r \in R}{\text{set } (\text{cell-succs } \Gamma r M) \subseteq R \cup Q} \quad (49)$$

### 5.1.5 Evaluation

It has been claimed that there is a fundamental choice involved between a split-heap model and a general, low-level memory model [5]. In contrast, we have shown that our framework can emulate on a linear memory model the efficient reasoning supported by the split-heap model, in the special case where the memory is used to store records (Section 5.1.3). The only new proof-obligation is the premise  $p \in N$  in (37). It reflects the requirement that the record at  $p$  is allocated, and therefore corresponds to a special hidden field `$allocated` introduced in other methodologies (e.g. [3]), which represents the same information and induces similar proof obligations. If the special case does not apply, the framework falls back on the general unfolding from Section 5.1.2.

Beyond the split-heap model, that unfolding procedure can prove disjointness for sets of nodes, as opposed to single nodes. The definitions in Section 5.1.4, and in particular the accessed nodes of reachability (42), exploit this generality in a formalization of object graphs that is close to an informal understanding and will yet be sufficient to verify the Schorr-Waite algorithm.

Furthermore, both the general and the split-heap unfoldings can be combined with the unfoldings introduced in Section 4. For instance, an array in a record field belonging to some object in a graph will be handled in the same manner as arrays allocated in the heap or local variables (Section 4.2). Going one step beyond, it is also possible to re-interpret, for instance, an `int` field in a record as an array of bytes and thus to manipulate the byte-representation of values [20].

Finally, the frame conjuncts in function specifications (Section 2.7) can be used to specify very precise modifies-assertions in a general fashion: using (34), the frame computation can isolate unmodified single nodes as well as sets of nodes from the set of nodes accessed by a function.

## 5.2 The Schorr-Waite Algorithm

This section gives our proof of the Schorr-Waite graph marking algorithm, building on the development of object graphs in Section 5.1. Due to the automated unfolding introduced there, the reasoning can closely follow the informal understanding. To highlight this connection, we will present both the informal and the formal view in some detail.

### 5.2.1 The Algorithm

The Schorr-Waite algorithm traverses an object graph to mark all nodes reachable from a given set of root pointers. Its distinguishing feature is the fact that it does not require additional space for a recursion stack, but keeps the stack within the traversed objects. Towards that end, it overwrites pointer fields in the objects with links to the next stack node and restores the original pointers when removing nodes from the stack.

The specification and code is given in Figure 2. Following previous studies, we use a version where objects have only two reference fields `l` and `r` and only a single root pointer is given. We include in the object definition a *mark field* `m` and *control field* `c`. The control field is used for the internal bookkeeping and will be explained below.

```
struct cell {
  bool m;
  bool c;
  struct cell *l;
  struct cell *r;
}
```

The verification environment [7] generates for this definition constants to access the four fields, named `cell-m-rd`, `cell-c-rd`, `cell-l-rd`, and `cell-r-rd`. These yield the stored byte sequences, which then can be lifted to HOL values using `to-bool` and `to-int`. For the subsequent development, we prefer these constants to the external syntax used in Section 4.1, since they reflect the involved memory accesses more explicitly.

The specification of the algorithm in Figure 2 states that it marks all nodes reachable from a given `root` (without crossing `null` pointers) and furthermore correctly restores the references within all objects to the values in the pre-state `M0`. The specification uses the following auxiliary definitions:

$$\begin{aligned} \text{marked } \Gamma \text{ p M} &\equiv \text{to-bool (cell-m-rd } \Gamma \text{ p M)} \\ \text{cell-succs } \Gamma \text{ p M} &\equiv [ \text{to-ptr (cell-l-rd } \Gamma \text{ p M)}, \text{to-ptr (cell-r-rd } \Gamma \text{ p M)} ] \end{aligned}$$

The latter definition is also used to obtain an instance `g` (for “graph”) of the generic theory from Section 5.1 by setting `succs` to `cell-succs`. Because that theory is independent of the variables defined in the execution context  $\Gamma$  (Section 2.1), we introduce a *static context*  $\Gamma_s$ . The predicate `ctx-var-change` asserts that  $\Gamma$  and  $\Gamma_s$  differ only in the defined variables. The algorithm’s pre-condition finally asserts that the local variables `root`, `t`, `p`, and `q`, as well as the nodes `N` are allocated.

The algorithm works with two pointers to represent the current state. The *tip* `t` is the next node to be examined, the *predecessor* `p` points to the node from which `t` was reached. At the same time, `p` is the head node of the linked recursion stack. The algorithm’s main loop continues to work until the stack is exhausted and the tip itself

```

{ M ▶ «root» || «t» || «p» || «q» || g.nodes Γs N ∧ ctx-var-change Γ Γs ∧
  reachable Γs { root } { null } N M ∧
  (∀n ∈ N. ¬ marked Γs n M) ∧ M = M0 }

t = root;
p = null;
while (p != null || (t != null && !t → m)) {
  if (t == null || t → m) {
    if (p → c) { // pop
      q = t; t = p; p = p → r; t → r = q;
    } else { // swing
      q = t; t = p → r; p → r = p → l; p → l = q; p → c = true;
    }
  } else { // push
    q = p; p = t; t = t → l; p → m = true; p → l = q; p → c = false;
  }
}

{ (∀p ∈ N. marked Γ p M) ∧ (∀p ∈ N. cell-succs Γ p M = cell-succs Γ p M0) }

```

Fig. 2 The Schorr-Waite Algorithm

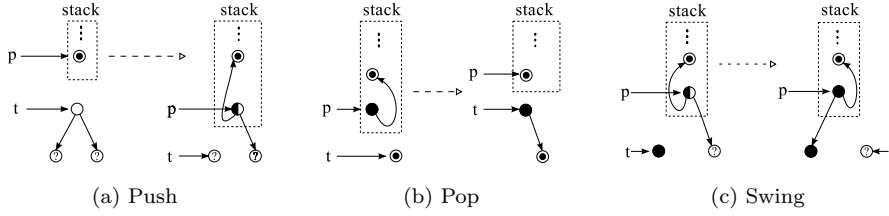


Fig. 3 Actions of the Schorr-Waite Algorithm

does not need to be marked. The loop body examines the fields of  $t$  and  $p$  to take one of the three actions *push*, *pop*, and *swing*, which are represented graphically in Figure 3.

Before we explain the single steps, the meaning of the control field  $c$  needs clarification. For nodes in the stack, it indicates which of the node's fields points to the next node in the stack, where  $c = \text{false}$  corresponds to field  $l$  and  $c = \text{true}$  corresponds to field  $r$ . In Figure (3), a half-filled circle indicates  $c = \text{false}$ , a filled circle  $c = \text{true}$ . Furthermore, a solid dot in the circle indicates  $m = \text{true}$  without knowledge about  $c$ , and a question mark denotes that the node's status is yet unknown.

The *push* step occurs when the tip is unmarked. In that case, the algorithm must prepare to visit both its left and its right successor. This is achieved by setting  $t$  to the left successor immediately and keeping the newly marked node in the stack, for a later traversal of its right successor. According to the above explanation,  $c$  is set to *false*. Note that  $p$  is, indeed, the predecessor of the new tip  $t$ . With this information, the field  $l$  in the new  $p$  becomes redundant and can be used to create the stack structure.

The *pop* step is complementary to *push*. It is taken when the tip is already marked and also the predecessor  $p$  is completely processed, which is seen by  $c = \text{true}$ . Then, field  $r$  in  $p$  is restored to its original value, where it pointed to  $t$ , and  $p$  and  $t$  are shifted one link up the stack.

Finally, the *swing* step occurs when the tip is already marked, but its predecessor is not completely processed ( $c = \text{false}$ ). In this case, the tip is known to be the original

l successor of  $p$  and must be set to the  $r$  successor of  $p$ . The corresponding pointer assignments and change in  $p \rightarrow c$  are straightforward.

### 5.2.2 Capturing the Stack Structure

We have developed a generic theory of singly linked lists, analogously to the development of object graphs in Section 5.1. It can be instantiated by defining a constant for reading the single successor of a list node and proving that its result depends only on that list node. The special stack structure for the Schorr-Waite algorithm is therefore given by the following constant, which evaluates the control field of object nodes to determine the next node in the stack.

$$\text{stack-succ } \Gamma \ p \ M \equiv \text{if to-bool (cell-c-rd } \Gamma \ p \ M) \\ \text{then to-ptr (cell-r-rd } \Gamma \ p \ M) \\ \text{else to-ptr (cell-l-rd } \Gamma \ p \ M)$$

The instance  $s$  (for “stack”) of the list theory then provides basic facts about the lists, and hence the stacks. The stack itself consists of the nodes of the singly linked list:

$$\text{stack } \Gamma \ p \ S \ M \equiv s.\text{nodes } \Gamma \ p \ \text{null } S \ M$$

Verification in lightweight separation proceeds by symbolic forward execution, which emulates the runtime modifications on the memory state by corresponding re-writing of assertions [7]. When the algorithm executes assignments that correspond to push- and pop-operations on the stack, the reasoning must perform corresponding operations on the assertions: Lemma (50) formalizes a *push* operation, Lemma (51) the corresponding *pop* operation ( $\#$  denotes the cons-operator in HOL). Unfolding the `stack-succ` constant in both distinguishes between different settings of the control field in  $p$ .

$$\frac{\text{stack } \Gamma \ p' \ S \ M \quad p \notin \text{set } S \quad p \neq \text{null} \quad \text{stack-succ } \Gamma \ p \ M = p'}{\text{stack } \Gamma \ p \ (p \# S) \ M} \quad (50)$$

$$\frac{\text{stack } \Gamma \ p \ S \ M \quad p \neq \text{null}}{\exists S'. S = p \# S' \wedge p \notin \text{set } S' \wedge \text{stack } \Gamma \ (\text{stack-succ } \Gamma \ p \ M) \ S' \ M} \quad (51)$$

For the initialization of the loop invariant and the deduction of the post-condition from the invariant after the loop finishes, we need the following lemma ( $[]$  is the empty list):

$$\text{stack } \Gamma \ \text{null } S \ M = (S = []) \quad (52)$$

### 5.2.3 The Loop Invariant

The algorithm’s loop repeatedly performs one of three atomic steps until neither the tip nor the stack contain further nodes to be processed. As usual, the loop invariant has to capture to current state of the overall marking process: it must describe which nodes have already been marked and which nodes still need to be marked. Because the object references of the graph are used temporarily for building the stack, it furthermore must describe how this stack can be reconstructed.

Our formulation follows that of Mehta and Nipkow [22]. However, it adds assertions to ensure allocatedness of all accessed nodes, which could be ignored in the split-heap model used there and makes use of the reachability predicate developed in Section 5.1.4.

$$\begin{aligned}
& \exists S. \text{stack } \Gamma_s \text{ p S M} \wedge \text{set S} \subseteq \mathbf{N} \wedge (\forall q \in \text{set S. marked } \Gamma_s \text{ q M}) \wedge \\
& (\text{t} = \text{null} \vee \text{t} \in \mathbf{N}) \wedge \\
& \text{reachable } \Gamma_s \\
& \quad (\{\text{t}\} \cup \text{set} (\text{map } (\lambda n. \text{to-ptr } (\text{cell-r-rd } \Gamma_s \text{ n M})) \text{S})) \\
& \quad \{\text{n. n} = \text{null} \vee \text{n} \in \mathbf{N} \wedge \text{marked } \Gamma_s \text{ n M}\} \\
& \quad \{\text{n} \in \mathbf{N}. \neg \text{marked } \Gamma_s \text{ n M}\} \text{M} \wedge \\
& (\forall \text{p} \in \mathbf{N}. \text{p} \notin \text{set S} \longrightarrow \text{cell-succs } \Gamma \text{ p M} = \text{cell-succs } \Gamma \text{ p M0}) \wedge \\
& \text{stack-reco } \Gamma \text{ t S M0 M} \wedge \\
& \text{reachable } \Gamma_s \{\text{root}\} \{\text{null}\} \mathbf{N} \text{M0} \wedge \\
& \text{M} \blacktriangleright \text{root} \parallel \text{t} \parallel \text{p} \parallel \text{q} \parallel \text{g.nodes } \Gamma_s \mathbf{N} \wedge \\
& \text{ctx-var-change } \Gamma \Gamma_s
\end{aligned}$$

**Fig. 4** Loop Invariant of Schorr-Waite Algorithm

The invariant is given in Figure 4. The first line describes the stack structure: the nodes  $S$  are all contained in the set  $N$  to be marked, and the nodes in  $S$  itself are already marked. The second line captures allocatedness of the tip node. The next conjunct expresses the progress of marking: any nodes from  $N$  that are not already marked are reachable from the tip or the ( $x$ -fields of) the stack nodes. Note how our boundary set (Section 5.1.4) expresses concisely that marked nodes need not be traversed.

The next two conjuncts capture that the modifications to the graph structure performed in the *push* and *swing* steps can be undone to leave the reference fields in nodes in their original state after the algorithm. Outside of the stack, the cell-successors are already set correctly. Furthermore, the stack itself is *reconstructible* according to the following recursive definition (adapted from [22]).

$$\begin{aligned}
\text{stack-reco } \Gamma \text{ t [] M0 M} &= \text{True} \\
\text{stack-reco } \Gamma \text{ t (p \# S) M0 M} &= \\
& (\text{if to-bool } (\text{cell-c-rd } \Gamma \text{ p M}) \\
& \quad \text{then } (\text{cell-l-rd } \Gamma \text{ p M0} = \text{cell-l-rd } \Gamma \text{ p M} \wedge \text{to-ptr } (\text{cell-r-rd } \Gamma \text{ p M0}) = \text{t}) \\
& \quad \text{else } (\text{cell-r-rd } \Gamma \text{ p M0} = \text{cell-r-rd } \Gamma \text{ p M} \wedge \text{to-ptr } (\text{cell-l-rd } \Gamma \text{ p M0}) = \text{t})) \wedge \\
& \text{stack-reco } \Gamma \text{ p S M0 M}
\end{aligned}$$

The last three conjuncts of the invariant add general bookkeeping information: we maintain that  $N$  was defined to be the set of reachable nodes, which entails that  $N$  is closed under following non-null references. Then, the memory layout consists of the local variables and the set  $N$  of memory objects.

#### 5.2.4 Verification of the Algorithm

We begin with the two straightforward proof obligations. First, we have to prove that the loop invariant holds after the initialization of  $p$  and  $t$ . This is accomplished directly by applying the Lemmas (52) and (46), using  $p = \text{null}$ , and (47) to show that  $t \in N$ . Second, the loop invariant implies the algorithm's post-condition when the loop finishes again by (52) and (46). The generic lemmas derived from the definition of reachability are thus sufficient in both cases.

The main proof obligation consists in the maintenance of the loop invariant by the loop body. We treat the three cases in the loop body separately, referring to Figure 3 for the idea of the proof. In each case, the proof consists of three phases, which closely follow the execution of the algorithm due to the forward proof style developed in [7].

The first phase occurs directly after the while- and if-conditions have been evaluated. There, the concrete situation among those of Figure 3 has been identified. We apply suitable lemmas to make all assertions about the node that is modified explicit. The second phase is automatic and executes the assignments symbolically by modifying the assertion valid for the current memory state. It uses the framework from Section 3 and hence the extension from Section 5.1 in the background. In the final third phase, we prove that the assertion thus obtained implies the loop invariant.

The first case in the algorithm is the *pop* case. Forward execution of the loop- and if-conditions gives us the invariant and the additional conjuncts

$$t = \text{null} \vee t \neq \text{null} \wedge \text{marked } \Gamma_s t M \quad p \neq \text{null} \quad p \in N$$

Since the head node  $p$  of the stack will be modified, we expose it in the assertion by (51), where the following conjuncts replace the original **stack**-assertion. Rewriting with the new equality yields the pointers exactly as depicted in Figure 3 (a).

$$\text{stack } \Gamma_s (\text{cell-r-rd } \Gamma_s p M) S' M \quad S = p \# S' \quad p \notin \text{set } S'$$

Automatic symbolic forward execution results in an assertion that directly describes the right part of Figure 3 (b): the previous  $t$  is known to be marked, and also  $t$  itself is already marked, and its control field is **true**. In particular, the reachability conjuncts are unchanged, because the manipulated node  $p$  was already marked and thus contained in the boundary set (see (42)). For the maintenance of the invariant, Isabelle eliminates all conjuncts except the real proof obligations: reachability of all unmarked nodes and the fact that nodes outside the stack are in their original state. The latter is solved by the definition of **cell-succs**. To prove reachability, Figure 3 (b) suggests that it is sufficient to prove that no unmarked nodes would have been found by following the previous  $p$ 's **r**-field, since this is already marked. This proof step is immediate by (48).

The *swing* branch is analogous to *pop*. The stack node  $p$  is exposed by (51). Automatic forward execution then results in the state depicted in the right part of Figure 3 (c). Differing from the *pop* case, we then push node  $p$  back onto the stack by (50). By the same reasoning as before, the previous  $t$  node is already marked, and we apply (48) to establish reachability of all unmarked nodes. Finally, we need to show that the  $t \in N$ , which is given by the **stack-reco** predicate and (49).

The *push* case in Figure 3 (a) marks a yet unmarked node  $t$ . By (42), this will influence the reachability of all unmarked nodes from  $t$  and the stack. Lemma (44) provides the solution in splitting the object graph at the node  $t$ . Similarly, we rewrite

$$\forall p \in N. p \notin \text{set } S' \longrightarrow \text{cell-succs } \Gamma_s p M = \text{cell-succs } \Gamma_s p M0$$

into

$$\begin{aligned} \text{cell-l-rd } \Gamma_s t M &= \text{cell-l-rd } \Gamma_s t M0 \wedge \\ \text{cell-r-rd } \Gamma_s t M &= \text{cell-r-rd } \Gamma_s t M0 \wedge \\ \forall y \in N - \{t\}. y \notin \text{set } S' &\longrightarrow \text{cell-succs } \Gamma_s y M = \text{cell-succs } \Gamma_s y M0 \end{aligned}$$

With this setup, automatic forward execution succeeds and leaves a state that directly corresponds to the right part of Figure 3 (a). Allocatedness of the new  $t$  is again proven by (49). Lemma (48) allows us to use that the new head  $p$  of the stack has just been marked. Then, Lemma (45) undoes the split by (44) and re-establishes the reachability of all unmarked nodes from  $t$  and the stack.

### 5.2.5 Evaluation

We have given a detailed exposition of the Schorr-Waite algorithm to demonstrate that even for technically involved examples, lightweight separation enables proofs that closely follow the understanding gained from an informal presentation (Section 5.2.1). In all three branches of the algorithm (Figure 2), the assertions generated by forward execution directly match the pointer diagrams from Figure 3, and the proof ideas are found by interpreting these diagrams.

The basis for this result is the automation developed in Section 5.1, which in turn relies on the framework from Section 3. The cornerstone of the proofs is a novel way of reasoning about reachability in the presence of memory updates: the introduction of a boundary set into the definition (41) of reachability enables automatic proofs of disjointness from a modified memory region, as well as manipulations of assertions by (44) and (45) that directly reflect the informal reasoning.

## 6 Related Work

The problem of proving the disjointness of memory regions has recently attracted much attention in connection with the verification of object-oriented programs. Kassios [35] proposes to express the memory region occupied by an object’s representation as an additional specification variable. The disjointness of the regions, hence the independence of assertions from particular memory operations, can then be asserted without breaking encapsulation. The approach has been implemented by several authors [4, 2]. The work relies on Burstall’s memory model and uses direct pointer comparisons throughout. It does therefore not scale directly to low-level memory models, such as the one considered in this paper. Greve [36] addresses assembler programs and proposes to express the memory region occupied by data structures by functions with the intention of proving disjointness of modified memory regions symbolically. He leaves open the question of how to reason about the memory footprint of these functions, i.e. how to prove that their result remains unchanged by modifications to memory. Neither of these approaches covers re-interpretations and structural refinement of memory layouts beyond the field-level access encompassed by Burstall’s memory model.

Separation logic [17, 6] enables the succinct statement of assertions about heap data structures. The Smallfoot tool [10, 37] automates reasoning for a restricted set of formulae that is sufficient to capture the shapes of data structures. Tuerk [21] formalizes the Smallfoot logic in HOL and extends the approach to handle the content of data structures in parallel with their structure. He uses [38] to cover call-by-reference with entire local variables, but not with more general memory objects. Botinčan et al. [39] apply a similar system to a variant of C and employ an SMT solver for the pure parts of formulae. Chlipala et al. [40] apply separation logic to imperative higher-order programs. Their tactic `sep` [40, §3] for reasoning about spatial formulae matches conjuncts (like [10, 21, 39]), taking into account embedded existential quantifiers and pure formulae. For user-defined data structures and predicates, it can be parameterized by ad-hoc tactics that apply structural unfolding rules before the match. All mentioned tools build on an unfolding mechanism with undirected search in a restricted form of separation logic. Although separation logic in principle can be applied to a linear memory model, they use more structured models mapping addresses to values.

Tuch [13] applies separation logic to structured data types for a low-level memory model. In particular, he develops a theory for struct types in C. The automatic reasoning provided is limited: a specialized tactic allows the user to fold and unfold struct definitions as needed. Neither arrays nor references to local variables are supported. Cohen et al. [16] establish a typed memory model over untyped C memory states by expressing the additional disjointness invariants using a ghost variable. Special statements `split` and `join` in the language enable the user to manipulate the layout. The approach handles structs and is extensible to bit-fields and arrays. The frame inference problem for functions is not discussed, nor are extensions to automatic unfolding.

The Schorr-Waite algorithm is commonly regarded as a benchmark problem for the expressiveness for verification methods [9] and has therefore been used as an example in many studies, in each case highlighting the special features of the approach under consideration. Bornat [9] pioneers the field of mechanized proofs by discussing the fundamental problem of heap updates in the presence of possible aliasing, the component-as-array model, and the treatment of inductively defined data structures. His definition of reachability [9, §7.1] introduces an *exclusion set*, which rules out cyclic paths and effectively imposes a DAG structure on the reachable nodes. Bornat observes that this definition spatially separates the root of the DAG, which is also currently manipulated by the algorithm, from other reachable nodes, which simplifies reasoning. In comparison, the boundary set of our definition (Section 5.1.4) more generally serves to delineate sets of reachable nodes, and enables a corresponding reasoning by (44) and (45); in particular, it is not restricted to the introduction of a DAG structure and the manipulation of its root node.

Giorgino et al. [26] verify the graph marking algorithm as a refinement of a tree marking variant, by reasoning about its behaviour on certain spanning trees of the general graph, instead of reasoning about a DAG.

Mehta and Nipkow [22] formulate the higher-order loop invariant on which our own development is based. The main differences arise precisely from our use of the framework introduced in this paper. Where Mehta and Nipkow prove special *separation lemmas* for each inductive predicate, which are formulated directly in terms of the updates in the component-as-array model, we only need to capture the memory region that a predicate depends on. The novel feature of our reachability definition, the introduction of a boundary set, enables the statement of (42), which in turn leads to a proof that closely reflects informal arguments about pointer diagrams by (44) and (45).

The particular problems arising from the use of first-order logic are discussed by Hubert and Marché [23] and Bubel [24]. Both studies take as given a binary reachability predicate between nodes, with a suitable axiomatization. The problems of heap updates and aliasing are not discussed specifically, and will be solved according to the component-as-array-model used in [9, 22].

Yang [25, Ch. 7] gives a pen-and-paper proof using local reasoning in BI. His loop invariant [25, §7.3.1] is remarkable in that it uses two views on the object graph, i.e. the same heap fragment: one describing the stack structure together with the later reconstruction of the nodes' fields, and the other one separating the marked from the unmarked graph nodes. He then shows how to lift the local reasoning (in the sense of [17]) about the loop body to a global reasoning about these two parts of the invariant. Yang, too, uses an argument about the spanning tree of the graph to isolate the node currently manipulated by the algorithm.

Our proof differs from those surveyed above by using a linear memory model, while the others are based on the component-as-array model or other more structured mod-

els (including the C implementation in [23]). The automatic treatment of separation reasoning by the framework developed in this paper, together with a novel formulation of reachability, enables the proof to follow closely the reasoning by pointer diagrams. In particular, we do not have to introduce auxiliary tree- or DAG structures. While Yang’s specification in BI also reflects a diagrammatic intuition, our reliance on classical logic enables us to employ Isabelle’s automatic provers where Yang has to argue about detailed formula manipulations in the non-classical BI.

## 7 Conclusion

We have presented a method for automatic reasoning about memory layouts in a flexible and extensible manner. A small language of layouts allows memory-related proof obligations arising in Hoare logics to be formulated succinctly. The actual elements of layouts are not fixed, but can be defined as needed: local variables, blocks accessed by pointers, structs, and arrays are readily formalized. Our reasoning framework supports a general notion of unfoldings of memory elements. Unfoldings comprise both structural refinements and re-interpretations of layout blocks. Structs and arrays can thus be split automatically into their constituent elements as needed, and local variables can be interpreted as memory objects, which allows them to be accessed by pointers in arbitrary ways. To introduce a new unfolding, it is generally sufficient to prove an unfolding equality theorem about a layout block. Using the flexibility, it is possible to verify idiomatic usages of C that are not currently covered by other verification methodologies.

The presented method is suitable for reasoning about a low-level, byte-addressed memory model. The key insight is to encode invariants about memory layouts into the definition of layout constants, and to propagate the invariants through unfoldings. As an illustration, the theory reduces reasoning about arrays to proving inequalities between indices without further side-conditions; overflows in the address arithmetic are excluded by the invariants associated with arrays.

At the same time, our framework also supports high-level reasoning about the manipulation of data structures by algorithms. We have verified the Schorr-Waite graph marking algorithm by giving an unfolding procedure that enables, for our low-level memory model, efficient disjointness proofs for heap-allocated records similarly to the split-heap memory model. Furthermore, it can directly reason about sets of nodes, which has enabled us to provide a theory about graphs of memory objects that is independent of the verified algorithm. A novel formulation of reachability with a boundary set yields theorems for manipulating assertions that closely reflect arguments about pointer diagrams. Since the introduced automation handles the technical details of aliasing and disjointness, the user only needs to solve proof obligations that are close to an informal presentation of the algorithm.

## References

1. Burstall, R.: Some techniques for proving correctness of programs which alter data structures. In Meltzer, B., Michie, D., eds.: Machine Intelligence. Number 7. Edinburgh University Press (1972)
2. Smans, J., Jacobs, B., Piessens, F., Schulte, W.: An automatic verifier for Java-like programs based on dynamic frames. In Fiadeiro, J.L., Inverardi, P., eds.: FASE. Volume 4961 of LNCS., Springer (2008) 261–275

3. Filiâtre, J.C., Marché, C.: The Why/Krakatoa/Caduceus platform for deductive program verification. In Damm, W., Hermanns, H., eds.: CAV. Volume 4590 of LNCS., Springer (2007) 173–177
4. Banerjee, A., Barnett, M., Naumann, D.A.: Boogie meets regions: a verification experience report. In Shankar, N., Woodcock, J., eds.: VSTTE'08. Volume 5295 of LNCS., Springer (2008) 177–191
5. Rakamaric, Z., Hu, A.J.: A scalable memory model for low-level code. In Jones, N.D., Müller-Olm, M., eds.: Verification, Model Checking, and Abstract Interpretation, 10th International Conference (VMCAI 2009). Volume 5403 of LNCS., Springer (2009)
6. O'Hearn, P.W., Reynolds, J.C., Yang, H.: Local reasoning about programs that alter data structures. In: Proceedings of the 15th International Workshop on Computer Science Logic. Number 2142 in LNCS, Springer (2001) 1–19
7. Gast, H.: Lightweight separation. In Ait Mohamed, O., Munoz, C., Tahar, S., eds.: Theorem Proving in Higher Order Logics 21st International Conference, TPHOLs 2008. Volume 5170 of LNCS., Springer (2008)
8. Gast, H.: Reasoning about memory layouts. In Cavalcanti, A., Dams, D., eds.: FM 2009: Formal Methods, Second World Congress. Volume 5850 of LNCS., Springer (2009)
9. Bornat, R.: Proving pointer programs in Hoare logic. In: Mathematics of Program Construction. (2000)
10. Berdine, J., Calcagno, C., O'Hearn, P.W.: Smallfoot: Modular automatic assertion checking with separation logic. In de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P., eds.: FMCO. Volume 4111 of LNCS., Springer (2005) 115–137
11. Norrish, M.: C formalised in HOL. PhD thesis, University of Cambridge (1998) Technical Report UCAM-CL-TR-453.
12. Dawson, J.E.: Isabelle theories for machine words. In: Seventh International Workshop on Automated Verification of Critical Systems (AVOCS'07). ENTCS (2007)
13. Tuch, H.: Structured types and separation logic. In: 3rd International Workshop on Systems Software Verification (SSV 08). (2008)
14. Schirmer, N.: Verification of Sequential Imperative Programs in Isabelle/HOL. PhD thesis, Technische Universität München (2005)
15. Kowaltowski, T.: Axiomatic approach to side effects and general jumps. *Acta Informatica* **7** (1977) 357–360
16. Cohen, E., Moskal, M., Schulte, W., Tobies, S.: A precise yet efficient memory model for C. In: 4th International Workshop on Systems Software Verification (SSV 2009). ENTCS, Elsevier Science B.V. (2009)
17. Reynolds, J.C.: 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)
18. Paulson, L.C.: Isabelle – A Generic Theorem Prover. Number 828 in LNCS. Springer, Berlin Heidelberg (1994)
19. Austern, M.H.: Generic Programming and the STL — using and extending the C++ Standard Template Library. Addison-Wesley (1998)
20. Gast, H., Trieflinger, J.: High-level reasoning about low-level programs. In Roggenbach, M., ed.: Automated Verification of Critical Systems 2009. Volume 23 of Electronic Communications of the EASST., EASST (2009)
21. Tuerk, T.: A formalisation of Smallfoot in HOL. In Berghofer, S., Nipkow, T., Urban, C., Wenzel, M., eds.: Theorem Proving in Higher Order Logics 22nd International Conference (TPHOLs 2009). Volume 5674 of LNCS., Springer (2009)
22. Mehta, F., Nipkow, T.: Proving pointer programs in higher-order logic. *Inf. Comput.* **199**(1-2) (2005) 200–227
23. Hubert, T., Marché, C.: A case study of C source code verification: the Schorr-Waite algorithm. In Aichernig, B.K., Beckert, B., eds.: SEFM, IEEE Computer Society (2005) 190–199
24. Bubel, R.: The Schorr-Waite-algorithm. In Beckert, B., Hähnle, R., Schmitt, P.H., eds.: Verification of Object-Oriented Software. The Key Approach. Volume 4334 of LNCS. Springer-Verlag New York, Inc., Secaucus, NJ, USA (2007)
25. Yang, H.: Local Reasoning for Stateful Programs. PhD thesis, Graduate College of the University of Illinois at Urbana-Champaign (2001)
26. Giorgino, M., Strecker, M., Matthes, R., Pantel, M.: Verification of the Schorr-Waite algorithm - from trees to graphs. In: Logic-Based Program Synthesis and Transformation (LOPSTR). (2010)

27. Chaieb, A., Wenzel, M.: Context aware calculation and deduction. In: *Calculemus '07 / MKM '07: Proceedings of the 14th symposium on Towards Mechanized Mathematical Assistants*, Berlin, Heidelberg, Springer-Verlag (2007) 27–39
28. Haftmann, F., Wenzel, M.: Local theory specifications in Isabelle/Isar. (2009) 153–168
29. McCreight, A., Shao, Z., Lin, C., Li, L.: A general framework for certifying garbage collectors and their mutators. *SIGPLAN Not.* **42**(6) (2007) 468–479
30. McCreight, A.: *The Mechanized Verification of Garbage Collector Implementations*. PhD thesis, Department of Computer Science, Yale University (2008)
31. Torp-Smith, N., Birkedal, L., Reynolds, J.C.: Local reasoning about a copying garbage collector. *ACM Trans. Program. Lang. Syst.* **30**(4) (2008) 1–58
32. Hawblitzel, C., Petrank, E.: Automated verification of practical garbage collectors. *SIGPLAN Not.* **44**(1) (2009) 441–453
33. Lin, C.X., Chen, Y.Y., Li, L., Hua, B.: Garbage collector verification for proof-carrying code. *JCST* **22**(3) (2007) 426–437
34. Filliâtre, J.C., Marché, C.: Multi-prover verification of C programs. In: *Sixth International Conference on Formal Engineering Methods (ICFEM'04)*. (2004)
35. Kassios, I.T.: Dynamic frames: Support for framing, dependencies and sharing without restrictions. In Misra, J., Nipkow, T., Sekerinski, E., eds.: *FM*. Volume 4085 of LNCS., Springer (2006) 268–283
36. Greve, D.: Scalable normalization for heap manipulating functions. In: *International Workshop on the ACL2 Theorem Prover and its Applications*. (2007)
37. Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P., Wies, T., Yang, H.: Shape analysis of composite data structures. In: *CAV 2007*. Volume 4590 of LNCS., Heidelberg, Springer (2007)
38. Parkinson, M., Bornat, R., Calcagno, C.: Variables as resource in Hoare logics. In: *LICS '06: Proceedings of the 21st Annual IEEE Symposium on Logic in Computer Science*, Washington, DC, USA, IEEE Computer Society (2006) 137–146
39. Botincan, M., Parkinson, M., Schulte, W.: Separation logic verification of C programs with an SMT solver. In: *4th International Workshop on Systems Software Verification (SSV 2009)*. *Electronic Notes in Theoretical Computer Science*, Elsevier Science B.V. (2009)
40. Chlipala, A., Malecha, G., Morrisett, G., Shinnar, A., Wisnesky, R.: Effective interactive proofs for higher-order imperative programs. In: *ICFP '09: Proceedings of the 14th ACM SIGPLAN international conference on functional programming*, New York, NY, USA, ACM (2009) 79–90