Being and Change: Reasoning About Invariance

Frank S. de Boer [1,2] and Stijn de Gouw[1]

[1] Centre Mathematics & Computer Science, Netherlands
[2] Leiden Advanced Institute of Computer Science, Netherlands
frb@cwi.nl, cdegouw@cwi.nl

Abstract. We introduce a new way of reasoning about invariance in terms of *foot-prints* in a Hoare logic for recursive programs with (unbounded) arrays. A foot-print of a statement is a predicate that describes that part of the state that can be changed by the statement. We define invariance of an assertion with respect to a foot-print by means of a logical operation. This new Hoare logic is applied in a new simpler and modular proof of correctness of the well-known Quicksort sorting algorithm.

1 Introduction

During a visit of Ernst-Ruediger Olderog at the CWI in 2014, together with Krzysztof R. Apt we discussed different alternative proofs of correctness of the well-known Quicksort sorting algorithm [Hoa62]. These discussions resulted in various proof strategies which have been further detailed by Ernst-Ruediger and which form the starting point of this paper.

Proving correctness of (imperative) programs in Hoare logic is in general a challenging task, even for what seem to be relatively simple programs (measured for example in terms of the lines of code). Most of the complications are due to the basic fact that an imperative program specifies what *changes*, whereas an assertion describes what *is*. Consequently, most of the effort in proving correctness goes in specifying and verifying what does *not* change, i.e., what is *invariant*.

Proving correctness of recursive programs in Hoare logic requires special auxiliary rules (axioms) for reasoning about invariance [AdBO09], the so-called *adaptation* rules. These rules are used to adapt a given correctness formula, for example by adding to the pre- and postcondition an invariant. Hoare introduced in [Hoa71] one rule, *the* adaptation rule, which generalizes these rules. In his seminal paper [Old83] Ernst-Ruediger Olderog studied the expressiveness and the completeness of the adaptation rule.

Programs with, for example, *array* variables give rise to *aliasing*, i.e., the phenomenon of two *syntactically* different expressions which refer to the same *memory location*. In the presence of aliasing we cannot determine *statically* anymore general invariant properties, whereas the standard adaptation rules are based on such a static determination, namely checking syntactically whether a given assertion contains occurrences of variables which appear in the given program. The adaptation rules, including Hoare's rule, therefore are of limited use

in proving invariant properties of (recursive) programs with arrays. This limitation in general greatly complicates the correctness proofs because it does not fully support *modularity*: invariant properties in general are verified in terms of the internal control structure of a given program.

For recursive programs with array variables we extend in this paper the standard pre-postcondition specification with a *footprint*. A footprint is a set of predicates indexed by an array name. The arity of the predicate equals that of the array (interpreted as a function). Such a predicate associated with an array describes that subset of the *domain* of the array which can be changed by the program. We show how to extend the syntactic characterization of invariance to these footprints by means of a logical operation. We prove soundness of this logical operation and apply the logic to the verification of the well-known Quicksort sorting algorithm, which results in a simpler and modular proof.

Related work A large body of related work focuses on reasoning about invariant properties of object-oriented programs. For example, *dynamic frames* [Wei11] have been introduced as an extension of Hoare logic where invariant properties are specified and verified in terms of an explicit heap representation. Such a representation however in general does not match the abstraction level of object-oriented languages like Java. In *separation logic* [Rey05] invariant properties of object-oriented programs are specified and verified in terms of a logical operation of *separation conjunction* which allows to split the heap into two disjoint parts. The resulting logic however is undecidable for its propositional subset [BK14]. Moreover, in [CYO01] it is shown that validity of the first-order language restricted to so-called "points to" predicate is not recursively enumerable, and as such not axiomatizable. In general, such axiomatizations are needed to prove formally the verification conditions which establish program correctness.

In contrast our approach, which can be extended to object-oriented programs (see Section 5), is based on standard predicate logic which allows established theorem proving techniques/engines. Further it allows reasoning at an abstraction level which coincides with the programming language, i.e., it does not require special predicates like the "points to" predicate.

2 Adaptation Rules

To allow for modular reasoning in Hoare logic, adaptation rules are needed to adapt the specifications in correctness formulas to a specific context. This section discusses four such adaptation rules [AdBO09], abstracting from the programming language: a conjunction rule, an existential introduction rule, an invariance rule, and a substitution rule. The invariance rule provides a basic form to reason about invariance using a simple *syntactic* test. A more precise *semantic* form, introduced in the next section, is needed for arrays. The adaptation rules are amenable to this extension; the next section shows that a modest addition to these rules suffices. In this section we clarify the relation between these adaptation rules and Hoare's single rule of adaptation [Hoa71], which was analyzed

by Olderog in [Old83]. The precise definitions of the adaptation rules are given below (for details about the standard logical operations used, like substitution $p[z := x]$ of z for x in p, we refer to [AdBO09]):

RULE A1: CONJUNCTION

$$\frac{\{p_1\}\ S\ \{q_1\}, \{p_2\}\ S\ \{q_2\}}{\{p_1 \wedge p_2\}\ S\ \{q_1 \wedge q_2\}}$$

RULE A2: $\exists$-INTRODUCTION

$$\frac{\{p\}\ S\ \{q\}}{\{\exists z : p\}\ S\ \{q\}}$$

where z does not occur in S or q.

RULE A3: INVARIANCE

$$\frac{\{p\}\ S\ \{q\}}{\{r \wedge p\}\ S\ \{r \wedge q\}}$$

where S does not assign to the variables in the formula r.

RULE A4: SUBSTITUTION

$$\frac{\{p\}\ S\ \{q\}}{\{p[z := t]\}\ S\ \{q[z := t]\}}$$

where z does not occur in S, and S does not assign the variables in the term t.

The invariance rule above provides a basic way to reason about assertions whose truth remains invariant under execution of S. However, in the presence of arrays, the invariance rule is rather crude. Due to the syntactic check in the side-condition, if *any* array element is assigned to, any assertion that mentions the array cannot be used with the invariance rule, even if the assertion accesses only those indices that are not assigned. The next section shows how to address this problem.

Hoare's rule of adaptation is:

RULE (H) : HOARE-ADAPT

$$\frac{\{p\}\ S\ \{q\}}{\{\exists z : (p \wedge \forall y : (q[x := y] \rightarrow r[x := y]))\}\ S\ \{r\}}$$

where z does not occur in S or r, x is the list of all variables occurring in S and y is a list of fresh variables.

The question arises: what is the relation between Hoare's rule (H) and the other adaptation rules? The next example by Olderog [Old83] shows that they differ in proof strength.

Lemma 1. *Let k be a variable that does not occur in S. From the correctness formula $\{x = k\}\ S\ \{x = k\}$ we can derive $\{x = k + 1\}\ S\ \{x = k + 1\}$ by A4, but not by rule (H).* □

Proof: To derive $\{x = k + 1\}\ S\ \{x = k + 1\}$ simply apply A4, substituting $k+1$ for k. To see that this is not derivable with (H), note that the precondition $\exists z : x = k \wedge \forall y : y = k \rightarrow y = k + 1$ given by (H) simplifies to $false$. □

Theorem 1. *In the presence of the consequence rule, (H) is derivable by A2, A3.* □

Proof: we show that from $\{p\}\ S\ \{q\}$, we can derive

$$\{\exists z : (p \wedge \forall y : (q[x := y] \rightarrow r[x := y]))\}\ S\ \{r\}$$

where z does not occur in S or r, x is the list of all variables occurring in S and y is a list of fresh variables. Assume

$$\{p\}\ S\ \{q\}$$

From A3:

$$\{p \wedge \forall y : (q[x := y] \rightarrow r[x := y])\}\ S\ \{q \wedge \forall y : (q[x := y] \rightarrow r[x := y])\}$$

Consequence rule:

$$\{p \wedge \forall y : (q[x := y] \rightarrow r[x := y])\}\ S\ \{r\}$$

From A2:

$$\{\exists z : (p \wedge \forall y : (q[x := y] \rightarrow r[x := y]))\}\ S\ \{r\}$$

□

Thus, in the presence of the consequence rule, theorem 1 and lemma 1 show that A1, A2, A3 and A4 are strictly stronger than rule (H).

3 Reasoning about invariance

The programming language We assume a basic imperative programming language featuring the usual sequential control structures. We distinguish between two kinds of (typed) variables: *simple* variables like $x, y, u, v, \ldots$ which range over elements of the included basic types **integer** or **Boolean**, and *array* variables like $a, b, \ldots$ of a higher type $T_1 \times \ldots \times T_n \rightarrow T$, where the argument types and the result type are basic types. Semantically arrays are functions, e.g., an array of type **integer** $\rightarrow$ **integer** is *unbounded*. Expressions are side-effect free (every operator in the language is semantically interpreted as a *total* function, e.g., division by zero results by definition to, for example, zero). A *subscripted* variable of type T is of the form $a[s_1, \ldots, s_n]$, where a is of some type $T_1 \times \ldots \times T_n \rightarrow T$

and s_i is an expression of type T_i, for $i = 1, \ldots, n$. For technical convenience only we restrict here to array assignments of the form $a[s_1, \ldots, s_n] := t$, where the argument expressions $s_1, \ldots, s_n$ do not contain subscripted variables. A program consists of a statement S and a set of procedure declarations $P(u_1, \ldots, u_n) ::= S$, with formal parameters $u_1, \ldots, u_n$ of a basic type and body S. A procedure call is of the form $P(t_1, \ldots, t_n)$, where t_i is an expression of a basic type which equals the one of the corresponding formal parameter.

Correctness formulas *Assertions* $p, q, \ldots$ are logical formula, defined as usual (as in [AdBO09]) (in contrast to program assignments, in assertions we do allow nested subscripted variables). By

$$F : \{p\}\ S\ \{q\}$$

we denote a *correctness formula* with a footprint F. A footprint F is a (finite) set of uniquely labeled formulas $a : p(x_1, \ldots, x_n)$, where n is the arity of array a. All the formulas of the footprint F are *syntactically invariant* in that they do not contain any program variables which can be affected by an execution of S

The *partial correctness* interpretation of $F : \{p\}\ S\ \{q\}$ (we assume an implicitly given set of procedure declarations D) extends that of $\{p\}\ S\ \{q\}$ with the following clause:

$\sigma \models p$ and
$< S, \sigma > \rightarrow^* < a[s_1, \ldots, s_n] := t; S', \sigma' >$ and $a : r(x_1, \ldots, x_n) \in F$
implies
$\sigma' \models r(s_1, \ldots, s_n)$.

Here $\rightarrow^*$ denotes the reflexive, transitive closure of the transition system for recursive programs (see Section 5.2 in [AdBO09]). In words, the above additional clause states that whenever an assignment to an array is executed the corresponding footprint should hold.

The Hoare logic of footprints The footprint of an array assignment is captured by the following rule:

RULE 1: ARRAY ASSIGNMENT

$$\frac{q[a[s_1, \ldots, s_n] := t] \rightarrow p(s_1, \ldots, s_n)}{\{a : p(x_1, \ldots, x_n)\} : \{q[a[s_1, \ldots, s_n] := t]\}\ a[s_1, \ldots, s_n] := t\ \{q\}}$$

Here the weakest precondition $q[a[s_1, \ldots, s_n] := t]$ is calculated by means of a *substitution* operation $[a[s_1, \ldots, s_n] := t]$ which takes into account aliasing (see Section 2.7 in [AdBO09]). For the soundness of this rule, we refer to [AdBO09] to a proof that $\sigma \models q[a[s_1, \ldots, s_n] := t]$ if and only if $\sigma[a[s_1, \ldots, s_n] := t] \models q$, where $\sigma[a[s_1, \ldots, s_n] := t]$ denotes the result of exexuting the assignment $a[s_1, \ldots, s_n] := t$ in σ. It remains to show that the above additional clause

defining the semantics of a foot-print is valid, i.e., $\sigma \models q[a[s_1, \ldots, s_n] := t]$ implies $\sigma \models p(s_1, \ldots, s_n)$. This follows immediately from the premise.

As a very simple example, it is trivial to derive

$$\{a : x = j\} : \{\mathbf{true}\}\ a[j] := 1\ \{\mathbf{true}\}$$

In order to reason *semantically* about invariance in terms of footprints we introduce the *restriction* $q \uparrow F$ of a formula q which "talks" only about that part of the state *disjoint* from the footprint. First we transform q in a formula q' in prenex normal form such that its matrix r is in disjunctive normal form. For technical convenience only and without loss of generality we assume that r contains no nested subscripted variables. The formula $q \uparrow F$ then can be obtained from q' by simply adding for each subscripted variable $a[s_1, \ldots, s_n]$ appearing as an argument in a literal the formula $\neg p_a(s_1, \ldots, s_n)$ to the conjunct in which the literal appears, where $p_a(x_1, \ldots, x_n) \in F$. More formally, we replace every literal $l(s_1, \ldots, s_n)$ in q' by $l(s_1, \ldots, s_n) \wedge \bigwedge_i \neg p_{a_i}(\bar{t}_i)$, where i ranges over those indices such that $s_i \equiv a_i(\bar{t}_i)$.

Given the above we can now introduce the following rule:

RULE 2: SEMANTIC INVARIANCE

$$\frac{F : \{p\}\ S\ \{q\}}{F : \{p \wedge r \uparrow F\}\ S\ \{q \wedge r \uparrow F\}}$$

where none of the *simple* variables which appear free in r appear in S at the right-hand-side of an assignment.

Let us illustrate the use of this latter rule by a very simple example. We want to prove

$$\{\forall i : i \neq j \rightarrow a[i] = 0\}\ a[j] := 1\ \{\forall i : i \neq j \rightarrow a[i] = 0\}$$

As already stated above it is trivial to derive from the above array assignment rule that

$$\{a : x = j\} : \{\mathbf{true}\}\ a[j] := 1\ \{\mathbf{true}\}$$

Calculating next

$$(\forall i : i \neq j \rightarrow a[i] = 0) \uparrow x \neq j$$

yields the formula

$$\forall i : i = j \vee (a[i] = 0 \wedge i \neq j).$$

Clearly this latter formula is logically equivalent to $\forall i : i \neq j \rightarrow a[i] = 0$ itself. So we can apply the above RULE 2 which gives us the desired result.

Soundness of the above proof system derives in a straightforward manner from the following lemma (soundness proofs of the remaining rules are standard, see [AdBO09]).

Lemma 2. *(Soundness): Let a be an array variable that does not appear (free) in the formulas of the footprint F. Further, let $\sigma' = \sigma[a[\bar{s}] := u]$ and $\sigma \models p_a(\bar{s})$. It follows that $\sigma \models r \uparrow F$ iff $\sigma' \models r \uparrow F$.*

Proof: By definition of $r \uparrow F$ it suffices to show the above for any literal l. By definition of σ', we have that $\sigma(t) = \sigma'(t)$, for any term t which does not involve the array variable a. So it suffices to show that $\sigma \models l \uparrow F$ or $\sigma' \models l \uparrow F$ implies that $\sigma(a[\bar{t}]) = \sigma'(a[\bar{t}])$, for any subscripted variable $a[\bar{t}]$ appearing as argument of l. By definition of $l \uparrow F$, we have that $l \uparrow F$ implies $\neg p_a(\bar{t})$. Consequently, since $\sigma \models p_a(\bar{s})$, we have that $\sigma(\bar{s}) \neq \sigma(\bar{t})$, which in turn implies that $\sigma(a[\bar{t}]) = \sigma[a[\bar{t}] := u](a[\bar{t}]) = \sigma'(a[\bar{t}])$. □

We have the following extension of the consequence rule.

RULE 3: CONSEQUENCE

$$\frac{p \to p' \quad F : \{p'\}\ S\ \{q'\} \quad q' \to q}{F : \{p\}\ S\ \{q\}}$$

The following two rules deal with recursion. For technical convenience only we restrict to procedure declarations with read-only formal parameters and procedure calls with actual parameters which are not affected by the call. In Rule 5 "$\vdash$" denotes the derivability in the proof system itself. It allows to introduce assumptions about recursive calls (see [AdBO09]).

RULE 4: INSTANTIATION

$$\frac{F : \{p\}\ P(\bar{u})\ \{q\}}{F[\bar{u} := \bar{t}] : \{p[\bar{u} := \bar{t}]\}\ P(\bar{t})\ \{q[\bar{u} := \bar{t}]\}}$$

where $P(\bar{u}) ::= S \in D$ and S does not assign to the variables appearing $\bar{t}$.

RULE 5: RECURSION

$$\frac{\begin{array}{l} F : \{p_1\}\ P_1(\bar{u}_1)\ \{q_1\}, \ldots, F : \{p_n\}\ P_n(\bar{u}_n)\ \{q_n\} \vdash F : \{p\}\ S\ \{q\}, \\ F : \{p_1\}\ P_1(\bar{u}_1)\ \{q_1\}, \ldots, F : \{p_n\}\ P_n(\bar{u}_n)\ \{q_n\} \vdash \\ \quad F : \{p_i\}\ S_i\ \{q_i\},\ i \in \{1, \ldots, n\} \end{array}}{F : \{p\}\ S\ \{q\}}$$

where $P_i(\bar{u}_i) :: S_i \in D$ for $i \in \{1, \ldots, n\}$.

To extend the standard auxiliary rules (as discussed in the previous section) to footprints is straightforward (we omit the similar straightforward extensions of the standard axiom and rules, e.g., the axiom for assignments to simple variables, the conditional rule, the while rule, and the rule for sequential composition).

RULE A5: CONJUNCTION

$$\frac{F : \{p_1\}\ S\ \{q_1\}, F : \{p_2\}\ S\ \{q_2\}}{F : \{p_1 \wedge p_2\}\ S\ \{q_1 \wedge q_2\}}$$

RULE A6: ∃-INTRODUCTION

$$\frac{F : \{p\}\ S\ \{q\}}{F : \{\exists z : p\}\ S\ \{q\}}$$

where z does not occur in F, S or q.

RULE A7: SUBSTITUTION

$$\frac{F : \{p\}\ S\ \{q\}}{F[z := t] : \{p[z := t]\}\ S\ \{q[z := t]\}}$$

where z does not occur in S and S does not change any of the variables in the term t.

The following invariance rule additionally allows to adjust the footprint.

RULE A8: INVARIANCE

$$\frac{(r \wedge F) \rightarrow F' \quad F : \{p\}\ S\ \{q\}}{F' : \{p \wedge r\}\ S\ \{q \wedge r\}}$$

where S does not assign to the variables in the formula r. Further, $(r \wedge F) \rightarrow F'$ holds if for every $a : p'(\bar{x}) \in F$ there exists $a : p''(\bar{x}) \in F'$ such that $(r \wedge p'(\bar{x})) \rightarrow p''(\bar{x})$.

4 Case study: Quicksort

We illustrate the use of footprints in a proof of correctness of the well-known quicksort sorting algorithm [Hoa62,FH71]:

$QS(l, r)$ ::
if $l < r$
then $P(l, r)$;
begin
local $u := m$;
$QS(l, u - 1)$;
$QS(u, r)$
end
fi

Here $P(l, r)$ calls the *partitioning* algorithm which operates on an array a : **integer** $\rightarrow$ **integer** and generates a value for the global integer variable m. The partitioning algorithm satisfies the following contracts.

- $A_1 \equiv x \in [l : r] : \{l < r\}\ P(l, r)\ \{m \in (l : r] \wedge a[l : m - 1] \leq a[m : r]\}$
- $A_2 \equiv x \in [l : r] : \{a = a_0\}\ P(l, r)\ \{perm(a, a_0, l, r)\}$

where $m \in (l : r]$ abbreviates $l < m \wedge m \leq r$ and $a[l : m-1] \leq a[m : r]$ abbreviates $\forall i \in [l : m-1] : \forall j \in [m : r] : a[i] \leq a[j]$. The postcondition of the first contract thus states that the array segment $a[l : r]$ can be split into two segments $a[l : m-1]$ and $a[m : r]$ such that all numbers in $a[l : m-1]$ are smaller or equal to all numbers in $a[m : r]$. The predicate $perm(a, a_0, l, r)$ states that the array a is a permutation of a_0 on the interval $[l : r]$, which can be expressed by the assertion

$$\exists b : \forall i, j \in [l : r] : \exists k \in [l : r] : (i \neq j \rightarrow b[i] \neq b[j]) \wedge b[i] = k \wedge a[i] = a_0[b[i]]$$

where b is an array of type $\mathbf{integer} \rightarrow \mathbf{integer}$. Finally, the footprint $x \in [l : r]$ of the array a states that the array a is only changed on the interval $[l : r]$ (we thus omit for notational convenience the label "a").

Given these contracts we want to prove the following specifications of Quicksort:

- $B_1 \equiv x \in [l : r] : \{\mathbf{true}\}\ QS(l, r)\ \{sorted(a[l : r])\}$
- $B_2 \equiv x \in [l : r] : \{a = a_0\}\ QS(l, r)\ \{perm(a, a_0, l, r)\}$

where $sorted(a[l : r])$ abbreviates the assertion

$$\forall i \in [l : r-1] : a[i] \leq a[i+1]$$

Let S_{QS} denote the body of the procedure QS. By the recursion rule it suffices to prove (assuming the contracts A_1 and A_2)

$$\begin{array}{c} A_1, A_2, B_1, B_2 \vdash \\ x \in [l : r] : \{\mathbf{true}\}\ S_{QS}\ \{sorted(a[l : r])\} \end{array} \tag{1}$$

and

$$\begin{array}{c} A_1, A_2, B_2 \vdash \\ x \in [l : r] : \{a = a_0\}\ S_{QS}\ \{perm(a, a_0, l, r)\} \end{array} \tag{2}$$

Proof of obligation (1) By the conditional rule it suffices to prove that

$$x \in [l : r] : \{l \geq r\}\ skip\ \{sorted(a[l : r])\} \tag{3}$$

and

$$A_1, B_1, B_2 \vdash x \in [l : r] : \{l < r\}\ T\ \{sorted(a[l : r])\} \tag{4}$$

where T denotes the then-branch of S.

The first follows directly from a trivial application of the consequence rule. Using A_1, the (standard) assignment axiom, the block [AdBO09] and sequential composition rule, it is straightforward to establish proof obligation (4) from

$$x \in [l : r] : \begin{array}{c} \{u \in (l : r] \wedge a[l : u-1] \leq a[u : r]\} \\ QS(l, u-1); QS(u, r) \\ \{sorted(a[l : r])\} \end{array} \tag{5}$$

In order to establish this proof obligation, we first instantiate l and r by u and r, respectively, in B_1:

$$x \in [u : r] : \{\mathbf{true}\}\ QS(u, r)\ \{sorted(a[u : r])\}$$

By instantiating l and r by u and r, respectively, in B_2, and the conjunction rule:

$$x \in [u : r] : \{a = a_0\}\ QS(u, r)\ \{sorted(a[u : r]) \wedge perm(a, a_0, u, r)\}$$

It is straightforward to check that

$$\begin{array}{l}(sorted(a[l : u-1]) \wedge a[l : u-1] \leq a_0[u : r]) \restriction x \in [u : r] \leftrightarrow \\ (sorted(a[l : u-1]) \wedge a[l : u-1] \leq a_0[u : r])\end{array}$$

Thus applying the footprint rule 2:

$$x \in [u : r] : \begin{array}{c}\{a = a_0 \wedge sorted(a[l : u-1]) \wedge a[l : u-1] \leq a_0[u : r]\} \\ QS(u, r) \\ \{sorted(a[u : r]) \wedge perm(a, a_0, u, r) \wedge sorted(a[l : u-1]) \wedge a[l : u-1] \leq a_0[u : r]\}\end{array}$$

Next observe that

- $perm(a, a_0, u, r) \wedge a[l : u-1] \leq a_0[u : r]$ implies $a[l : u-1] \leq a[u : r]$, and
- $sorted(a[l : u-1]) \wedge sorted(a[u : r]) \wedge a[l : u-1] \leq a[u : r]$ implies $sorted(a[l : r])$.

Thus by the consequence rule:

$$x \in [u : r] : \begin{array}{c}\{a = a_0 \wedge sorted(a[l : u-1]) \wedge a[l : u-1] \leq a_0[u : r]\} \\ QS(u, r) \\ \{sorted(a[l : r])\}\end{array}$$

Existential elimination (of a_0) and consequence rule:

$$x \in [u : r] : \begin{array}{c}\{sorted(a[l : u-1]) \wedge a[l : u-1] \leq a[u : r]\} \\ QS(u, r) \\ \{sorted(a[l : r])\}\end{array}$$

Invariance rule A8 (adjusting the footprint: $u \in (l : r] \wedge x \in [u : r]$ implies $x \in [l : r]$):

$$x \in [l : r] : \begin{array}{c}\{u \in (l : r] \wedge sorted(a[l : u-1]) \wedge a[l : u-1] \leq a[u : r]\} \\ QS(u, r) \\ \{u \in (l : r] \wedge sorted(a[l : r])\}\end{array}$$

Consequence rule:

$$x \in [l:r]: \begin{array}{c} \{u \in (l:r] \wedge \mathit{sorted}(a[l:u-1]) \wedge a[l:u-1] \leq a[u:r]\} \\ QS(u,r) \\ \{\mathit{sorted}(a[l:r])\} \end{array} \tag{6}$$

Following a similar pattern as above, from assumption B_1 we obtain by instantiation

$$x \in [l:u-1] : \{\mathbf{true}\}\ QS(l,u-1)\ \{\mathit{sorted}(a[l:u-1])\}$$

By instantiating assumption B_2 and the conjunction rule:

$$x \in [l:u-1] : \{a = a_0\}\ QS(l,u-1)\ \{\mathit{sorted}(a[l:u-1]) \wedge \mathit{perm}(a,a_0,l,u-1)\}$$

It is straightforward to check that

$$(a_0[l:u-1] \leq a[u:r]) \uparrow x \in [l:u-1] \leftrightarrow (a_0[l:u-1] \leq a[u:r])$$

Thus applying the footprint rule 2:

$$x \in [l:u-1]: \begin{array}{c} \{a = a_0 \wedge a_0[l:u-1] \leq a[u:r]\} \\ QS(l,u-1) \\ \{\mathit{sorted}(a[l:u-1]) \wedge \mathit{perm}(a,a_0,l,u-1) \wedge a_0[l:u-1] \leq a[u:r]\} \end{array}$$

Since

$$\mathit{perm}(a,a_0,l,u-1) \wedge a_0[l:u-1] \leq a[u:r]$$

implies $a[l:u-1] \leq a[u:r]$, we obtain by the consequence rule:

$$x \in [l:u-1]: \begin{array}{c} \{a = a_0 \wedge a_0[l:u-1] \leq a[u:r]\} \\ QS(l,u-1) \\ \{\mathit{sorted}(a[l:u-1]) \wedge a[l:u-1] \leq a[u:r]\} \end{array}$$

Existential elimination (of a_0) and consequence rule:

$$x \in [l:u-1]: \begin{array}{c} \{a[l:u-1] \leq a[u:r]\} \\ QS(l,u-1) \\ \{\mathit{sorted}(a[l:u-1]) \wedge a[l:u-1] \leq a[u:r]\} \end{array}$$

Invariance rule A8 (adjusting the footprint: $u \in (l:r] \wedge x \in [l:u-1]$ implies $x \in [l:r]$):

$$x \in [l:r]: \begin{array}{c} \{u \in (l:r] \wedge a[l:u-1] \leq a[u:r]\} \\ QS(l,u-1) \\ \{u \in (l:r] \wedge \mathit{sorted}(a[l:u-1]) \wedge a[l:u-1] \leq a[u:r]\} \end{array} \tag{7}$$

Sequential composition applied to the correctness formulas (6) and (7) finally establishes proof obligation (5).

Proof of obligation (2) In this proof we use the next lemma.

Lemma 3. *Suppose z does not occur in S and let p be a binary transitive relation. From $F : \{x = z\}\ S\ \{p(x, z)\}$ we can derive $F : \{p(x, z)\}\ S\ \{p(x, z)\}$.*

Proof: Assume $F : \{x = z\}\ S\ \{p(x, z)\}$. Apply the standard invariance rule, where y is a fresh variable:

$$F : \{x = z \wedge p(z, y)\}\ S\ \{p(x, z) \wedge p(z, y)\}$$

Consequence (rule 3):

$$F : \{x = z \wedge p(z, y)\}\ S\ \{p(x, y)\}$$

Existential elimination (rule A6) and consequence (rule 3):

$$F : \{p(x, y)\}\ S\ \{p(x, y)\}$$

Substitution (rule A7):

$$F : \{p(x, z)\}\ S\ \{p(x, z)\}$$

□

We are now ready to establish proof obligation (2). By the conditional rule it suffices to prove

$$x \in [l : r] : \{l \geq r \wedge a = a_0\}\ \mathit{skip}\ \{perm(a, a_0, l, r)\} \tag{8}$$

and

$$B_2 \vdash x \in [l : r] : \{l < r \wedge a = a_0\}\ T\ \{perm(a, a_0, l, r)\} \tag{9}$$

where T denotes the then-branch of S. The first follows trivially by the consequence rule. To establish proof obligation (9), we obtain from assumption B_2 by instantiation

$$x \in [u : r] : \{a = a_0\}\ QS(u, r)\ \{perm(a, a_0, u, r)\}$$

Note that $perm(a, a_0, l, u-1) \uparrow x \in [u : r]$ is logically equivalent to $perm(a, a_0, l, u-1)$ itself. Thus applying the footprint rule 2:

$$x \in [u : r] :$$
$$\{a = a_0 \wedge perm(a, a_0, l, u - 1)\}\ QS(u, r)\ \{perm(a, a_0, l, u - 1) \wedge perm(a, a_0, u, r)\}$$

Consequence rule:

$$x \in [u : r] : \{a = a_0\}\ QS(u, r)\ \{perm(a, a_0, l, r)\}$$

Note that *perm* is a binary, transitive predicate in the first two arguments. Thus lemma 3 gives:

$$x \in [u : r] : \{perm(a, a_0, l, r)\}\ QS(u, r)\ \{perm(a, a_0, l, r)\}$$

Invariance rule A8 (adjusting the footprint) and consequence rule (weakening the postcondition):

$$x \in [l:r] : \{u \in (l:r] \wedge perm(a, a_0, l, r)\}\ QS(u, r)\ \{perm(a, a_0, l, r)\} \quad (10)$$

As above, we can derive

$$x \in [l : u-1] : \{perm(a, a_0, l, r)\}\ QS(l, u-1)\ \{perm(a, a_0, l, r)\}$$

Invariance rule A8 (adjusting the footprint again):

$$\begin{gathered} x \in [l:r] : \\ \{u \in (l:r] \wedge perm(a, a_0, l, r)\}\ QS(l, u-1)\ \{u \in (l:r] \wedge perm(a, a_0, l, r)\} \end{gathered} \quad (11)$$

Sequential composition (applied to the correctness formulas (10) and (11)):

$$\begin{gathered} x \in [l:r] : \\ \{u \in (l:r] \wedge perm(a, a_0, l, r)\}\ QS(l, u-1); QS(r, u)\ \{perm(a, a_0, l, r)\} \end{gathered}$$

The remainder of the proof follows in a straightforward manner by instantiating A_1 and A_2, and applying the conjunction rule, the block rule (for $u := m$), and the rule for sequential composition.

5 Future Work

First we want to prove (relative) completeness of the Hoare logic extended with footprints. We conjecture that this requires a straightforward extension of the usual Gorelick (relative) completeness proof (see [Apt84]).

It is not difficult to extend the notion of footprints to reasoning about invariant properties of object-oriented programs. The basic idea is simply to include for each *field* f a monadic predicate p_f : **Object** $\rightarrow$ **Boolean** which holds for all those objects which have updated their field f. This extension we want to integrate with our proof theory of *abstract object creation* [AdBG09] which allows specification and verification of dynamic heap structures at an abstraction level that coincides with the Java programming language and which already has been implemented in the KeY theorem prover [BHS07].

References

[AdBG09] Wolfgang Ahrendt, Frank S. de Boer, and Immo Grabe. Abstract object creation in dynamic logic. In *FM 2009: Formal Methods, Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings*, pages 612–627, 2009.

[AdBO09] Krzysztof R. Apt, Frank S. de Boer, and Ernst-Rüdiger Olderog. *Verification of Sequential and Concurrent Programs*. Texts in Computer Science. Springer, 2009.

[Apt84] Krzysztof R. Apt. Ten years of hoare's logic: A survey part II: nondeterminism. *Theor. Comput. Sci.*, 28:83–109, 1984.

[BHS07] Bernhard Beckert, Reiner Hähnle, and Peter H Schmitt. *Verification of object-oriented software: The KeY approach.* Springer-Verlag, 2007.

[BK14] James Brotherston and Max I. Kanovich. Undecidability of propositional separation logic and its neighbours. *J. ACM*, 61(2):14, 2014.

[CYO01] Cristiano Calcagno, Hongseok Yang, and Peter W. O'Hearn. Computability and complexity results for a spatial assertion language for data structures. In Ramesh Hariharan, V. Vinay, and Madhavan Mukund, editors, *FST TCS 2001: Foundations of Software Technology and Theoretical Computer Science*, volume 2245 of *Lecture Notes in Computer Science*, pages 108–119. Springer Berlin Heidelberg, 2001.

[FH71] M. Foley and C. A. R. Hoare. Proof of a recursive program: Quicksort. *Comput. J.*, 14(4):391–395, 1971.

[Hoa62] C. A. R. Hoare. Quicksort. *Comput. J.*, 5(1):10–15, 1962.

[Hoa71] C. A. R. Hoare. Procedures and parameters: An axiomatic approach. In *Symposium on Semantics of Algorithmic Languages*, pages 102–116. 1971.

[Old83] Ernst-Rüdiger Olderog. On the notion of expressiveness and the rule of adaption. *Theor. Comput. Sci.*, 24:337–347, 1983.

[Rey05] John C. Reynolds. An overview of separation logic. In *Verified Software: Theories, Tools, Experiments, First IFIP TC 2/WG 2.3 Conference, VSTTE 2005, Zurich, Switzerland, October 10-13, 2005, Revised Selected Papers and Discussions*, pages 460–469, 2005.

[Wei11] Benjamin Weiß. *Deductive Verification of Object-Oriented Software: Dynamic Frames, Dynamic Logic and Predicate Abstraction.* PhD thesis, Karlsruhe Institute of Technology, 2011.