Using Isabelle/UTP for the Verification of Sorting Algorithms A Case Study

Joshua A Bockenek, Peter Lammich, Yakoub Nemouchi, Burkhart Wolff

Using Isabelle/UTP for the Verification of Sorting Algorithms

A Case Study

Joshua A. Bockenek[1][0000−0002−1055−8003], Peter Lammich[1,2], Yakoub Nemouchi[1], and Burkhart Wolff[3⋆]

[1] The Bradley Department of Electrical and Computer Engineering, Virginia Tech, Blacksburg, VA 24061, USA
jabocken@vt.edu nemouchi@vt.edu lpeter1@vt.edu

[2] Technische Universität München, Institut für Informatik, Boltzmannstr. 3, 85748 Garching, Deutschland
lammich@in.tum.de

[3] University of Paris-Sud (Orthay), 15 Rue Georges Clemenceau, 91400 Orsay, France
wolff@lri.fr

Abstract. We verify functional correctness of insertion sort as well as the partition function of quicksort. We use Isabelle/UTP and its denotational semantics for imperative programs as a verification framework. We propose a forward Hoare VCG for our reasoning and we discuss the different technical challenges encountered while using Isabelle/UTP.

Keywords: Unifying Theories of Programming · HOL · Isabelle · Program Verification · Denotational Semantics.

1 Introduction

In this paper, we are interested in program verification techniques such as those of Floyd [10], Hoare [17], and Dijkstra [8]. More precisely, our work focuses on using Isabelle/UTP [12] for program verification. Isabelle/UTP is a mechanization of Hoare's *Unifying Theories of Programming* (UTP) book [19].

Different reasoning techniques on semantics exist for proving the correctness of programs, such as Hoare logic [18], weakest precondition (WP) calculus [9], and predicate transformers. The basic idea is to prove that a program, in its formal form, implements its requirements, i.e. its *specification*. The semantics of the correctness relation depend on the calculus, e.g., in the case of Hoare logic it is a refinement relation stating that the possible states reached by the execution of the program are less than the states covered by the specification. The specification consists of a *precondition* and a *postcondition*. The precondition is an *assumption*, a predicate on the *initial* state of the program. The postcondition is an *assertion*, a predicate on the *final* state of the program. Other kinds of

⋆ The authors list is sorted alphabetically.

assertions such as loop invariants can be used as *annotations* to carry through additional information useful for proving the correctness relation.

The reasoning calculus, such as Hoare logic, shows the effects of the semantics of the program's constructs on the assertions. Since the calculus is syntax directed, it can be automated, which is the role of a *verification condition generator (VCG)*. A VCG breaks down the proof of the correctness relation to a proof of a set of assertions called *verification conditions (VCs)*. The semantics of a programming language such as C are rich and complex, and verifying the correctness of a program with such rich and complex semantics may lead to a complicated process for the generation and the proving of the VCs. Thus, the usage of an interactive theorem prover (ITP) is mandatory for such development.

In this paper, we applied Hoare logic to UTP programs. The goal of our work is to build, in the future, a scalable toolchain for verifying functional correctness on the system code level. In order to have a scalable verification toolchain, the toolchain must be extensible and provide a usable framework allowing modular and structured reasoning on programs. Extensibility is offered by UTP; however, for structured and modular reasoning, UTP still needs case studies. Our contribution is a case study where we use UTP [19] and its implementation Isabelle/UTP [12] to verify the functional correctness of insertion sort and quicksort represented as UTP programs. We used those algorithms as benchmarks to test the usability and modular reasoning of Isabelle/UTP. The results of this somewhat standard verification exercise turned out to be informative for building, in the future, a functional correctness toolchain for low-level system code based on Isabelle/UTP.

The paper is organized as follows: we describe UTP and Isabelle/UTP in section 2. In section 3, we introduce forward Hoare logic for Isabelle/UTP. We implement a prototype of VCG using Eisbach [25,26]. The VCG uses Hoare rules to compute strongest postconditions (SPs) in the assertion logic of UTP. Our case studies on the sorting algorithms insertion sort and quicksort are presented in section 4. Finally, in section 5, we discuss the results and further steps for building a functional correctness toolchain based on Isabelle/UTP.

Additionally, much of this work is also covered in the Master's thesis of Joshua Bockenek [2], albeit in a form intended for a more general audience. To try out the case studies, download and extract the Isabelle session provided at `https://drive.google.com/open?id=1KbkeHjGCbARj7_3MWDaVXh_RYTh6BxAT`.

2 Background

2.1 UTP

In their UTP book [19], Hoare and He provided a generic framework for defining and working with formal denotational semantics that extends to different programming paradigms. In UTP, a programming language is defined as a *theory*, consisting of the following components:

alphabet The state of the program.

signature The set of operations that manipulate the state.
healthiness conditions Semantic restrictions represented by `H P = P`, where `H` is an idempotent sanitizer function and `P` is the program under consideration.
order A relation on the alphabet used to represent *correctness*.

Assertions in UTP are represented by the theory of *alphabetized predicates*. In a logical deep embedding setting, an alphabetized predicate is simply a pair of alphabet and predicate such that the predicate only uses variables from its associated alphabet.

The *alphabetized relations* theory is built on top of alphabetized predicates. The theory of relations is used to provide a denotational semantics for the different programming constructs. In this setting, the alphabet consists of input variables (the value of a variable in the program's initial state) and output variables (the value of the variable at the program's final state). Two types of alphabetized relations are distinguished: 1. *Heterogeneous relations*, where the input and output alphabets belong to different state spaces. An example of such a relation is a hardware description language (HDL) such as Verilog, where modules can have inputs and outputs with different types. 2. *Homogeneous relations*, where the input and output alphabets belong to the same state space. An example of such a relation is the set of state transitions over the Instruction Set Architecture (ISA) instructions of a microprocessor [4]. In the theory of alphabetized relational calculus, for the case of partial correctness of imperative programs, the healthiness conditions are the identity function. Our work is solely interested in partial correctness for now, meaning the theory of relations suffices for our case study.

In UTP, specifications are represented using the predicate theory. Programs are represented using the relation theory, which is a subset of the predicate theory. This allows for a unified framework where predicates and relations can manipulate the same externally-viewable alphabet. In this way, the correctness relation is represented by an order between specifications and relations stating that a specification is refined by a program. In other words, a specification contains more nondeterminism and covers more final states than the actual program. The formal notation for refinement in UTP is $A \sqsupseteq B$. It can be seen as universal reverse implication on the alphabet $[B \Rightarrow A]$ [5].

2.2 Isabelle/UTP

Among many other mechanizations, we use Foster's Isabelle/UTP [12]. Isabelle/UTP is a mechanization of the UTP book in the ITP Isabelle [27]. As our long-term goal is to build a verification toolchain for imperative code, we chose to use Foster's Isabelle/UTP for the following reasons: 1. In general, UTP is an extensible framework; it is not specific to one programming language nor to one programming paradigm. 2. Foster's UTP mechanization uses Isabelle and its powerful infrastructure allowing for parallel proof checking, tactic customization, sophisticated method implementation via Eisbach [25,26], and strong back-end tools such as sledgehammer [3,28]. 3. Foster's shallow embedding for UTP reduces the work required for proof automation at the expense of metatheoretic

```
typedef ('t, 'α) uexpr = <UNIV :: ('α ⇒ 't) set> ..
type_synonym 'α upred = <(bool, 'α) uexpr>
translations
  (type) <'α upred> <= (type) "(bool, 'α) uexpr"
type_synonym 'α cond = <'α upred>
type_synonym ('α, 'β) rel = <('α × 'β) upred>
type_synonym 'α hrel = <('α × 'α) upred>
type_synonym ('a, 'α) hexpr = <('a, 'α × 'α) uexpr>
translations
  (type) <('α, 'β) rel> <= (type) <('α × 'β) upred>
lift_definition lit :: <'t ⇒ ('t, 'α) uexpr> is <λ v b. v> .
lift_definition uop :: <('a ⇒ 'b) ⇒ ('a, 'α) uexpr ⇒ ('b, 'α) uexpr>
    is
  <λ f e b. f (e b)> .
lift_definition bop :: <('a ⇒ 'b ⇒ 'c) ⇒ ('a, 'α) uexpr ⇒ ('b, 'α)
    uexpr ⇒ ('c, 'α) uexpr> is
  <λ f u v b. f (u b) (v b)> .
(...)
```

Listing 1.1: Expression language in Isabelle/UTP

manipulation; 4. and lenses [11,13], which provide a general abstraction of views into data; that is, variables are represented as lenses whose state space is left completely polymorphic. Thus, pointwise algebraic laws of programming [18] can be expressed and a generic framework for memory models can be instantiated for different programming languages.

The expressions model as well as the predicate and relational model are represented by Foster as in listing 1.1. On top of this expression language, the predicate theory, i.e. the assertion logic, is defined as in listing 1.2. The type ('a ⟹ 'α) denotes a lens with state type 'α and view type 'a.

```
lift_definition impl :: <'α upred ⇒ 'α upred ⇒ 'α upred> is
  <λ P Q A. P A ⟶ Q A> .

lift_definition iff_upred :: <'α upred ⇒ 'α upred ⇒ 'α upred> is
  <λ P Q A. P A ⟷ Q A> .

lift_definition ex :: <('a ⟹ 'α) ⇒ 'α upred ⇒ 'α upred> is
  <λ x P b. (∃ v. P (put x b v))> .

lift_definition shEx :: <['β ⇒' α upred] ⇒ 'α upred> is
  <λ P A. ∃ x. (P x) A> .

lift_definition all :: <('a ⟹ 'α) ⇒ 'α upred ⇒ 'α upred> is
  <λ x P b. (∀ v. P (put x b v))> .

lift_definition shAll :: <['β ⇒' α upred] ⇒ 'α upred> is
```

```
17  ‹λ P A. ∀ x. (P x) A› .
```

Listing 1.2: Assertion logic in Isabelle/UTP

On top of the predicate calculus, a denotational semantics for a nondeterministic imperative programming language is defined as in listing 1.3. In this notation, $\lceil \texttt{cond} \rceil_<$ is a lifting of `cond` from predicates to relations. The lifting is done using $\lceil \texttt{cond} \rceil_< \equiv \texttt{cond} \oplus_\texttt{p} \texttt{fst}_\texttt{L}$ such that $\oplus_\texttt{p}$ is defined as `λ P x b. P (get x b)`, $\texttt{fst}_\texttt{L}$ is a lens with the standard projection function `fst` as a lens `get` function, and `(λ (σ, ϱ) u. (σ, u))` as a lens `put` function.

The developers of Isabelle/UTP provided many syntax translations to mimic the notation used in the UTP book. This occasionally causes problems with proof reconstruction in sledgehammer and ambiguous parse warnings or errors due to heavy usage of ad-hoc overloading, but is useful for those who want their work to look more like the UTP book. Figure 1 shows some of the equivalences between UTP and higher-order logic (HOL) features, while some additional features are described below.

«v» An HOL literal; depending on the context, explicit usage may not be necessary (such as with numeric literals) but is usually required when reasoning with logical variables.

$[\![P]\!]_\texttt{e}\ (s, t)$ A statement that input state s and output state t are consistent for program/predicate P.

$P[\![u/x]\!]$ A substitution of the (input) expression u for accesses of lens x in program/predicate P; this notation can also be used for input ($\$x$) and output ($\$x'$) substitutions, in which case u must be lifted with $\lceil u \rceil_<$ and $\lceil u \rceil_>$ (previously seen in listing 1.4), respectively.

$\$\Sigma$ The entire input alphabet (i.e. the whole state) as a lens; as usual, the corresponding output lens is $\$\Sigma'$.

$\&x$ Gets value as expression from lens.

`II` The null operation, a no-op, `SKIP`; used when a statement is required but a state change is undesired.

`:==` Basic assignment of an expression to a lens.

`;;` Sequential composition; this is used to compose statements.

$P \lhd b \rhd Q$ The conditional; $\texttt{if}_\texttt{u}$ `b then P else Q` is the wrapper notation we use for a nicer look.

$\mu X \bullet P$ Least fixed point (LFP) recursion of program P with X occurring in P and representing the point where the recursion is "unfolded", so to speak.

$\nu X \bullet P$ Greatest fixed point (GFP) recursion, otherwise same as the above.

Iteration Achieved with a wrapper around the recursion construct, either LFP or GFP depending on total or partial correctness, with notation akin to `while b do P od` where `P` is repeatedly executed until `b` becomes false.

3 Forward reasoning in Isabelle/UTP

The UTP book as well as the initial version of Isabelle/UTP both utilize Dijkstra's WP, or backwards, reasoning [9]. In our work we use SP, or forward, reasoning [14].

```
abbreviation cond ::
  «('a,'α) uexpr ⇒ 'α upred ⇒ ('a,'α) uexpr ⇒ ('a,'α) uexpr»
  ("(3_ ◁ _ ▷/ _)" [52,0,53] 52) where
    «P ◁ b ▷ Q ≡ trop If b P Q»
abbreviation rcond ::
  «('α, 'β) rel ⇒ 'α cond ⇒ ('α, 'β) rel ⇒ ('α, 'β) rel»
  ("(3_ ◁ _ ▷ᵣ/ _)" [52,0,53] 52) where
«(P ◁ b ▷ᵣ Q) ≡ (P ◁ ⌈b⌋< ▷ Q)»

lift_definition seqr :: «('α, 'β) rel ⇒ ('β, 'γ) rel ⇒ ('α, 'γ) rel»
    is
  «λ P Q r. r ∈ ({p. P p} O {q. Q q})» . - «Heterogeneous Sequential
    composition»

abbreviation seqh :: «'α hrel ⇒ 'α hrel ⇒ 'α hrel» (infixr ";;ₕ" 71)
    where
  «seqh P Q ≡ (P ;; Q)» - «Homogeneous Sequential composition»

lift_definition assigns_r :: «'α usubst ⇒ 'α hrel» is
  «λ σ (A, A'). A' = σ(A)» . - «assignement»

definition skip_r :: «'α hrel» where
  [urel_defs]: «skip_r = assigns_r id» - «No op»

definition rassume :: «'α upred ⇒ 'α hrel» ("_⊤" [999] 999) where
  [urel_defs]: «rassume c = II ◁ c ▷ᵣ false» - «assumption»

definition rassert :: «'α upred ⇒ 'α hrel» ("_⊥" [999] 999) where
  [urel_defs]: «rassert c = II ◁ c ▷ᵣ true» - «assertion»

abbreviation truer :: «'α hrel» ("trueₕ") where
  «truer ≡ true» - «the abort program»

abbreviation falser :: «'α hrel» ("falseₕ") where
  «falser ≡ false» - «the miracle program»

definition
  «(while⊤ b do body od) = (ν X • bif b then (body ;; X) else SKIPᵣ eif)»

definition
 «(while⊥ b do body od) = (μ X • bif b then (body ;; X) else SKIPᵣ eif)»
```

Listing 1.3: Programming constructs as relations in Isabelle/UTP

`length xs`	$\#_u(xs)$
`card s`	$\#_u(s)$
`pcard p`	$\#_u(p)$
`[1, 2, 3]`	$\langle 1, 2, 3 \rangle$
`xs @ ys`	$xs \mathbin{^\frown_u} ys$
`xs ! i`	$xs(i)_a$
`f x`	$f(x)_a$
`x :: 'a`	$x :_u 'a$
`fst t`	$\pi_1(t)$
`snd t`	$\pi_2(t)$
`xs[i := x]`	$xs(i \mapsto x)_u$
`set xs`	$ran_u(xs)$
`λx. p`	$\lambda\ x \bullet p$
$\longrightarrow$	$\Rightarrow$
(a) Isabelle/HOL	(b) Isabelle/UTP

Fig. 1: Some Isabelle/UTP Syntax Comparisons

```
1 definition hoare_r :: ‹'α cond ⇒ 'α hrel ⇒ 'α cond ⇒ bool› ("⦃_⦄_⦃_⦄ᵤ
      ") where
2   ‹⦃p⦄Q⦃r⦄ᵤ = ((⌈p⌉< ⇒ ⌈r⌉>) ⊑ Q)›
```

Listing 1.4: Hoare Logic

In forward reasoning, we fix the precondition and the body of the program under verification, and then we compute, in a syntax directed way, the strongest postcondition using Hoare rules. A Hoare triple in Isabelle/UTP is a refinement relation between specification and program defined as in listing 1.4. On this basis we defined Hoare rules for forward reasoning, shown in listing 1.5. For the most part, we used the same Hoare rules that Isabelle/UTP supplies, only adding those additional rules needed for forward reasoning. On line 1 of listing 1.5, `p` is a UTP predicate, `x` is a lens, `e` is a UTP expression, and `v` is an HOL logical variable lifted to a UTP expression using `«v»`. The notation `e⟦«v»/x⟧` means that `«v»` is substituted for `x` in `e`. This rule is referred to as the Floyd assignment rule [14] after Robert W. Floyd, and it is easy to express in our setting as Isabelle/UTP has a substitution theory[4]. Namely, reasoning on assignment boils down to reasoning on substitutions. More details on the substitution theory in use can be found in [13].

Lines 6 and 15 in listing 1.5 are the conditional rules we used for forward reasoning. Note that both conditional rules require an annotation for branches in order to avoid unnecessary reasoning about two different postconditions, thereby preventing exponential subgoal growth by doing a merge immediately after the application of the rule. We have two rules for conditional statements as separate

[4] https://github.com/isabelle-utp/utp-main/blob/master/utp/utp_subst.thy

```
1  lemma assigns_floyd_r [hoare]:
2    assumes <vwb_lens x>
3    shows   <{|p|}x :== e{|∃ v • p[|«v»/x|] ∧ &x =ᵤ e[|«v»/x|]|}ᵤ>
4    proof <>
5
6  lemma cond_assert_hoare_r[hoare_rules]:
7    assumes <{|b ∧ p|}C₁{|q|}ᵤ>
8        and <{|¬b ∧ p|}C₂{|s|}ᵤ>
9        and <‘q ⇒ A‘>
10       and <‘s ⇒ A‘>
11       and <{|A|}P{|A’|}ᵤ>
12     shows <{|p|}(ifᵤ b then C₁ else C₂);; A⊥;; P{|A’|}ᵤ>
13   proof <>
14
15 lemma cond_assert_last_hoare_r[hoare_rules]:
16   assumes <{|b ∧ p|}C₁{|q|}ᵤ>
17       and <{|¬b ∧ p|}C₂{|s|}ᵤ>
18       and <‘q ⇒ A‘>
19       and <‘s ⇒ A‘>
20   shows <{|p|}(ifᵤ b then C₁ else C₂);; A⊥{|A|}ᵤ>
21   proof <>
22
23 lemma while_invr_hoare_r’[hoare_rules]:
24   assumes <‘p ⇒ i‘> and <{|i ∧ b|}C{|q|}ᵤ> and <‘q ⇒ i‘>
25   shows <{|p|}while b invr i do C od{|¬b ∧ i|}ᵤ>
26   by (metis while_inv_def assms hoare_post_weak hoare_pre_str
       while_hoare_r)
```

Listing 1.5: Forward Hoare rules

handling is required for conditionals at the end of a series of statements versus conditionals that have more statements following.

For iterations, we used the while rule on line 23 in listing 1.5. That rule asserts that the invariant `i` holds after completion of the loop, as well as requiring that the precondition `p` implies `i`, there is some postcondition `q` that holds after each iteration of the loop given `i` (though `q` does not necessarily hold after the loop terminates), and `q` implies `i`. Reading the while rule leads to drawing the control flow graph of the while statement!

While applying the forward Hoare rules, and in order to avoid the VCG running into infinite loops, our first solution was to enforce a given order for premises in the different rules. This allowed an automatic execution while generating the VCs; however, in Dr. Lammich's course materials[5], he used a better solution where his VCG automatically defers a goal as soon as a Hoare rule cannot be applied on that goal.

[5] https://bitbucket.org/plammich/certprog_public/src/4698b3509065369693ed144b9bb99ba8b632440b/Project/IMPlusPlus.thy

```
method exp_vcg_pre = (simp only: seqr_assoc[symmetric])?, rule
    hoare_post_weak
method solve_dests = safe?; simp?; drule vcg_dests; assumption?; (simp
    add: vcg_simps)?
method solve_vcg = assumption|pred_simp?, (simp add: vcg_simps)?;(
    solve_dests; fail)?
method vcg_hoare_rule = rule hoare_rules_extra|rule hoare_rules
method exp_vcg_step = vcg_hoare_rule|solve_vcg; fail
method exp_vcg = exp_vcg_pre, exp_vcg_step+
```

Listing 1.6: VCG Methods

As a last note, the additional quantifiers added by the forward assignment rule could decrease performance if used with proofs involving satisfiability modulo theories (SMT) solvers [22] and possibly with other proof methods as well. A solution for dealing with nested existential quantifiers can be found in [14], but for our work, we have judged that is not necessarily an issue as the pseudocode of the algorithms in the case study is quite small.

3.1 VCG

While tactic implementation in Isabelle is typically done on the Meta Language (ML) level, in recent years the introduction of Eisbach [25, 26] has allowed development of advanced proof methods on the Isar [31] level. Though not necessarily as efficient or in-depth as ML-level functionality, usage of Eisbach allows faster and easier development of complex methods, and thus we primarily utilized Eisbach for the initial work on the VCG. The basic VCG methods we developed are shown in listing 1.6, with the main entry point being `exp_vcg`. The method `exp_vcg` starts by transforming the goal state into a suitable form for forward verification and then repeatedly applies the VCG's step method. The `exp_vcg_step` method attempts to apply one of a set of Hoare rules (from the `hoare_rules_extra` or `hoare_rules` lemma sets) to break down the current statement to a VC. If that fails, it applies `solve_vcg` to discharge the current subgoal. Discharging the goal may be simple if a trivial subgoal was introduced or may be more complicated, requiring either simplification using a developed library of simplification rules (`vcg_simps`) or application of destruction rules (`vcg_dests`) followed by additional simplifications. The VCG stops when it reaches a subgoal it cannot discharge, at which point the user must develop additional lemmas to add to `vcg_simps` or `vcg_dests`.

4 Case Studies

As relatively simple but non-trivial algorithms, standard imperative sorts seemed useful as basic case studies for testing modular reasoning in Isabelle/UTP. We

focused on two sorts, insertion sort and quicksort; while we proved the full algorithm for the former, the latter's proof was limited to only a component of the algorithm due to the limitations of Isabelle/UTP at the time. For both case studies, we focused on partial rather than total correctness as we did not have a working version of the theory of designs [32] or a proper implementation of measure function handling at the time.

4.1 Insertion sort

A simple but relatively efficient algorithm for small data sets, insertion sort [24] served as a useful case for testing the modular handling of nested programs in Isabelle/UTP. Insertion sort contains two nested programs such that: the outer program does sorting and the inner program does insertion. We represent the latters by a basic procedure that involves an inner and outer loop with the outer loop maintaining the portion of the list/array that is sorted (which increases by one on each iteration) while the inner loop moves the next element to be sorted into its sorted position. This continues until the sorted portion of the list encompasses the entire list.

Proof Setup The insertion sort algorithm does not require any particular preconditions other than some basic lens property assumptions and type constraints (which are satisfied by type restrictions of the syntax here); for the postcondition, we have two properties:

1. The list is sorted afterwards.
2. The contents of the list remain the same.

Strictly speaking, insertion sort also provides *stability* (items that compare equal given a specific sort key function maintain the same relative order in the sorted list), but that was not covered in our proof as we only used the notion of order to restrict our types.

We used an auxiliary variable to ensure the contents of the list did not change as the list was sorted, and the `swap_at`$_{\mathtt{u}}$[6] function on line 14 produces a new list with the elements at `&j` and `&j - 1` swapped.

Invariants Each loop in the insertion sort algorithm requires an invariant, as you can see on lines 10 and 13. For modularity and ease of proving, the invariants were extracted and represented as Isabelle/HOL definitions lifted to Isabelle/UTP expressions (shown in listings 1.8 and 1.9). This allowed us to formulate lemmas about the invariant interactions strictly on the HOL level.

[6] For our purposes, usage of $_{\mathtt{u}}$ with functions like `swap_at` on line 14 indicates an HOL function that has been lifted to UTP and is thus usable with Isabelle/UTP expressions.

```
lemma insertion_sort:
  assumes ‹lens_indep_all [i, j]›
      and ‹vwb_lens array› and ‹array ♯ old_array›
      and ‹i ⋈ array› and ‹i ♯ old_array›
      and ‹j ⋈ array› and ‹j ♯ old_array›
  shows
  ‹⦇&array =u old_array⦈
  i :== 1;;
  while &i <u #u(&array)
  invr outer_invru (&i) (&array) old_array do
    j :== &i;;
    (while &j >u 0 ∧ &array(&j - 1)a >u &array(&j)a
    invr inner_invru (&i) (&j) (&array) old_array do
      array :== swap_atu (&j) (&array);;
      j :== (&j - 1)
    od);;
    i :== (&i + 1)
  od
  ⦇msetu(&array) =u msetu(old_array) ∧ sortedu(&array)⦈u›
  by (insert assms) exp_vcg
```

Listing 1.7: Proof of Insertion Sort Correctness

```
definition ‹outer_invr i array old_array ≡
  mset array = mset old_array
∧ sorted (take i array) (* everything up to i-1 is sorted *)
›
abbreviation ‹outer_invru ≡ trop outer_invr›
```

Listing 1.8: Insertion Sort Outer Invariant

```
definition ‹inner_invr i j array old_array ≡
  i < length array
∧ i ≥ j
∧ mset array = mset old_array
∧ (let xs1 = take j array; x = array!j; xs2 = drop (Suc j) (take (Suc i)
    array)
   in sorted (xs1 @ xs2) ∧ (∀y ∈ set xs2. x < y))
›
abbreviation ‹inner_invru ≡ qtop inner_invr›
```

Listing 1.9: Insertion Sort Inner Invariant

The lemmas required to satisfy the verification conditions generated for the outer loop are shown in listings 1.10 to 1.12. For the inner loop, the initial condition was somewhat trivial (listing 1.13), but because we did not develop a good library of simplification lemmas for expressions involving `swap_at`, the step condition needed a large proof (listing 1.14). There was no need for a lemma for the final state of the inner loop as that was already handled by listing 1.11.

```
lemma outer_invr_init[vcg_simps]:
  assumes ‹mset array = mset old_array›
  shows ‹outer_invr (Suc 0) array old_array›
  using assms unfolding outer_invr_def
  by (metis sorted_single sorted_take take_0 take_Suc)
```

Listing 1.10: Insertion Sort Outer Invariant Initial Condition

```
lemma outer_invr_step[vcg_simps]:
  assumes ‹inner_invr i j array old_array›
      and ‹j = 0 ∨ ¬ array ! j < array ! (j - Suc 0)›
  shows ‹outer_invr (Suc i) array old_array›
  proof <>
```

Listing 1.11: Insertion Sort Outer Invariant Step Condition

```
lemma outer_invr_final[vcg_dests]:
  assumes ‹outer_invr i array old_array›
      and ‹¬ i < length array›
  shows ‹mset array = mset old_array›
    and ‹sorted array›
  using assms unfolding outer_invr_def
  by auto
```

Listing 1.12: Insertion Sort Outer Invariant Final Condition

```
lemma inner_invr_init[vcg_simps]:
  assumes ‹outer_invr i array old_array›
      and ‹j = i›
      and ‹i < length array›
    shows ‹inner_invr i j array old_array›
  using assms unfolding outer_invr_def inner_invr_def
  by auto
```

Listing 1.13: Insertion Sort Inner Invariant Initial Condition

```
lemma inner_invr_step[vcg_simps]:
  assumes ‹inner_invr i j array old_array›
      and ‹j > 0›
      and ‹array!(j - Suc 0) > array!j›
  shows ‹inner_invr i (j - Suc 0) (swap_at j array) old_array›
  proof <>
```

Listing 1.14: Insertion Sort Inner Invariant Step Condition

4.2 Quicksort

Coincidentally developed by Hoare [16], quicksort is a reasonably efficient recursive algorithm even for large data sets. As with insertion sort, it is an in-place sort, but unlike insertion sort, it is not stable. We are interested in quicksort since it allows testing the handling of recursive calls in Isabelle/UTP.

The algorithm basically works by choosing a value in the portion of the list under consideration as a "pivot", rearranging that portion of the list until everything in it less than or equal to the pivot is below the pivot and everything greater than or equal to the pivot is above the pivot, and then recursing on the regions now above and below the pivot until the list is sorted. There are multiple possible partitioning schemes usable for quicksort with varying levels of efficiency; in our case, we used a Lomuto-style methodology [1,7]. Lomuto-style is less efficient than Hoare's original formulation, but it was easier to reason about and prove.

As we did not have a fully developed rule and VCG procedure for working with generalized recursion and handling functions with local variables at the time this case study was done, the quicksort proof was restricted to the partitioning component of the algorithm, the our work representation which is shown in listing 1.15.

Proof Setup Because `lo` and `hi` are never assigned to, they do not need to be lenses and can simply be UTP expressions, but they still need some simple preconditions to ensure correctness (`lo` is less than `hi` and `hi` is less than the length of the list to sort, plus the various assumptions for lenses). For the postconditions, the proof must show that, after partitioning:

- Everything below the pivot in the slice of the list operated on is less than or equal to the pivot.
- Everything above it in the slice of the list operated on is greater than or equal to the pivot.
- The contents of the slice are the same.
- The rest of the list is not modified by the partitioning.

Invariant While the partition invariant in listing 1.16 is longer than either of the insertion sort invariants, most of the reasoning is either easy to intuit (maintaining orders between variables) or is just a repeat of postcondition requirements. The important bits in terms of the loop behavior are lines 9 and 10; with these lines, the invariant establishes that, during each iteration, everything less than `i` in the list slice is less than or equal to the pivot and everything from `i` to `j-1` (the upper value of `slice` is exclusive) is greater than or equal to the pivot. The simplification lemmas in listings 1.17, 1.18, 1.20, and 1.22 were developed to resolve the generated verification conditions for the quicksort partition loop, and note in particular listings 1.18 and 1.20; two separate step lemmas were necessary for the two branches of the conditional (one when the next element is found to be less than the pivot, the other when it is not). the `if`-statement

```
lemma quicksort_partition:
  fixes pivot :: <_::linorder ⟹ _>
  assumes <lens_indep_all [i, j]>
    and <vwb_lens pivot> and <vwb_lens A>
    and <pivot ⋈ i> and <pivot ⋈ j>
    and <A ♯ oldA> and <A ♯ lo> and <A ♯ hi>
    and <i ⋈ A> and <i ♯ oldA> and <i ♯ lo> and <i ♯ hi>
    and <j ⋈ A> and <j ♯ oldA> and <j ♯ lo> and <j ♯ hi>
    and <pivot ⋈ A> and <pivot ♯ oldA> and <pivot ♯ lo> and <pivot ♯ hi>
  shows
  <{&A =u oldA
∧ lo <u hi
∧ hi <u #u(&A)}
  pivot :== &A(hi)a;;
  i :== lo;;
  j :== lo;;
  (while &j <u hi invr qs_partition_invru (&A) oldA lo hi (&i) (&j) (&
    pivot) do
  (ifu &A(&j)a <u &pivot then
    A :== swapu (&i) (&j) (&A);;
    i :== (&i + 1)
  else II);;
  (qs_partition_invru (&A) oldA lo hi (&i) (&j + 1) (&pivot))⊥;;
  j :== (&j + 1)
  od);;
  A :== swapu (&i) hi (&A)
  {msetu(sliceu lo (hi + 1) (&A)) =u msetu(sliceu lo (hi + 1) oldA)
∧ takeu(lo, &A) =u takeu(lo, oldA)
∧ dropu(hi + 1, &A) =u dropu(hi + 1, oldA)
∧ pivot_invru (&i - lo) (sliceu lo (hi + 1) (&A))}u>
  by (insert assms) exp_vcg
```

Listing 1.15: Proof of Quicksort Partition Correctness

```
definition ‹qs_partition_invr A oldA lo hi i j pivot ≡
  mset (slice lo (Suc hi) A) = mset (slice lo (Suc hi) oldA)
∧ take lo A = take lo oldA
∧ drop (Suc hi) A = drop (Suc hi) oldA
∧ lo ≤ i
∧ i ≤ j
∧ j ≤ hi
∧ hi < length A
∧ (∀x ∈ set (slice lo i A). x ≤ pivot)
∧ (∀x ∈ set (slice i j A). pivot ≤ x)
∧ pivot = A!hi
›
abbreviation ‹qs_partition_invr_u ≡ sepop qs_partition_invr›
```

Listing 1.16: Quicksort Partition Invariant

```
lemma qs_partition_invr_init[vcg_simps]:
  assumes ‹A = oldA›
      and ‹lo < hi›
      and ‹hi < length A›
  shows ‹qs_partition_invr A oldA lo hi lo lo (A!hi)›
  using assms unfolding qs_partition_invr_def pivot_invr_def slice_def
  by (smt drop_all empty_iff length_take less_imp_le less_trans list.set
    (1) min.absorb2 order_refl)
```

Listing 1.17: Quicksort Partition Invariant Initial Condition

annotation mentioned in section 3 can be seen on line 22, explicitly stating that both branches must preserve the invariant.

Unlike for insertion sort, a set of helper lemmas were developed for the simplification of expressions involving `swap` and `slice`, which resulted in cleaner proofs for the invariant-related lemmas. Other subproofs were also extracted and proved separately (listings 1.19 and 1.21).

5 General conclusion

5.1 Related work

Many works attempt to provide a formal framework for verifying functional correctness of imperative code. In [30], a generic formal imperative language called Simpl was introduced together with a sophisticated definition for Hoare triples that deals with behaviors such as abrupt termination and mutually recursive functions. A VCG was implemented on top, but modular reasoning was not discussed. The language was successfully used in the L4.verified project [23], which introduced a highly automated toolchain for verification of the secure

```
lemma qs_partition_invr_step1[vcg_simps]:
  fixes A :: <_::order list>
  assumes <qs_partition_invr A oldA lo hi i j pivot>
      and <j < hi>
      and <A!j < pivot> -- <version requiring swap and i increment>
  shows <qs_partition_invr (swap i j A) oldA lo hi (Suc i) (Suc j) pivot>
  proof <>
```

Listing 1.18: Quicksort Partition Invariant First Step Condition

```
lemma qs_partition_invr_step2_helper:
  fixes A :: <_::order list>
  assumes <∀x ∈ set (slice i j A). p ≤ x>
      and <p ≤ A!j>
      and <j < length A>
  shows <∀x ∈ set (slice i (Suc j) A). p ≤ x>
  using assms
  by (cases <i ≤ j>) (auto simp: slice_suc2_eq)
```

Listing 1.19: Quicksort Partition Invariant Step 2 Helper

```
lemma qs_partition_invr_step2[vcg_simps]:
  fixes A :: <_::linorder list> -- <Can't do everything with partial
    ordering.>
  assumes <qs_partition_invr A oldA lo hi i j pivot>
    and <j < hi>
    and <¬ A!j < pivot> -- <so array doesn't change this step>
  shows <qs_partition_invr A oldA lo hi i (Suc j) pivot>
  using assms unfolding qs_partition_invr_def pivot_invr_def
  using qs_partition_invr_step2_helper
 by (auto simp: slice_suc2_eq)
```

Listing 1.20: Quicksort Partition Invariant Second Step Condition

```
lemma pivot_slice_swap:
  fixes xs :: <_::order list>
  assumes <lo ≤ i>
    and <i ≤ hi>
    and <hi < length xs>
    and <∀x ∈ set (slice lo i xs). x ≤ xs!hi>
    and <∀x ∈ set (slice i hi xs). xs!hi ≤ x>
  shows <pivot_invr (i - lo) (slice lo (Suc hi) (swap i hi xs))>
  using assms unfolding pivot_invr_def
  by (auto simp: min.absorb1) (meson assms(4) order_trans
    qs_partition_invr_step2_helper)
```

Listing 1.21: Pivot-Slice-Swap Helper

```
lemma qs_partition_invr_final[vcg_simps]:
  fixes A :: <_::order list>
  assumes <qs_partition_invr A oldA lo hi i j pivot>
    and <¬ j < hi>
  shows <mset (slice lo (Suc hi) (swap i hi A)) = mset (slice lo (Suc hi)
     oldA)>
  and <pivot_invr (i - lo) (slice lo (Suc hi) (swap i hi A))>
  and <drop (Suc hi) (swap i hi A) = drop (Suc hi) oldA>
  and <take lo (swap i hi A) = take lo oldA>
  using assms unfolding qs_partition_invr_def
  by (auto simp: pivot_slice_swap)
```

Listing 1.22: Quicksort Partition Invariant Final Condition

embedded L4 (seL4) microkernel. Another attempt at providing a verification framework for imperative code is [15]. The tool, called AutoCorres and taking Simpl as input, has now been integrated into the L4v toolchain. A refinement approach was proposed where Simpl programs are abstracted to a functional, shallowly-embedded monadic form [6]. AutoCorres is equipped with a VCG for computing weakest precondition as well as a debugging mode that traces the applied rules. As with Simpl's VCG, AutoCorres does not provide modular reasoning.

5.2 Discussion on scalability

A framework that does not scale is infeasible for use with large programs such as compilers or even operating systems. Structuring proofs modularly (say, on the function level) such that the proofs can be reused whenever the corresponding code is used in a larger program is one way to handle issues of scalability, but can be difficult to deal with when using a setup that requires statements about memory to be carried through preconditions and postconditions even when those statements are not necessary within the proof.

One methodology designed to minimize this issue is *separation logic* [29], which adds a separation operator to Hoare logic that allows stating various variables and regions of memory are disjunct. A different but similar methodology is to use the concept of *framing* [20,21]. In this style, one lists the variables or regions of memory that are modified by a certain statement or set of statements and all others are considered unaffected by the verification framework. As Isabelle/UTP does provide frame and antiframe rules, albeit for WP reasoning, we intended to use this methodology to handle modular memory reasoning, but we did not develop proper SP rules in time.

5.3 Conclusion

The case studies in section 4 show that our fledgling methodology, performing VC-based formal proofs of correctness of a C-style language using SP methodology,

can indeed be used in Isabelle/UTP. One observation that was made from those case studies was that isolating the proofs of individual VCs to work purely on the HOL level makes for cleaner proofs in general, particularly when good libraries of helper lemmas are developed to ease simplification. However, the work on the quicksort case study in particular highlighted some of the features necessary for further work that we currently lack, such as proper handling of (recursive) function calls and memory modularization in general. Total correctness is also a goal, for which more recent versions of the framework under development have added the usage of measures/variants.

Acknowledgments

This work is supported in part by the Office of Naval Research (ONR) under grants N00014-17-1-2297 and N00014-16-1-2818, and the Naval Sea Systems Command (NAVSEA)/the Naval Engineering Education Consortium (NEEC) under grant N00174-16-C-0018. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of ONR or NAVSEA/NEEC.

References

1. Bentley, J.: Programming Pearls. ACM, New York, NY, USA (1986)
2. Bockenek, J.A.: USIMPL: An Extension of Isabelle/UTP with Simpl-like Control Flow. Master's thesis, Virginia Tech (Dec 2017)
3. Böhme, S., Nipkow, T.: Sledgehammer: Judgement day. In: Giesl, J., Hähnle, R. (eds.) Automated Reasoning. pp. 107–121. Springer-Verlag, Berlin, Heidelberg (2010)
4. Brucker, A.D., Feliachi, A., Nemouchi, Y., Wolff, B.: Test program generation for a microprocessor: A case-study. In: Veanes, M., Viganò, L. (eds.) Tests and Proofs - 7th International Conference, TAP 2013, Budapest, Hungary, June 16-20, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7942, pp. 76–95. Springer-Verlag (2013). https://doi.org/10.1007/978-3-642-38916-0_5
5. Cavalcanti, A., Woodcock, J.: A tutorial introduction to CSP in Unifying Theories of Programming. In: Cavalcanti, A., Sampaio, A., Woodcock, J. (eds.) Refinement Techniques in Software Engineering: First Pernambuco Summer School on Software Engineering, PSSE 2004, Recife, Brazil, November 23-December 5, 2004 Revised Lectures, Lecture Notes in Computer Science, vol. 3167, pp. 220–268. Springer-Verlag, Berlin, Heidelberg (2006). https://doi.org/10.1007/11889229_6
6. Cock, D., Klein, G., Sewell, T.: Secure microkernels, state monads and scalable refinement. In: Theorem Proving in Higher Order Logics: 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008, Proceedings. Lecture Notes in Computer Science, vol. 5170, pp. 167–182. Springer-Verlag (2008)
7. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, chap. Quicksort, pp. 170–190. MIT Press and McGraw-Hill, Cambridge, MA, USA, third edn. (2009)

8. Dijkstra, E.W.: Notes on structured programming. In: Dahl, O.J., Dijkstra, E.W., Hoare, C.A.R. (eds.) Structured Programming, chap. I, pp. 1–82. Academic Press Ltd., London, UK, UK (1972), http://dl.acm.org/citation.cfm?id=1243380.1243381
9. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall Series in Automatic Computation, Prentice-Hall, Englewood Cliffs, NJ (Oct 1976)
10. Floyd, R.W.: Assigning meanings to programs. In: Proceedings of Symposia in Applied Mathematics. vol. 19, pp. 19–32. American Mathematical Society, Providence, Rhode Island (1967)
11. Foster, S., Zeyda, F.: Optics. Archive of Formal Proofs (May 2017), http://isa-afp.org/entries/Optics.shtml, Formal proof development
12. Foster, S., Zeyda, F., Woodcock, J.: Isabelle/UTP: A mechanised theory engineering framework. In: Naumann, D. (ed.) Unifying Theories of Programming: 5th International Symposium, UTP 2014, Singapore, May 13, 2014, Revised Selected Papers, Lecture Notes in Computer Science, vol. 8963, pp. 21–41. Springer-Verlag, Cham (2015). https://doi.org/10.1007/978-3-319-14806-9_2
13. Foster, S., Zeyda, F., Woodcock, J.: Unifying heterogeneous state-spaces with lenses. In: 13th International Colloquium on Theoretical Aspects of Computing. pp. 295–314. ICTAC 2016, Springer-Verlag, Cham (Oct 2016). https://doi.org/10.1007/978-3-319-46750-4_17
14. Gordon, M.J.C., Collavizza, H.: Forward with Hoare. In: Roscoe, A.W., Jones, C.B., Wood, K.R. (eds.) Reflections on the Work of C. A. R. Hoare, pp. 101–121. Springer-Verlag, London (2010). https://doi.org/10.1007/978-1-84882-912-1_5
15. Greenaway, D., Andronick, J., Klein, G.: Bridging the gap: Automatic verified abstraction of C. In: Beringer, L., Felty, A. (eds.) International Conference on Interactive Theorem Proving. pp. 99–115. ITP 2012, Springer-Verlag, Berlin, Heidelberg (Aug 2012)
16. Hoare, C.A.R.: Quicksort. The Computer Journal **5**(1), 10–16 (Jan 1962). https://doi.org/10.1093/comjnl/5.1.10
17. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM **12**(10), 576–580 (Oct 1969). https://doi.org/10.1145/363235.363259
18. Hoare, C.A.R., Hayes, I.J., He Jifeng, Morgan, C.C., Roscoe, A.W., Sanders, J.W., Sørensen, I.H., Spivey, J.M., Sufrin, B.A.: Laws of programming. Communications of the ACM **30**(8), 672–686 (Aug 1987). https://doi.org/10.1145/27651.27653
19. Hoare, C.A.R., He Jifeng: Unifying Theories of Programming. Prentice-Hall, Englewood Cliffs, NJ, USA (1998)
20. Kassios, I.T.: Dynamic frames: Support for framing, dependencies and sharing without restrictions. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) 14th International Symposium on Formal Methods. pp. 268–283. FM 2006, Springer-Verlag, Berlin, Heidelberg (Aug 2006). https://doi.org/10.1007/11813040_19
21. Kassios, I.T.: The dynamic frames theory. Formal Aspects of Computing **23**(3), 267–288 (2011). https://doi.org/10.1007/s00165-010-0152-5
22. Kassios, I.T., Müller, P., Schwerhoff, M.: Comparing verification condition generation with symbolic execution: An experience report. In: Joshi, R., Müller, P., Podelski, A. (eds.) Verified Software: Theories, Tools, Experiments. Lecture Notes in Computer Science, vol. 7152, pp. 196–208. Springer-Verlag, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-27705-4_16
23. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: Formal verification of an OS kernel. In: Proceedings of the ACM SIGOPS 22nd

Symposium on Operating Systems Principles. pp. 207–220. SOSP '09, ACM Press, New York, NY, USA (2009). https://doi.org/10.1145/1629575.1629596, https://sel4.systems/
24. Knuth, D.E.: The Art of Computer Programming, vol. 3: Sorting and Searching, chap. 5.2.1: Sorting by Insertion, pp. 80–105. Addison-Wesley, Boston, MA, USA, second edn. (1998)
25. Matichuk, D., Murray, T., Wenzel, M.: Eisbach: A proof method language for Isabelle. Journal of Automated Reasoning **56**(3), 261–282 (Mar 2016). https://doi.org/10.1007/s10817-015-9360-2
26. Matichuk, D., Wenzel, M., Murray, T.: An Isabelle proof method language. In: Klein, G., Gamboa, R. (eds.) Interactive Theorem Proving: 5th International Conference, Held as Part of the Vienna Summer of Logic. pp. 390–405. ITP 2014, VSL 2014, Springer International Publishing, Cham (Jul 2014). https://doi.org/10.1007/978-3-319-08970-6_25
27. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer (2002)
28. Paulson, L.C., Blanchette, J.C.: Three years of experience with Sledgehammer, a practical link between automatic and interactive theorem provers. In: Proceedings of the 8th International Workshop on the Implementation of Logics. vol. 1. Citeseer (2010)
29. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: 17th Annual IEEE Symposium on Logic in Computer Science. pp. 55–74. IEEE, Piscataway, NJ, USA (2002)
30. Schirmer, N.: Verification of Sequential Imperative Programs in Isabelle/HOL. Ph.D. thesis, Technical University of Munich (2006)
31. Wenzel, M.M.: Isabelle/Isar—A Versatile Environment for Human-Readable Formal Proof Documents. Ph.D. thesis, Technische Universität München, Universitätsbibliothek (2002)
32. Woodcock, J., Foster, S.: UTP by example: Designs. In: Bowen, J.P., Liu, Z., Zhang, Z. (eds.) Engineering Trustworthy Software Systems: Second International School, March 28–April 2, Tutorial Lectures, pp. 16–50. SETSS 2016, Springer-Verlag, Cham (2016). https://doi.org/10.1007/978-3-319-56841-6_2