USIMPL: An Extension of Isabelle/UTP with Simpl-like Control Flow

Joshua Alexander Bockenek

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Masters of Science
in
Computer Engineering

Binoy Ravindran, Chair
Peter Lammich
Robert P. Broadwater

December 8, 2017
Blacksburg, Virginia

Keywords: Formal Verification, Formal Methods, Isabelle, Unifying Theories of Programming, Verification Condition Generation

USIMPL: An Extension of Isabelle/UTP with Simpl-like Control Flow

Joshua Alexander Bockenek

(ABSTRACT)

Writing bug-free code is fraught with difficulty, and existing tools for the formal verification of programs do not scale well to large, complicated codebases such as that of systems software. This thesis presents USIMPL, a component of the Orca project for formal verification that builds on Foster's Isabelle/UTP with features of Schirmer's Simpl in order to achieve a modular, scalable framework for deductive proofs of program correctness utilizing Hoare logic and Hoare-style algebraic laws of programming.

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 thesis are those of the author and do not necessarily reflect the views of ONR or NAVSEA/NEEC.

USIMPL: An Extension of Isabelle/UTP with Simpl-like Control Flow

Joshua Alexander Bockenek

(GENERAL AUDIENCE ABSTRACT)

Writing bug-free code is fraught with difficulty, and existing tools for the formal verification of programs do not scale well to large, complicated codebases such as that of systems software (OSes, compilers, and similar programs that have a high level of complexity but work on a lower level than typical user applications such as text editors, image viewers, and the like). This thesis presents USIMPL, a component of the Orca project for formal verification that builds on an existing framework for computer-aided, deductive mathematical proofs (Foster's Isabelle/UTP) with features inspired by a simple but featureful language used for verification (Schirmer's Simpl) in order to achieve a modular, scalable framework for proofs of program correctness utilizing the rule-based mathematical representation of program behavior known as Hoare logic and Hoare-style algebraic laws of programming, which provide a formal methodology for transforming programs to equivalent formulations.

This work is supported in part by ONR under grants N00014-17-1-2297 and N00014-16-1-2818, and NAVSEA/NEEC under grant N00174-16-C-0018. Any opinions, findings, and conclusions or recommendations expressed in this thesis are those of the author and do not necessarily reflect the views of ONR or NAVSEA/NEEC.

This thesis is dedicated to Mike Gordon, a significant figure in the field of formal verification who unfortunately passed away earlier this year.

Acknowledgments

This work was done in collaboration with Dr. Yakoub Nemouchi, the showrunner of the Orca project, who has implemented most of the extensions to Isabelle/UTP on which I have based my verification condition generator work; Dr. Lammich's assistance was also invaluable due to his experience in developing VCGs and his insights into invariants. I would also like to thank my advisor, Dr. Ravindran, for his support in my academic career so far and want to give a general thanks to the members of committee for taking time out of their schedules to serve on it. It would also be untoward of me to leave out my comrades in the Systems Software Research Group, who deserve thanks as well.

Lastly, but not leastly, I would like to send a shout-out to my family, the members of which have fully supported me in my endeavors of obtaining graduate degrees.

Contents

List of Figures

List of Tables

List of Algorithms

List of Listings

List of Abbreviations

AFP Archive of Formal Proofs

API application programming interface

AST abstract syntax tree

ATP automated theorem proving

CDCL Conflict-Driven Clause Learning

CLR common language runtime

CRAB CoRnucopia of ABstractions

CVC Cooperating Validity Checker

DPLL Davis–Putnam–Logemann–Loveland

FSM finite state machine

GFP greatest fixed point

HOL higher-order logic

IDE integrated development environment

ISM infinite state machine

ITP interactive theorem prover

LCF Logic for Computable Functions

LFP least fixed point

LHS left-hand side

MBT model-based testing

ML Meta Language

NAVSEA the Naval Sea Systems Command

NEEC the Naval Engineering Education Consortium

NP nondeterministic polynomial time

ONR the Office of Naval Research

OS operating system

PVS Prototype Verification System

RHS right-hand side

SAL Symbolic Analysis Laboratory

SAT satisfiability

seL4 secure embedded L4

SLOC source lines of code

SML Standard ML

SMT satisfiability modulo theories

SP strongest postcondition

SSRG Systems Software Research Group

STS Space Transportation System

TCB trusted computing base

UTP Unifying Theories of Programming

VC verification condition

VCG verification condition generator

WP weakest precondition

Chapter 1

Introduction

1.1 General Motivation

Writing software is a complicated business. Programmers encounter bugs in their own code on a regular basis, and in critical systems bugs can have deadly consequences (such as the lethal radiation exposures caused by a mix of issues with the Therac-25 electron accelerator just under four decades ago [63] or the arithmetic errors in the Patriot missile defense system that resulted in the deaths of 28 soldiers during the Gulf War [11]); even in more commonplace modern systems, bugs can cause serious security flaws, such as the OpenSSL Heartbleed vulnerability caused by a missing bounds check (which was present for three years without [publicly-announced] detection!) [92] or the recent macOS Sierra vulnerability that allowed root access without a password even remotely [3][1]. It would be very nice to be able to know with certainty that a program is bug-free, and the field of *formal methods* (using math to reliably prove properties of a system, further explicated in section 2.1), or more specifically *formal verification* [10] (also covered in that section), provides ways of doing this as well as of establishing other correctness properties for security (tools like ProVerif[2]), timing constraints [1, 27], and the like.

Even with existing tools that have been developed over the years, however, the process of formal verification often proves intractable for large pieces of software, with the amount of correctness proofs developed for theorem-based proving (the focus of this thesis and described briefly in section 1.4) often eclipsing the amount of code being verified [56]. Model checking (described in section 2.1.4) and model-based testing (MBT) (described in section 2.1.5) can also be used, but those methodologies have their own scalability issues (such as state [space] explosion for the state-machine-based model checkers, again discussed in the model-checker

---

[1] Strictly speaking, a similar vulnerability also exists in many Linux systems that provide recovery accounts/single-user mode with superuser access that do not have root passwords, but people do not make as big a deal out of that as it is limited to physical access only.

[2] http://prosecco.gforge.inria.fr/personal/bblanche/proverif/

section).

Ultimately, what is needed are lightweight, modular, extensible systems that can handle scaling to large codebases of existing and future systems software such as operating system (OS) kernels and modern compiler collections like GCC[3] and LLVM[4]; this thesis features work on such a system being developed within Virginia Tech's Systems Software Research Group (SSRG), named Orca, that aims to fit those requirements by combining a theorem-based proving approach with MBT. Those two approaches provide complementary methodologies for proving programs correct, with theorem-proving approaches being a form of static verification and MBT providing dynamic testing, much like the unit tests that many programmers already do (but more formal).

1.2 Contributions

As previously mentioned, this thesis presents the theorem-based proving aspect of Orca, referred to as USIMPL; it builds upon Isabelle/UTP [30, 31] and the Simpl language presented by Schirmer [87] as implied by the title of this thesis, adding additional features such as:

- extensions to the Isabelle/UTP language with Simpl constructs, providing a fuller-featured language for verification (though the implementation of these features was not fully complete and tested by the time of this thesis);
- algebraic laws of programming for the full set of features provided by USIMPL, progress on which is still progressing as we do not yet have an exhaustive library;
- and my major contribution, the forward (strongest postcondition (SP)) verification condition generator (VCG) (also still a work in progress).

Isabelle/UTP is an implementation of the formalized programming language introduced by the book Unifying Theories of Programming (UTP) of Hoare and He [47] (described in detail in section 2.3) in the interactive theorem prover (ITP) Isabelle (described in section 2.2) and is elaborated on in section 2.4. Meanwhile, Simpl is an extension of the IMP language of Winskel [100] that adds a few nice features like scoping and abrupt termination; further reading on Simpl can be found in section 2.5.

Those components that are most important for usage but are still a work in progress include proper handling for procedure calls as well as nested scoping, plus a fully operational heap model; lack of these features currently limits the ability of USIMPL to model many typical real-world programs at this point in time. Another aspect that has not been fully tested is support for *total correctness*, which allows proving program termination; currently, USIMPL

---

[3] https://gcc.gnu.org/

[4] https://llvm.org/

only uses *partial correctness*, which assumes termination and thus that there are no infinite loops or infinitely recursing procedures (though as mentioned, USIMPL does not currently support those anyway).

A short elaboration on the usefulness of the above contributions is covered below as well.

1.2.1 Contribution Motivation

More Language Features

By providing more features for improved control flow, we can better model fancier control-flow features such as early loop termination or loop continuations. In doing so, we thus reduce the amount of work that must be done to convert programs utilizing such features into our own formulation, which eases the task of verification as our representation remains closer to the original. This does require more work in the early stages, as in order to represent such features formally their precise semantics must be determined; the difficulty of doing so can differ depending on the feature in question.

Algebraic Laws of Programming

Such laws provide formal equivalences between program fragments, which can be useful for verified compiler transformations and simplification of programs as it is often easy to represent the same calculations or effective control flow in different ways. This has potential usefulness for improving reusability of USIMPL proofs as well by providing the capability to convert common programming paradigms into some canonical form that has already been proven correct.

Forward VCG

The toolset chosen for verifying program correctness in USIMPL, *Hoare logic* [46] (covered in section 2.1.2), requires generation of intermediate *verification conditions (VCs)* that must themselves be proven in order to prove the correctness of the full program. This process would take far too much time and result in long, boring proofs if done manually, and thus USIMPL requires a VCG in order to automate the process of VC generation. There are two main ways of approaching this process as well: start at the beginning and proceed to the end or start at the end and proceed to the beginning. The latter, called weakest precondition (WP) or backward reasoning [24], seems to be more common, but for our purposes we found SP reasoning [41] to be more reasonable.

1.3 Further Organization

USIMPL itself is explained in chapter 3, and if you are familiar enough with the previously-mentioned concepts then you can skip directly there to read; otherwise, it is my recommendation that you peruse the background information mentioned above as well as the other background concepts and work described in chapter 2. After reading about the design of USIMPL, you can check out chapter 4 to browse correctness proofs for two commonly-used algorithms that prove the usability of USIMPL even in its incomplete state. Finally, the features of USIMPL that are incomplete or are planned for future development are again discussed in chapter 5, which concludes this thesis in a succinct manner.

1.4 Automated theorem proving

Not everyone reading this thesis knows what theorem proving in the sense I'm using it in my thesis is, of course, so here is a brief introduction before diving into the bulk of things.

The theorem proving discussed here is specifically automated theorem proving (ATP) [15, 64] (often done via tools such as the ITPs described in section 2.1.6, though noninteractive development is possible as well), which is much like proving by hand, but done in a manner such that users do not have to rely on their own understanding to know that the proof is correct, rather relying on a trusted system to determine parts of the proof/validate existing components of the proof. This enables much more rapid development of correct proofs than could be done simply by sticking with pencil or pen and paper, though it of course requires trusting the system that validates the proof (but then, manual proofs require trusting the person who developed the proof, so ultimately you have to trust something, and in the end you can at least rely on a computer to—usually—give you the same result for the same input).

1.5 Notation

As this thesis contains a lot of Isabelle text, I have used similar notation in other parts of the document as well. For example, the type of a two-tuple is represented as $a \times b$ (or $'a \times {}'b$, if type variable distinction is required); a function with two arguments has type $a \Rightarrow b \Rightarrow c$; and a set of two-tuples with the same type for each field (in other words, a relation) has type $(a \times a)$ set. Theorems and lemmas will be presented in the form $P \Longrightarrow Q$, $A \Longrightarrow B \Longrightarrow C$, etc., where $\Longrightarrow$ represents a top-level implication ($\longrightarrow$ is used for inner implications in Isabelle/HOL). As with the two different implication types, Isabelle/HOL offers both $\bigvee$ and $\exists$ for existential quantifiers and $\bigwedge$ and $\forall$ for universal quantifiers. To match typical functional programming notation, function calls such as $f(a, b)$ will instead be written as $f\ a\ b$. This usage is not entirely consistent due to differences in Isabelle/UTP style as well as some simplification of notation used for section 3.4.1.

Chapter 2

Background

This chapter covers the ground work and backing topics that USIMPL builds upon, giving an introduction to formal methods, Isabelle, UTP, Isabelle/UTP, and the Simpl language as well as some other related works.

2.1 Formal Methods

The field of formal methods is broad, but can be succinctly generalized as the usage of logical inference rules to derive well-formed conclusions in sound mathematical frameworks applied to the domains of hardware and software development [14]. In general, this means conclusions derived via formal methods can be trusted with a high degree of certainty, which is of great importance for security and safety purposes as noted in chapter 1. This work is primarily focusing on using formal methods for software verification; in other words, formal verification [10].

2.1.1 Orders of Logic

First-order logic, also known as predicate logic, is an extension of typical propositional (zeroth-order) logic that adds universal (∀) and existential (∃) quantifiers for individual elements in the *domain of discourse*[1]; extending that to quantifying over sets/relations/functions gets you second-order or, more generally (sets of sets, sets of sets of sets, functions, etc.), higher-order logic (HOL) [91]. HOL is the most flexible, but usage can increase the complexity of proofs. This thesis involves work that builds on HOL in Isabelle.

---

[1] domain/universe/set over which individual elements may be quantified

Zeroth	First	Second	Higher
$P \longrightarrow Q$	$\exists a.\ P\ a$	$\forall P\ x.\ (x \in P \vee x \notin P)$	$\exists f.\ \forall x.\ f\ x = x$

Table 2.1: Statements in Various Orders of Logic

2.1.2 Semantics

There are three main types of semantics used in formal methods: denotational, operational, and axiomatic. The methodologies involved are described as follows.

Denotational

Originally presented by Scott [88], denotational semantics provide a method of representing expressions in a program language as composable mathematical objects, such as functions of the form $s \Rightarrow s$ or relations between states ($(s \times s)$ set), with s being the type of the program state. In Isabelle/HOL, this type of semantics can be implemented using *shallow embedding*, wherein program language features are directly mapped to HOL constructs [99]. While popular, denotational semantics prevent introspection; it is not possible to observe the intermediate state of a program, only the starting and ending states.

Operational

As first used in the development of ALGOL 68 [95] and named by Scott [88], operational semantics are unlike denotational semantics (which are a direct translation to a mathematical form) in that they reason about execution; that is, transitions between program states. To some extent, this can be thought of as (a mathematical formalization of) program interpretation. There are two major forms of operational semantics:

small-step Also known as structural operational semantics [84, 85], wherein a program is evaluated step by step. One can think of it as a (potentially recursive) function of the form $c \times s \Rightarrow c \times s$ that evaluates the first part of a program/command (represented with type c) on some input state of type s and produces the remainder of the program to evaluate along with the state after evaluation of that first component. If the output program is a no-op (`skip`), the resultant state is the final state of the program.

big-step Also known as natural semantics [51]; unlike small-step semantics, a big-step operation will generally be represented with inductive definitions of the form $(c \times s) \Rightarrow s$. As it is concerned solely with the state after program execution, there is no easy way to make determinations regarding intermediate state as can be done with small-step.

Operational semantics are often implemented using *deep embedding*, wherein the language constructs are structured in an abstract syntax tree (AST) and then the function(s) encoding the semantics is/are used to evaluate the AST [99].

Axiomatic

Unlike denotational or operational semantics, axiomatic semantics state program behavior in terms of assertions (logical predicates) on state. The most famous example of axiomatic semantics is Hoare logic [46]; in this formalization, the notation $\{P\}C\{Q\}$ represents a program C with precondition P and postcondition Q. It is often used with Dijkstra's WP reasoning [24], as $\{P\}C\{Q\}$ holds if and only if $P \implies C \text{ wp } Q$, but can be used with SP methodologies as well ($P \text{ sp } C \implies Q$) [41]; in fact, the equivalence is provable in Isabelle/UTP despite the original formulation being axiomatic. For my purposes, this is the most relevant type of semantics as the VCG designed for USIMPL is built off of Hoare rules as shown in section 3.4.1.

A closely related concept is that of algebraic semantics [37, 97] such as Hoare's algebraic laws of programming [48]; this methodology gives algebraic meaning to program semantics and, at least in the case of Hoare's laws of programming, establishes formal equivalences between commonly-encountered program expressions.

2.1.3 Solvers

Satisfiability (SAT) solvers [38, 96] work on Boolean formulas, identifying satisfying assignments for the variables involved (meaning they are zeroth-order solvers). The SAT problem (with more than two variables) is famous for being the first known nondeterministic polynomial time (NP)-complete problem [17], which means that there is no algorithm to determine a solution in polynomial time[2] or faster, but any potential solution can be verified in at most polynomial time. In general, SAT problems tend to be solvable in exponential time[3], which is still not very feasible for extremely large problem sets. Thus, modern SAT solvers focus on optimizing algorithms and minimizing execution time even if worst-case time is still exponential, and they have become very good at it [55]. At the same time, the necessary heuristics may mean decreased flexibility; many SAT solvers work well on problems with specific formulations but are not guaranteed to perform as well on more general problems. That is one of the unfortunate trade-offs that must be made for NP problems and means that no single SAT solver is best for all SAT problems.

SAT solvers can be grouped into two main categories: *conflict-driven backtrackers* (example algorithms being Conflict-Driven Clause Learning (CDCL), and its predecessor, the Davis–

---

[2] Runtime bounded as $\mathcal{O}(n^k)$

[3] $\mathcal{O}(2^{n^k})$

Putnam–Logemann–Loveland (DPLL) algorithm [22]) and *stochastic local searchers* such as GSAT [82] and its successor WalkSat [90]. There are also *look-ahead* solvers, which use backtracking algorithms based on DPLL like conflict-driven solvers but in a predictive fashion [12, 34, 44]; these do not seem to be as common, however. The main differentiation for CDCL solvers and DPLL solvers is that CDCL solvers perform non-chronological backjumping rather than simple backtracking in the search tree when conflicts occur. Another useful SAT solver is Glucose[4], based on the older MiniSAT[5] [28].

In contrast to SAT solvers, satisfiability modulo theories (SMT) solvers [6, 23, 71] are first-order, working with formulas involving quantifiers. The performance limitations of SAT solvers still apply to SMT solvers, unfortunately, but certain SMT problems have known polynomial-time conversions to SAT forms, which can reduce performance costs if the available SMT solvers are not so useful on the specific problems under consideration. Some commonly-used SMT solvers are Microsoft Research's Z3 solver[6] and the Cooperating Validity Checker (CVC) series, with the most recent iteration being CVC4[7]; another one often integrated with other tools is Yices [26] (now up to version 2 [25]) as well as MathSAT5[8].

2.1.4 Model Checkers

While not directly related to my thesis work, model checkers are an interesting comparison for verification. As briefly mentioned in the introduction, these tools represent the systems they analyze as finite state machines (FSMs), or sometimes infinite state machines (ISMs), exhaustively examining the space of possible states for a system in order to prove properties about it [2, 33]. The usage of state machines can be troublesome as the number of states can become quite large, particularly for very complex systems; workarounds include abstraction (simplifying the model), representing the state machines implicitly via Boolean formulas (which reduces the model checking to a SAT or an SMT problem), and putting bounds on the number of state transitions (very useful for ISMs), but the problem will likely never go away completely [16, 83]. Some of the model checkers I have previously been exposed to are Sally[9], which can use the Yices 2 and MathSAT5 solvers as well as Z3 if available, and (for concurrent systems) Symbolic Analysis Laboratory (SAL)[10], which uses Yices 1 (not 2!).

---

[4] `https://www.labri.fr/perso/lsimon/glucose/`

[5] `http://minisat.se/`

[6] `https://github.com/Z3Prover/z3`

[7] `http://cvc4.cs.stanford.edu/web/`

[8] `http://mathsat.fbk.eu/`

[9] `https://sri-csl.github.io/sally/`

[10] `http://sal.csl.sri.com/`

2.1.5 Model-based testing

Again not directly related to my thesis work, MBT [8, 35, 94] provides another way of checking the correctness of programs by formally generating tests that a program must pass to verify that it matches the formal model (i.e. the specification) used to generate the tests. It can be thought of as a form of dynamic testing, much like the unit and integration tests many programmers write by hand, and serves as a complement to static, theorem-based proving by allowing testing of compiled programs on actual hardware, meaning the hardware and compiler can be excluded from the trusted computing base (TCB). The downside of this approach is that tests and their expected results cannot necessarily be accurately generated for all possible inputs due to the fact that the model may not account for certain aspects of the language the code is written in; put another way, the abstract tests generated from the specification may not be mappable to concrete tests that can actually be executed.

There are various methodologies for MBT, including model checking as well as theorem proving; we plan to integrate the theorem-based test-case generator HOL-TestGen[11] [13] as part of Orca eventually.

2.1.6 Proof Assistants

The most relevant formal methods tools for this thesis are ITPs, also known as proof assistants. These tools consist of generic frameworks, and often environments, for the development and discharging of formal proofs, possibly with integration of SAT and SMT solvers. Some major examples are: Coq[12] [9], used for the CompCert compiler certification project[13] [62]; Prototype Verification System (PVS)[14] [78–80], used to formalize software requirements for the Space Transportation System (STS) [21] among other things [81]; and, of course, Isabelle[15] [72], discussed in section 2.2.

2.2 Isabelle

Designed as a successor to the HOL[16] series of ITPs [39] (which were and are, in turn, based on Logic for Computable Functions (LCF) [68, 89]), Isabelle is written in a descendant of the functional language Meta Language (ML) (developed for the aforementioned LCF) called Standard ML (SML) [69]. The implementation of SML used, Poly/ML[17], supports

---

[11] https://www.brucker.ch/projects/hol-testgen/

[12] https://coq.inria.fr/

[13] http://compcert.inria.fr/

[14] http://pvs.csl.sri.com/

[15] https://isabelle.in.tum.de/

[16] https://hol-theorem-prover.org/

[17] http://www.polyml.org/

multi-threaded execution, which is widely used by Isabelle for parallel and asynchronous proof checking when editing Isabelle files (called *theories* and having the `.thy` extension). Isabelle's SML-level application programming interface (API) is available for use by user code as well, which can be written within the Isabelle interface and even embedded in theories if so desired.

At the core of Isabelle is its logical kernel, which handles type-checking, term implementations, and the management of global contexts that contain, among many other things, signature information and basic logical axioms. The kernel certifies that all theorems are derived only from theory contexts and cannot be removed or overwritten. Many different logics can be implemented on top of the kernel using a minimal metalogic known as *Pure*. For example, the HOL extension of Pure, which is probably the most widely-used base theory, consists of only eleven additional axioms [73]. As axioms, we must of course trust that these statements and those of Pure provide a sound framework, but they are all we need to trust as all further extensions of HOL are *conservative*. That is, the vast majority of Isabelle theories building on HOL and the like use specification constructs that call directly to the primitives of the logical kernel, allowing theories to be extended without affecting the soundness of the aforementioned axioms. Plenty of such theories are distributed with Isabelle, such as implementations of the Owicki-Gries [77] and rely-guarantee [102] logics for verification of parallel algorithms; even more libraries, maintained to work with current versions of Isabelle, are available from the online Archive of Formal Proofs (AFP)[18]. The current Isabelle interface is a version of the integrated development environment (IDE) jEdit[19], though previous versions[20] permitted usage of the venerable text editor emacs and experimental support has been added for Visual Studio Code in the most recent version of Isabelle released at the time of this thesis (Isabelle 2017). The rest of this section covers general features of Isabelle relevant to our work.

2.2.1 Types and Functions

Isabelle/HOL supports the typical primitive types used in functional languages, such as integers, natural numbers, reals, and booleans as well as characters. It also provides the usual structural types, such as tuples, (linked) lists, and sets, and most importantly it provides a strong setup for definitions of functions. There is notation for macro-style abbreviations with the `abbreviation` keyword, non-recursive functions with the `definition` keyword, and recursive pattern-matching functions with the `fun` keyword (or, for more primitive functionality, `primrec`). The ability to create notations for commonly-used functions, definitions, and abbreviations further extends Isabelle's usability and can be encountered quite often in the declarations of many built-in types. Users may also provide their own types, building on existing types and using the `datatype` keyword for recursive implementations (also used by many built-in types such as the aforementioned lists and sets as shown in

---

[18] https://www.isa-afp.org/

[19] http://www.jedit.org/

[20] http://isabelle.in.tum.de/download_past.html

```
(* From Isabelle’s HOL/List.thy *)
datatype (set: ’a) list =
    Nil  ("[]")
  | Cons (hd: ’a) (tl: ‹’a list›)  (infixr "#" 65)
for -- ‹extra bits to flesh out the type, not strictly necessary›
  map: map
  rel: list_all2
  pred: list_all
where
  ‹tl [] = []›

(* Record example *)
record Point2D =
  x :: real
  y :: real

record Point3D = Point2D +
  z :: real
```

Listing 2.1: Recursive Types and Records

listing 2.1). Additionally, inductive predicates may be designed using the `inductive` keyword. As an extension to tuples, Isabelle has an extensible record type with named fields, also shown in listing 2.1; records provide `<field>` and `update_<field>` functions for each of their fields to get and update their values, respectively. The `transfer` package, along with the `lift_definition` command, can also be used to ease the pain of building layers of syntax.

Unlike systems such as PVS, Isabelle does not provide predicate subtyping (as an example, PVS defines natural numbers as the subset of integers that are greater than or equal to zero); instead, such restrictions must be established directly as predicates/assumptions in lemmas and theorems. However, Isabelle does provide *type classes*, which define non-type-specific properties. For example, one can specify a type variable restricted to those types that have partial ordering with `’a :: order` or total ordering with `’a :: linorder` (with `linorder` being an extension of the `order` class). This is much like how non-type variables can be annotated with their types (`a :: int`).

2.2.2 Syntax Translations

For cases where simple notation definitions are not enough, users may define their own syntax tokens and the necessary translations as shown in listing 2.2, which demonstrates how such translations can be used to preserve a name; the `v` used there is replaced by whatever is used with the syntax, so $\mathcal{S}$ `v` becomes `select` $\mathcal{V}$ `x` becomes `select (x, x_update)`, which produces `x`. This sort of setup can be used when program state is stored in a concrete record as records provide select/get and update/put functions for each of their fields.

As a side note, that example also demonstrates a simple shallow embedding of expressions and commands (statements), though it is very minimal.

2.2.3 Proofs

The most important component of Isabelle and its primary purpose is the proof system. Every new lemma or theorem added by users introduces one or more proof goals that must be discharged to introduce the lemma/theorem into the containing theory context, and as an LCF-based tool, Isabelle's logical kernel ensures that only valid, proved theorems and lemmas can be used in further proofs (though there are workarounds such as use of the **`sorry`** command, which is solely intended as a placeholder for a real proof and is barred from proofs intended for archival [57]).

To discharge proofs, Isabelle/HOL uses *term rewriting* [49] and *natural deduction* [36, 50]. For simple rewriting of terms in proof goals, the basic `simp` *method* can be used. This method relies on a standard set of simplification theorems (the *simpset*) to rewrite terms from the left-hand side (LHS) of HOL equalities to the right-hand side (RHS). The simpset can be extended with new theorems by attaching the `simp` attribute to a given proven theorem either in the definition itself or at a later point via the **`declare`** command (or supplied directly to `simp` itself) as shown in listing 2.3. For goals with logical connectives, natural deduction can be used via the *tactics* `intro` for introduction rules, `elim` for elimination rules, and `dest` for destruction rules. For more complicated proof goals, Isabelle provides the `auto` method. In addition to rewriting, `auto` performs many other modifications on proof goals such as application of arithmetic and natural deduction rules. There are many more methods for use in specific situations, such as `blast` (for first-order logic and introduction rules for sets) and `fastforce`.

Going beyond individual tactics and methods, there are two ways of discharging proof contexts in Isabelle: apply-script style and Isar (structured) style. While the two can be mixed, Isar methodology tends to look more like the sort of proof a mathematician would develop and thus is preferred by those who like their proofs to be human-readable in isolation. However, apply style (the traditional way of doing Isabelle proofs; Isar was introduced later by Wenzel [98]) is quite useful and produces just the same results; it simply makes most of the proof context implicit and thus requires examination of the proof and subgoal state via Isabelle mechanisms rather than in isolation. A simple example comparing Isar style and apply style can be seen in fig. 2.1.

Isabelle also provides the **`sledgehammer`** command, which will attempt to find proofs for a given subgoal using a variety of different solvers (including the aforementioned Z3 and CVC4, section 2.1.3). If successful, the tool will construct the proof and display it in the output

```
(* From HOL/List.thy *)
syntax
  -- ‹list Enumeration›
  "_list" :: ‹args ⇒ ’a list› ("[(_)]")

translations
  "[x, xs]" ⇌ "x # [xs]"
  "[x]" ⇌ "x # []"

value ‹[1, 2, 3] = 1 # 2 # 3 # []›

(* Other stuff *)
text {*An expression takes a state and returns some value*}
type_synonym (’τ, ’σ) exp = ‹’σ ⇒ ’τ›

text {*A command operates on a state, producing a (possibly new) state.*}
type_synonym ’σ com = ‹’σ ⇒ ’σ›

type_synonym (’τ, ’σ) var = ‹(’τ, ’σ) exp × ((’τ ⇒ ’τ) ⇒ ’σ com)›

lift_definition select :: ‹(’τ, ’σ) var ⇒ (’τ, ’σ) exp› is
  ‹λv. fst v› .

lift_definition update :: ‹(’τ, ’σ) var ⇒ ’τ ⇒ ’σ com› is
  ‹λv x. snd v (λ_. x)› .

syntax
  "_VAR" :: ‹id ⇒ (’τ, ’σ) var› ("𝒱_")
  "select_VAR" :: ‹id ⇒ (’τ, ’σ) exp› ("𝒮_")
  "update_VAR" :: ‹id ⇒ (’τ, ’σ) exp› ("𝒰_")

translations
  "𝒱 v" ⇌ "(v, _update_name v)"
  "𝒮 v" ⇀ "CONST select 𝒱 v"
  "𝒰 v" ⇀ "CONST update 𝒱 v"
```

Listing 2.2: Syntax Translation Examples

```
lemma swap_id[simp]: ‹i < length xs ⟹ swap i i xs = xs›
  unfolding swap_def
  by simp

declare swap_id[simp]

lemma ‹swap 1 1 [1, 2, 3] = [1, 2, 3]›
  by (simp add: swap_id)
```

Listing 2.3: Ways of Adding to the Simpset

```
lemma ‹length (tl xs) = length xs - 1›
  by (cases xs) simp_all

```

(a) Apply Style

```
lemma ‹length (tl xs) = length xs - 1›
proof (cases xs)
  case Nil
  then show ?thesis by simp
next
  case Cons
  then show ?thesis by simp
qed

```

(b) Isar

Figure 2.1: Apply Style Versus Isar

panel, though those proofs are occasionally quite unreadable if the discovered proof requires the `metis` or `smt` tactics with many helper lemmas. As another caveat, generated proofs that use `smt` are not also accepted in archive-ready proofs due to prohibition of the oracle methodology [57]; in such cases, **`sledgehammer`** works best as a starting point for developing a more readable proof.

Also of use for our purposes is the Eisbach library developed by Matichuk et al. [65, 66], which provides a way of composing proof tactics and methods into other methods without needing to work on the SML level; it even works within proofs, as shown in listing 2.4. This greatly reduces the amount of work needed to automate application of the component methods and tactics of a VCG.

2.2.4 Locales

For modularity, users can create new locales that provide additional assumptions, type fixings, and the like that restrict the contained theorems and lemmas, much like type classes. As an example, the lens properties described in section 2.4.1 are associated with specific lenses

```
(* tries simp, then tries auto if simp doesn’t fully discharge the goal *)
method simp_or_auto = simp; fail|auto

lemma ‹A ⊆ B ∩ C ⟹ A ⊆ B ∪ C›
  by (simp; fail|auto)

lemma ‹A ⊆ B ∩ C ⟹ A ⊆ B ∪ C›
  by simp_or_auto
```

Listing 2.4: Eisbach Example

```
locale less_than_100 =
  fixes x :: int
  assumes ‹x < 100›

lemma ‹less_than_100 x ⟹ x < 100›
  by (simp add: less_than_100_def)

locale between_0_100 = less_than_100 +
  assumes ‹x > 0›

lemma ‹between_0_100 x ⟹ 0 < x ∧ x < 100›
  by (simp add: between_0_100_axioms_def between_0_100_def less_than_100_def)
```

Listing 2.5: Locale Example

by means of locales (e.g. using `vwb_lens x` as an assumption). Another, simpler example of using locales is presented in listing 2.5.

2.3 Unifying Theories of Programming (UTP)

A product of the work of Hoare and He [47], UTP was written to provide denotational semantics for a generalized nondeterministic imperative programming language expressed in a common setting. In the UTP framework, a *theory* consists of three parts: an *alphabet*, a *signature*, and *healthiness conditions*. A further introduction to UTP can be found in Woodcock and Foster [101], if so desired.

alphabet The set of observational variables characterizing the state of the studied system.

signature The different operations used to change the variables in the alphabet.

healthiness conditions Restrictions on the semantic machinery; can be implemented using auxiliary/logical variables to capture given behaviors.

order Not an explicit component, but defines how individual programs relate to each other.

The base UTP theory is the *alphabetized predicate calculus*, where the alphabet draws from `UNIV` and there are no healthiness conditions. An alphabetized predicate is then a pair of alphabet and predicate such that all variables used in the predicate are in the alphabet. More restrictive is the alphabetized *relational* calculus, which Isabelle/UTP is built around; this theory requires that a relation's alphabet contains input (unmarked, e.g. v) variables, which are observations of program state before execution, and output (v') variables, which are observations of state after execution. As an example, consider a simple one-statement program $x \coloneqq x + 1$; in UTP, this would be a relation of the form $x' = x + 1$, where x represents the input value of the variable x (more pedantically, the value of the variable x in the initial state s), $x + 1$ is an expression evaluated on the initial state, and x' is the output value of the variable x (the value of the variable x in the final state s', which is an updated version of the state s after application of the assignment).

Relations

Unlike the common meaning of relation stated in section 1.5, UTP represents relations as functions with type $\alpha \times \beta \Rightarrow \mathrm{bool}$, taking a pair of states (input and output) and evaluating to booleans indicating whether the output state can be produced from the input state; of course, the state spaces (in other words, the state types) do not have to correspond. Relations in this form are *heterogeneous* ((α, β) rel); when the state spaces are the same, the relations are *homogeneous* (α hrel). In most of our work (including the previous assignment example), our extension of Isabelle/UTP utilizes homogeneous relations, which simplifies things down the line.

Ordering and Refinement

Though not invented by Hoare or He, *refinement* serves an important purpose in UTP. This is because programs and specifications are one and the same: a specification is just a more abstract/less restrictive program than its implementation, and the two share a *correctness relation*. If an implementation manipulates the same externally-viewable variables as its specification, their correctness relation is refinement [101]. The notation used for refinement in UTP is $A \sqsupseteq B$, meaning A refines B, or $A \sqsubseteq B$, meaning A is refined by B (this is the version Isabelle/UTP uses). More formally, refinement is the universal closure of an implication over the given alphabet; that is, $A \sqsubseteq B$ iff $[B \Rightarrow A]$ (using the UTP notation for implication, $\Rightarrow$, and universal closure[21], $[\_]$). In this way, the hierarchy of programs

---

[21] universally quantifying all free variables in the enclosed expression

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

lemma wp_hoare: ‹`P ⇒ C wp Q` = ⦃P⦄C⦃Q⦄ᵤ›
  by rel_blast

lemma sp_hoare: ‹`P sp C ⇒ Q` = ⦃P⦄C⦃Q⦄ᵤ›
  by rel_blast
```

Listing 2.6: Hoare Logic and WP/SP

(relations) can be said to form a *complete lattice*, a partially-ordered set wherein the program that does everything (“miracle”) is the top element of the lattice ($\top$, also `false`) and the “abort” program is the bottom element ($\bot$ or `true`).

Recursion

Due to this lattice behavior, we can handle recursion, and thus iteration, via *fixed points* (put simply, the values of x for which $f\ x = x$; also often called *fixpoints*), which themselves form a complete lattice. The most important fixed points in recursion are the greatest lower bound (least fixed point (LFP), μ) and least upper bound (greatest fixed point (GFP), ν), which are used for total (handles nontermination) and partial (does not handle nontermination) correctness, respectively; for a start on further reading on fixpoint recursion, see the work of Lassez et al. [59].

2.4 Isabelle/UTP

As mentioned in the introduction, this thesis would not exist in its current form without the development of an Isabelle implementation of UTP by Foster et al. [31]. While there are other mechanizations of UTP that have come before, such as the Circus implementation [74, 76], Foster's Isabelle/UTP was chosen in part because of their usage of shallow embedding, which allows for increased proof automation due to not needing an AST parser, as well as because of their comparatively elegant generalization of UTP expressions that allows for natural derivations of the different theorems in HOL and their usage of *lenses* as variable and state abstraction. As an example of the strength of Isabelle/UTP, listing 2.6 shows how the Isabelle/UTP framework can be used to prove the WP/SP connections to Hoare logic described in section 2.1.2 (expressed via refinement in this framework).

The rest of this section covers some of the other design decisions made when implementing UTP in Isabelle; if you want to see their own explanations and play around with it yourself, check out the main web page (which at the time of writing of this thesis shows an example of their own VCG, it appears) or you can go directly to the github repository and clone that.

2.4.1 Lenses

Though a deep theory in its own right, this formalization, also by Foster et al. [32], is structured quite simply; a lens consists of get : $s \Rightarrow v$ and put : $s \Rightarrow v \Rightarrow s$ functions that provide a "view" into some source object, where v is the view type and s is the source type. Individual variables, groups of variables, and even more exotic structures can all be represented by lenses. The notation used by Foster and co. to describe the above function pair is $v \Longrightarrow s$ and will be used from now on when typing individual lenses.

All properties of lenses are based on the interaction of their `get` and `put` functions and those of other lenses. For example, two independent lenses $(x \bowtie y)$ satisfy the properties

$$\text{put}_x\,(\text{put}_y\ \sigma\ v)\ u = \text{put}_x\,(\text{put}_y\ \sigma\ u)\ v \tag{2.1}$$
$$\text{get}_x\,(\text{put}_y\ \sigma\ v) = \text{get}_x\ \sigma \tag{2.2}$$
$$\text{get}_y\,(\text{put}_x\ \sigma\ u) = \text{get}_y\ \sigma \tag{2.3}$$

and a lens is *unrestricted* [75, 76] by an expression $(l \sharp e)$ if the following property holds:

$$\forall \sigma\ v.\ e\ (\text{put}_l\ \sigma\ v) = e\ \sigma. \tag{2.4}$$

The main hierarchy of lens properties is described below.

weak A lens that satisfies the `PutGet` property

$$\text{get}\ (\text{put}\ \ \sigma\ v) = v. \tag{2.5}$$

well-behaved A weak lens that also satisfies the `GetPut` property:

$$\text{put}\ \ \sigma\ (\text{get}\ \ \sigma) = \sigma. \tag{2.6}$$

mainly-well-behaved A weak lens that also satisfies the `PutPut` property:

$$\text{put}\ \ \sigma\ (\text{put}\ \ \sigma\ v)\ u = \text{put}\ \ \sigma\ u. \tag{2.7}$$

very-well-behaved A lens with both well-behaved and mainly-well-behaved properties.

bijective A weak lens that also satisfies the `StrongGetPut` property (making it a well-behaved lens as well):

$$\text{put}\ \ \sigma\ (\text{get}\ \ \rho) = \rho. \tag{2.8}$$

ineffectual A read-only weak lens (put $\sigma\ v = \sigma$). Satisfies the very-well-behaved properties by its nature.

Lens Operations

Lenses can also be combined in various ways, of which the following is merely a sample.

composition (`;`) Produces a view of a view: $(b \Longrightarrow c) \Rightarrow (a \Longrightarrow b) \Rightarrow (a \Longrightarrow c)$.

sum (`+`) Combines two views with the same source: $(a \Longrightarrow s) \Rightarrow (b \Longrightarrow s) \Rightarrow (a \times b \Longrightarrow s)$.

product ($\times$) Combines two views with different sources: $(a \Longrightarrow b) \Rightarrow (c \Longrightarrow d) \Rightarrow (a \times c \Longrightarrow b \times d)$.

There is also a developed notion of *sublenses*; x is a sublens of y, denoted as $x \preceq y$, if the source region of x is contained within that of y. More formally, $x \preceq y$ if there is some well-behaved lens z such that $x = z; y$.

2.4.2 Notation and Semantics

In order to reproduce the notation used in the UTP book, the developers of Isabelle/UTP used liberal application of syntax translations (described in section 2.2.2), though the notation does not exactly match that used in the UTP book or lens papers due to Isabelle considerations (such as usage of $\langle\rangle$ rather than <> for building lists/sequences and $\subseteq_{\mathtt{L}}$ instead of $\preceq$ to indicate a sublens relationship). While this works in standardizing the interface, it can cause problems when additional Isabelle libraries are utilized by theory files that incorporate UTP usage, though the developers have provided some workarounds for that. There are also occasional issues when using `sledgehammer` and term-generating tools, as the Isabelle setup is designed with the assumption that generated terms can be evaluated as-is but Isabelle/UTP violates that assumption at times due to ad-hoc overloading that requires explicit typing to resolve (when the usual Isabelle functions are differentiated enough such that explicit typing is not often required). Some examples of notation equivalence are shown in fig. 2.2, which also demonstrates the ad-hoc overloading done by Isabelle/UTP. Additional notation that is useful for future reference:

«v» This indicates 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.

$[\![P]\!]_{\mathtt{e}}$ (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 2.6), respectively.

`length xs (* for lists *)`	$\texttt{\#}_{\texttt{u}}\texttt{(xs)}$
`card s (* set cardinality *)`	$\texttt{\#}_{\texttt{u}}\texttt{(s)}$
`pcard p (* for partial functions *)`	$\texttt{\#}_{\texttt{u}}\texttt{(p)}$
`[1, 2, 3] (* list literal *)`	`⟨1, 2, 3⟩`
`xs @ ys (* list concatentation *)`	$\texttt{xs}\ \hat{}_{\texttt{u}}\ \texttt{ys}$
`xs ! i (* list indexing *)`	$\texttt{xs(i)}_{\texttt{a}}$
`f x (* (p)fun application *)`	$\texttt{f(x)}_{\texttt{a}}$
`x :: 'a (* type fixing *)`	$\texttt{x :}_{\texttt{u}}\ \texttt{'a}$
`fst t (* tuple accessing *)`	$\pi_1\texttt{(t)}$
`snd t (*   "       " *)`	$\pi_2\texttt{(t)}$
`xs[i := x] (* list update *)`	$\texttt{xs(i} \mapsto \texttt{x)}_{\texttt{u}}$ `(* also fun/map update *)`
`set xs (* set of list elements *)`	$\texttt{ran}_{\texttt{u}}\texttt{(xs)}$
`λx. p`	`λ x · p`
`⟶ (* HOL implication *)`	`⇒`

(a) Isabelle/HOL (b) Isabelle/UTP

Figure 2.2: Some Isabelle/UTP Syntax Comparisons

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

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

2.4.3 Language Constructs

Isabelle/UTP provides most of the language constructs detailed in the UTP book, including some of the features for parallel computation, but the main ones for our sequential, imperative purposes are

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

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

$\nu X \bullet P$ 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.

$a : \llbracket P \rrbracket$ Frame; executes P and then restores all elements of the state referenced by lens a to their values before execution of P.

$a : [P]$ Antiframe; executes P and then restores all elements of the state other than those referenced by lens a to their values before execution of P.

2.5 Simpl

Essentially an extension of the IMP language presented by Winskel [100] (meaning it lacks many fancy—or messy, depending on your perspective—programming language features), Schirmer's Simpl adds abrupt termination via `try...catch`-style notation as well as scoping and local-to-procedure variables via a `block` notation, plus state-dependent dynamic commands. Simpl also provides a one-branch `if` statement where the body is executed when the condition is true and is otherwise a no-op, but that is not strictly required as all such expressions can be replaced with `if` statements wherein the false branch is a skip (and, in fact, that is how the expression is ultimately implemented in the underlying abstract syntax). It additionally provides a guard statement that can be used to check for division by zero, integer underflow and overflow, out-of-bounds array accesses, and other failure conditions, working much like assertions in common imperative programming languages. There is also support for side effects in expressions, but we do not deal with those other than for the possibility of procedures modifying global/nonlocal state. listing 2.7 shows a basic program written in Simpl's concrete syntax.

2.5.1 Abrupt Termination

This is just a way of skipping over statements in a program in a controlled fashion. Basic usage occurs via the `THROW` statement, which puts the program in an abrupt state. If the `THROW` does not occur within a `TRY` region, this means the program just terminates; otherwise, the program will proceed to the corresponding `CATCH`, emerge from the abrupt state, and execute the body of the `CATCH` (which may place the program back into an abrupt state). This is patterned off of exception handling in languages such as C++ and can be used to implement that behavior. Abrupt termination can also be used to implement `break` and `continue` for loops as shown in listing 2.8; `break` can be represented with try-catch containing a loop while `continue` can be represented with a loop containing a try-catch. This methodology can also be used to handle early `returns` in procedures.

```
n := length A;
swapped := true;
WHILE swapped DO
    swapped := false;
    i := 1;
    WHILE i < n DO
        IF A[i-1] > A[i] THEN
            A := A[i := A[i-1], i-1 := A[i]];
            swapped := true
        FI
    OD
OD
```

Listing 2.7: Bubble Sort in Simpl

```
TRY c1 CATCH c2 END

break: TRY WHILE b DO ... THROW ... OD CATCH SKIP END

continue: WHILE b DO TRY ... THROW ... CATCH SKIP END OD
```

Listing 2.8: Abrupt Termination Syntax

As there may be many different causes of abrupt termination, Simpl provides an auxiliary variable called `Abr` that should be set to an appropriate value before triggering of abrupt termination via the `THROW` keyword.

2.5.2 Blocks

This feature is ultimately quite simple; blocks provide a way of capturing state at the entry and exit point(s) of a scope or procedure call in order to restore specific variables and, if so desired, perform initialization on entry (for example, assigning function parameters that are expressions to the corresponding named arguments). Initialization is performed via the `init` function and state restoration is done via the `return` function; `c` is used to handle cases where the return value of a procedure is assigned to a variable determined via an evaluated expression that could have been affected by execution of the procedure (e.g. assigning the result to a nested object/pointer that could have been modified by the procedure), which is why it requires access to state on block entry. The states are captured via usage of Simpl's dynamic commands, as Simpl commands normally do not have explicit access to the current state.

As noted in fig. 2.3 and listing 2.9, abrupt termination requires special handling in blocks.

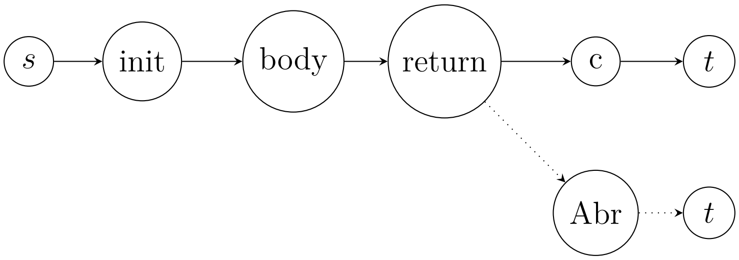

Figure 2.3: Control Flow for Blocks

```
definition block :: ‹('s ⇒ 's) ⇒ ('s, 'p, 'f) com ⇒ ('s ⇒ 's ⇒ 's) ⇒ ('s ⇒
    's ⇒ ('s, 'p, 'f) com) ⇒ ('s, 'p, 'f) com› where
  ‹block init bdy return c ≡ DynCom
    (λs. TRY Basic init; bdy CATCH Basic (return s); THROW END; DynCom
      (λt. Basic (return s); c s t))›
```

Listing 2.9: Block Syntax

If abrupt termination occurs in a block and is not resolved in the body of that block, the return function of the block must still be executed to restore the values of local variables on exit, and so every block has a try-catch construct to ensure that restoration occurs.

2.6 Other Related Work

There has been plenty of work over the years on development of tools for software verification that uses Isabelle, PVS, or some other verification framework; two works that I found very relevant to my research were the secure embedded L4 (seL4) project and the tool and language Dafny, both discussed below.

2.6.1 seL4

A major example of success in the development and verification of a specific piece of software is the seL4 project [56], which used Isabelle for all of its formal proofs. L4[22] is a family of OS *microkernels*, designed to provide the minimal amount of support necessary for a computer to run. Everything that could possibly be done in user space is excluded from kernel space, which greatly minimizes the amount of kernel code that must be written (albeit with a corresponding increase of user-space code) and can provide increased security and reliability [93]. In the case of seL4, that meant the amount of code that would need to be verified could be much less than that which would be required for large kernels like those

---

[22] http://www.l4hq.org/

used in Windows, Linux, and similar OSes (around 10 000 source lines of code (SLOC), to be specific); even so, with a kernel designed from scratch, the project took eight years to complete from its start in 2006. Usefully, the project provides a verified framework for parsing the C code of the project (restricted to a more easily verifiable subset of C) into syntax suitable for verification, which is more trustworthy than relying on manual conversion.

Leaving aside further details of the project itself, their VCG utilizes a Hoare-logic-based approach much like the one presented in this thesis. At one point I attempted to use their VCG as an inspiration for my own work, but it proved to be too specific to their purposes for any adaptation on my part.

2.6.2 Dafny

Taking a different approach to that of external verifiers and the like, Dafny[23] is a tool developed at Microsoft Research, inspired by and building on previously-existing Microsoft tools such as Spec# [5] and Boogie[24] [4, 61]; Written in C#, it was originally created for the purpose of analyzing dynamic frames [52–54], but it has since grown into a full-fledged imperative object-oriented language that includes a built-in verification system. For verification, Dafny code is converted to the Boogie language, which is then passed to a Boogie tool that converts the description into a form suitable for SMT solving and then supplies it to (by default) Microsoft's Z3 solver. When using the proper tools, this happens in real time as users write code, providing responsive feedback much as an ITP does. Compilation uses Microsoft's common language runtime (CLR) framework, producing either C# code or, by default, libraries or executables. An example of an algorithm implemented in Dafny is shown in listing 2.10, though that is quite simple and does not take advantage of Dafny's more advanced features such as classes, dynamic frames, or theorem-proving (yes, Dafny can do that too, albeit in an equality-based manner via `calc` blocks); it does, however, demonstrate preconditions (the `requires` clauses), postconditions (the `ensures` clause), and loop invariants (the `invariant` clauses). This usage of pre-/postconditions and invariants is very much like the work that must be done for verification via Hoare logic.

In terms of interface, Dafny has bindings for Visual Studio as well as emacs (via Boogie friends[25]); there are also syntax highlighting packages for Sublime Text[26] and Atom[27], that do not provide real-time feedback. Listing 2.10 also shows one of the nice features of the emacs interface, its ability to pretty-print Dafny code; the code is written in plain text but displayed with more mathy characters (such as ∀ for `forall`, ≠ for `!=`, etc.). The old Isabelle bindings for emacs used similar functionality to pretty-print theories.

---

[23] https://github.com/Microsoft/dafny

[24] https://github.com/boogie-org/boogie

[25] https://github.com/boogie-org/boogie-friends

[26] https://github.com/tvi/sublime-dafny

[27] https://atom.io/packages/language-dafny

```
method BinSearch(a: array<int>, key: int) returns (present: bool)
    requires a ≠ null
    requires ∀ i,j • 0 ≤ i < j < a.Length ⟹ a[i] ≤ a[j] // sorted
    ensures key ∈ a[..] ⟺ present
{
    var low, high := 0, a.Length - 1;

    while low ≤ high
        invariant 0 ≤ low ≤ a.Length
        invariant 0 ≤ high + 1 ≤ a.Length
        invariant key ∉ a[..low]
        invariant key ∉ a[high+1..a.Length]
    {
        // use ‘low + (high - low) / 2‘ for languages with limited precision
        var i := (low + high) / 2;

        if a[i] < key {
            low := i + 1;
        } else if a[i] > key {
            high := i - 1;
        } else {
            return true;
        }
    }

    present := false;
}
```

Listing 2.10: Binary Search in Dafny

As a final note, Dafny itself has not been formally verified; thus, its compiler and verifier must be treated as part of the TCB when using Dafny to create verified programs.

Chapter 3

Extending Isabelle/UTP

While Isabelle/UTP is interesting by itself, its purpose here is to serve as a foundation for our work, which is an extension of Isabelle/UTP with verified algebraic laws like those mentioned in section 2.1.2, scoping and distinction between local/global variables, SP reasoning and corresponding Hoare rules, and building on that an SP VCG.

Planned features that have been implemented but not properly tested include total correctness, abrupt termination, and expressions with side effects; as such, they are not covered in this thesis aside from being mentioned for future work.

3.1 Program State

As we build on Isabelle/UTP, we also use lenses to model individual variables; in fact, using lens composition, we can divide up the regions of the state that represent different types of variables, such as locals versus globals (though this is not the only way to handle locals versus globals; it is also possible to represent the full state using lens addition, like $\texttt{vars = gvars} +_L \texttt{lvars}$). This can complicate the methodology, as shown in listing 3.1, but provides potentially useful behavior in situations where we do not have a full heap model to manage global state. We have not yet used this capability to any significant extent, however. Ultimately, the full state is again an alphabet that can be represented using Isabelle records.

Additionally, the Isabelle/UTP framework does not have a requirement for variable initialization or declaration; thus, we can simply set up local and global variable state in the preconditions of our Hoare triples if need be. Otherwise, we assign values at the start of whatever program fragment is being worked with. This does mean we do not capture the potential error condition of C programs wherein a variable is assigned to before it is declared, but in most C programs that will simply be a compile-time error rather than a runtime bug, and the times it would be a runtime bug involve scoping, which is covered in section 3.3.

```
locale globloc =
  fixes lvars :: ‹'l ⟹ 's›
    and gvars :: ‹'g ⟹ 's›
    and   ret :: ‹nat ⟹ 'g›
    and     x :: ‹nat ⟹ 'l›
    and     y :: ‹nat ⟹ 'l›
  assumes INDEP: ‹vwb_lens lvars› ‹vwb_lens gvars› ‹vwb_lens ret› ‹vwb_lens x›
                 ‹vwb_lens y› ‹lvars ⋈ gvars› ‹x ⋈ y›
begin

abbreviation ‹Lx ≡ x ;L lvars›
abbreviation ‹Ly ≡ y ;L lvars›
abbreviation ‹Gret ≡ ret ;L gvars›

end
```

Listing 3.1: Differentiating Local and Global Variables

3.2 Algebraic Laws

Utilization of algebraic laws that have been implemented as lemmas in Isabelle allows increased potential for verified transformation (improving on the unverified transformations done by modern compilers) and simplification of the features provided by Isabelle/UTP and USIMPL; in fact, many of the laws are themselves based on the laws presented by Hoare [48], for which Isabelle/UTP provides a basic set already, but that set of laws is far from comprehensive. Some examples of the additional algebraic laws we have implemented for USIMPL are shown in listing 3.2; the full set that has been developed so far can be found in appendix A.1, and of course we plan on publishing a more comprehensive set at a later date.

3.3 Scoping

We initially planned on using Simpl's block notation to implement local variable scoping and procedure calls, but we found that Isabelle/UTP offers *framing* and *antiframing* mechanisms built on lenses that appear more elegant than the methodology used for blocks. Much like separation logic [86], framing provides a way of restricting the state that a program fragment accesses; similar concepts of framing have appeared in the works of Leino [60] and Kassios [52]. In fact, the concept of framing goes back to initial developments in artificial intelligence and the *frame problem* [67].

Unfortunately, we were not able to determine a proper set of Hoare rules for framing by the time of writing of this thesis, in part due to the way lenses are formulated; this currently

```
lemma skip_r_alpha_eq: ‹II = ($Σ´ =ᵤ $Σ)›
  by rel_auto

lemma skip_r_refine_orig: ‹(p ⇒ p) ⊑ II›
  by pred_blast

lemma skip_r_eq[simp]: ‹⟦II⟧ₑ (a, b) ⟷ a = b›
  by rel_auto

lemma assign_test[symbolic_exec]:
  assumes ‹mwb_lens x›
  shows ‹(x :== «u» ;; x :== «v») = (x :== «v»)›
  using assms
  by (simp add: assigns_comp subst_upd_comp subst_lit usubst_upd_idem)

lemma assign_r_comp[symbolic_exec]: ‹(x :== u ;; P) = P⟦⌈u⌉</$x⟧›
  by (simp add: assigns_r_comp usubst)

lemma assign_twice[symbolic_exec]:
  assumes ‹mwb_lens x› and ‹x ♯ f›
  shows ‹(x :== e ;; x :== f) = (x :== f)›
  using assms
  by (simp add: assigns_comp usubst)

lemma assign_commute:
  assumes ‹x ⋈ y› ‹x ♯ f› ‹y ♯ e›
  shows ‹(x :== e ;; y :== f) = (y :== f ;; x :== e)›
  using assms
  by (rel_auto, simp_all add: lens_indep_comm)

lemma cond_ueq_distr[urel_cond]:
  ‹((P =ᵤ Q) ◁ b ▷ (R =ᵤ S)) = ((P ◁ b ▷ R) =ᵤ (Q ◁ b ▷ S))›
  by rel_auto

lemma cond_conj_distr[urel_cond]: ‹((P ∧ Q) ◁ b ▷ (P ∧ S)) = (P ∧ (Q ◁ b ▷ S))›
  by rel_auto

lemma cond_disj_distr [urel_cond]: ‹((P ∨ Q) ◁ b ▷ (P ∨ S)) = (P ∨ (Q ◁ b ▷ S))›
  by rel_auto

lemma comp_cond_left_distr: ‹((P ◁ b ▷ᵣ Q) ;; R) = ((P ;; R) ◁ b ▷ᵣ (Q ;; R))›
  by rel_auto
```

Listing 3.2: A Sampling of Algebraic Laws

```c
int f(int x) {
    return x + 1;
}
```

(a) C

```
definition ‹f r a = gvars:[
    Lx :== a;;
    Gret :== (&Lx + 1)
];; r :== &Gret›
```

(b) USIMPL

Figure 3.1: Procedure Scoping

restricts the usability of USIMPL, which is unfortunate due to the ease of the scoping as shown in fig. 3.1 (this example uses the variables introduced in listing 3.1; the assignment to `r` functions like the usage of *c* mentioned in the description of Simpl's blocks as `r` can be calculated before the procedure call and `a` is some expression that could be as simple as a lens get or something more complicated).

3.4 Strongest postcondition (SP) VCG

WP reasoning, mentioned in section 2.1.2, is also known as *backward* reasoning due to the direction of proving; by contrast, SP reasoning is known as *forward* reasoning. WP reasoning is more common, possibly due to the previously-mentioned influence of Dijkstra, but both methodologies are perfectly valid as explained by Gordon and Collavizza [41]. For our purposes, SP reasoning turned out to be easier to automate and easier to handle when encountering situations where the automated reasoning was not enough, and thus that is what we have gone with for our VCG. As we have proof of the connections between Hoare logic and SP reasoning, mentioned in listing 2.6 and shown in more detail in appendix A.3, we have a strong argument for the correctness of the VCs generated by our VCG (that is, it generates the strongest possible postconditions based on the supplied precondition(s)). Those VCs are generated by the Hoare rules presented below.

3.4.1 Forward Hoare Rules

Already mentioned in listing 2.6 and further detailed in appendix A.2, we work with a definition of Hoare logic wherein the program is a refinement of the statement of the precondition implying the postcondition. This works for partial correctness, but proper total correctness requires an additional statement of program termination/maintaining of healthiness conditions.

The Hoare rules used for an SP VCG differ slightly from those for backward reasoning due to the focus on postconditions rather than preconditions, and the placement of annotations differs as well as described in section 3.4.1. The order of premises of the following rules is, in fact, important, as on application of a rule, the rule's premises become the generated subgoals that must then be proved in order for verification (and we want to approach that proving in a forward manner). The notation used here does not directly correspond to that used in the Isabelle/UTP and USIMPL source, as that notation is tweaked to avoid confusion with normal Isabelle notation; it is primarily intended to give an understanding of the basic logic without worrying about typing or Isabelle HOL-versus-Pure subtleties. Refer to appendix A.4 for the "correct" notation, which is currently just for partial correctness as total correctness had not been fully implemented and tested by the time this thesis was written.

Sequencing and Assignment

$$\frac{\{p\}c_1\{q\} \qquad \{q\}c_2\{r\}}{\{p\}c_1; c_2\{r\}}\ \text{Sequential Composition}$$

$$\frac{}{\{p\}\mathrm{II}\{p\}}\ \text{Skip}$$

Isabelle/UTP provides Hoare rules for assignment,but those are geared towards WP reasoning. For forward proving, rather than relying on the usual Hoare assignment rule, we must use Floyd's rule [29] (which we have shown produces the SP for assignment as seen in listing 3.3); as is usual for Hoare-style rules and much like the usage in Isabelle/UTP, the $e[v/x]$ notation means "substitute v for x in expression/predicate e".

$$\frac{\text{vwb\_lens}\ \ x}{\{p\}x \leftarrow e\{\exists v.\ x = e[v/x] \land p[v/x]\}}\ \text{Floyd Assignment}$$

While this does introduce an existential quantifier for every assignment in a program, the `{pred|rel}_*` methods provided by Isabelle/UTP do a fine job of eliminating the quantifier when discharging the verification conditions our VCG generates. Also note the requirement that the variable (lens) assigned to must be very-well-behaved; this is a consequence of the generality of lenses, which need restrictions to behave more like typical variables in an imperative language.

Control Flow

For simplicity, our imperative control flow is handled by `if` statements and `while` loops alone, though we do have support for general recursion (rules in section 3.4.1). Part of the reason, of course, is that Hoare rules for `for` loops are tricky to set up as such loops are surprisingly complicated [40].

```
lemma sp_assigns_r [sp]:
  ‹vwb_lens x ⟹ (p sp x :== e) = (∃v · p⟦«v»/x⟧ ∧ &x =u e⟦«v»/x⟧)›
  apply rel_simp
  apply transfer
  apply auto
   apply (rule_tac x = ‹getx y› in exI)
   apply simp
  apply (metis vwb_lens.put_eq)
  done
```

Listing 3.3: Floyd is strongest postcondition

The different levels of abstraction in Isabelle/UTP and USIMPL use subtly different notations for conditionals; for this descriptive context, I chose a notation that is reasonable and consistent with the other imperative control-flow statement we use.

$$\frac{\{b \land p\}c_1\{q\} \qquad \{\neg b \land p\}c_2\{r\}}{\{p\}\text{if } b \text{ then } c_1 \text{ else } c_2 \text{ fi}\{q \lor r\}}\ \text{Conditional}$$

USIMPL currently requires usage of an invariant-annotated while loop, but it would be equally feasible to introduce the invariants as part of the VCG. Either way, we still must use explicit invariants as there does not currently seem to be a highly generic framework for invariant inference.

$$\frac{p \longrightarrow i \qquad \{i \land b\}c\{q\} \qquad q \longrightarrow i}{\{p\}\text{while } b \text{ invr } i \text{ do } c \text{ od}\{\neg b \land i\}}\ \text{While with Invariant}$$

Recursion

As mentioned in section 2.3, recursion in UTP is based off of the least- and greatest-fixed-point operators, respectively representing total and partial correctness. However, as mentioned in the chapter introduction, our current recursion setup is only for partial correctness as our initial total correctness paradigm proved inconsistent. This means USIMPL in its current state can only be used to prove properties of programs that terminate, which is certainly a flaw but not a terrible one.

$$\frac{\forall c.\ \{P\}c\{Q\} \longrightarrow \{P\}F\ c\{Q\}}{\{P'\}\nu\ F\{Q\}}\ \text{Recursion}$$

The following rule was used in our (unfinished) recursion testing:

$$\frac{P' \longrightarrow P \qquad \forall c.\ \{P\}c\{Q\} \longrightarrow \{P\}F\ c\{Q\}}{\{P'\}\nu'\ P\ F\{Q\}}\ \text{Annotated Recursion}$$

where ν' is just a wrapper for ν that discards P.

We also have a rule regarding refinement of recursion:

$$\frac{(C \longrightarrow S) \sqsubseteq F\ (C \longrightarrow S)}{(C \longrightarrow S) \sqsubseteq \nu\ F}\ \nu\ \text{Refinement}$$

Assumptions and Assertions

These annotations, predicates on state marked with $^\top$ for assumptions and $_\perp$ for assertions (this notation coming from UTP), are necessary for a full-featured VCG. For our tool, we insert them into the (converted) code directly (as opposed to introducing them during VCG application). Unlike in backward proving, assertions are not required before any non-initial control-flow statements. Instead, they may be required after conditionals and the like. To avoid having to do SML-level pattern-matching in the VCG itself in order to apply the necessary rules for assertions, we placed the pattern-matching into Hoare rules that perform the necessary matching.

$$\frac{}{\{p\}q^\top\{p \wedge q\}}\ \text{Assumption}$$

$$\frac{p \longrightarrow q}{\{p\}q_\perp\{p \wedge q\}}\ \text{Assertion}$$

$$\frac{\{b \wedge p\}c_1\{q\} \qquad \{\neg b \wedge p\}c_2\{s\} \qquad q \longrightarrow A \qquad s \longrightarrow A \qquad \{A\}c_3\{A'\}}{\{p\}\text{if } b \text{ then } c_1 \text{ else } c_2 \text{ fi}; A_\perp; c_3\{A'\}}\ \text{Cond with Assert}$$

$$\frac{\{b \wedge p\}c_1\{q\} \qquad \{\neg b \wedge p\}c_2\{s\} \qquad q \longrightarrow A \qquad s \longrightarrow A}{\{p\}\text{if } b \text{ then } c_1 \text{ else } c_2 \text{ fi}; A_\perp\{A\}}\ \text{Final Cond with Assert}$$

This methodology is somewhat different from other systems wherein the annotations are a necessary element of the syntax, such as in the Isabelle implementation of the Owicki-Gries method or our while loop formulation. Strictly speaking, that methodology forces users to provide the necessary annotations when porting the code to be verified, but our VCG setup, discussed in section 3.4.2, is such that the VCG will stop at any point where it cannot discharge any more VCs and our ultimate plan is for the code to be automatically transpiled from its original language with a verified source-to-source compiler, in which case users may not have the right invariants initially if they are working with legacy code.

Others

These rules allow greater flexibility in rule application, and the rule of postcondition weakening in particular is a requirement to start our VCG.

$$\frac{p \longrightarrow p' \qquad \{p'\}c\{q\}}{\{p\}c\{q\}} \text{ Precondition Strengthening}$$

$$\frac{\{p\}c\{q'\} \qquad q' \longrightarrow q}{\{p\}c\{q\}} \text{ Postcondition Weakening}$$

$$\frac{p \longrightarrow p' \qquad \{p'\}c\{q'\} \qquad q' \longrightarrow q}{\{p\}c\{q\}} \text{ Consequence}$$

$$\frac{\{p\}c\{r\} \qquad \{p\}c\{s\}}{\{p\}c\{r \wedge s\}} \text{ Specification conjunction}$$

3.4.2 Simp Rules

Currently, the USIMPL VCG is structured as shown in listing 3.4; separate methods are used for modularity and ease of experimentation, as well as a clearer picture of how the VCG is composed. Note the usage of `drule`; some of the generated VCs require destruction rules rather than introduction rules. That aside, the general methodology is shown in algorithm 3.1. When, in the process of a proof, the VCG fails to discharge all VCs, You can examine the goal state at the point of failure and determine what must be proved manually for the VCG to continue. This proof can be done separately and added to either `vcg_simps` or `vcg_dests`, depending on what type of rule it is, at which point the VCG should be able to discharge the subgoal it previously failed on and continue to the next goal. If the pre- and postconditions are modular enough, this allows development of libraries of potentially-reusable proof goal lemmas, which may be used to lessen proof development for future work.

```
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 3.4: VCG Methods

Algorithm 3.1 VCG

```
1: procedure VCG
2:     Reorder sequencing if necessary.            ▷ Isabelle/UTP’s is geared towards WP
3:     Weaken the postcondition.
4:     while goals remain do
5:         Apply a Hoare rule.
6:         If Hoare rule application fails, try discharging the current goal by assumption
           application or via Isabelle/UTP simp (to drop to HOL) followed by application of existing
           VC lemmas.
7:     end while
8: end procedure
```

Chapter 4

Case Studies

As algorithms with real-world applications that are not overly-complicated to reason about yet still have significant depth in their usage of iteration and/or recursion, sorting algorithms function as nigh-perfect introductory case studies for formal verification. For these reasons, I selected two sorting algorithms to test the USIMPL VCG on: one more simple (insertion sort) and one more complex (quicksort). However, USIMPL does not currently have a complete heap model, so types that would be arrays in languages like C are instead handled as Isabelle/HOL lists. This does mean that we have some additional guarantees C programmers do not (we can determine the length of a list directly and use other library functions such as `take`, `drop`, and `set`), but may introduce complexity in developing the proof framework due to adding additional abstraction that would require proving equivalence between the list and array representations.

You may also notice that, unlike much of the notation used in chapters 2 and 3, the notation in the listings in this chapter directly matches that actually used by Isabelle/UTP and USIMPL; that is due to this this chapter covering practical usage of the notation rather than giving a general insight into its meaning. Additionally, many, but not all, of the proofs shown in this chapter rely on the additional theories described in appendix B.3 due to the usage of custom definitions such as `swap`, `swap_at`, and `slice`. This is a typical methodology when using definitions in Isabelle, as the alternative is to unfold all such definitions and prove the necessary concepts inline, which can make for messy proofs as illustrated in listing 4.9.

4.1 Insertion Sort

An algorithm that provides a nice balance of simplicity and efficiency, insertion sort [58] is commonly taught in many introductory algorithms courses. Though not scalable to large collections of data, insertion sort is widely used for smaller data sets and as a component in more complex sorting algorithms that divide large lists into small fragments; thus, it served

```
1 i :== 1;;
2 while &i <ᵤ #ᵤ(&array)
3     j :== &i;;
4     (while &j >ᵤ 0 ∧ &array(&j - 1)ₐ >ᵤ &array(&j)ₐ
5         array :== swap_atᵤ (&j) (&array);;
6         j :== (&j - 1)
7     od);;
8     i :== (&i + 1)
9 od
```

Listing 4.1: Insertion Sort in Isabelle/UTP

as a nice starting point for testing USIMPL on more complex algorithms.

As shown in algorithm 4.2, which sorts a list in place in ascending order, this sort is typically implemented using a pair of loops, one nested in the other. The outer loop is used to keep track of all the elements in the list that are known to be sorted before execution of the inner loop, (that is, everything below i is sorted with respect to that sublist), while the inner loop then moves the next element down through the list until it is in a sorted position. Once complete, all elements in the list are sorted.

Algorithm 4.2 Insertion Sort

```
function INSERTIONSORT($A$)
    $i \leftarrow 1$
    while $i < \text{LENGTH}(A)$ do
        $j \leftarrow i$
        while $j > 0 \wedge A[j-1] > A[j]$ do
            SWAP($A[j], A[j-1]$)
            $j \leftarrow j - 1$
        end while
        $i \leftarrow i + 1$
    end while
end function
```

Using Isabelle/UTP notation (described in section 2.4.2) and USIMPL notation for loops, the code looks like listing 4.1; due to type inference, no variables needed explicit types. Also note the $_u$ appended to `swap_at` on line 5; for our purposes, this indicates an HOL function that has been lifted to UTP and is thus usable with Isabelle/UTP expressions.

4.1.1 Proof Setup

In general, the insertion sort algorithm does not require any particular preconditions other than some basic lens laws (refer back to section 2.4.1) and type constraints (which in our case are satisfied by type restrictions of the syntax); however, the postcondition complicates things a bit. There are two main properties that must be proved about an insertion sort algorithm[1]:

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

By typical UTP logic, we could compare the input and output variables for the algorithm to check the second property; unfortunately, Isabelle/UTP does not allow such usage easily in proofs due to typing issues with its expressions. Thus, the easiest method was to establish an auxiliary logical variable that represents the list to be sorted in the precondition and then use that in the postcondition; this is also useful for composition of proofs later on as, when used in context, the denotational semantics of UTP do not allow reasoning about intermediate state. The full lemma used to prove the correctness of insertion sort is shown in listing 4.2; further explanation follows.

4.1.2 Invariants

Because the insertion sort algorithm has loops, it must also have invariants (section 2.3), as you can see from lines 10 and 13. For modularity and ease of proving, the invariants were extracted and represented as Isabelle/HOL definitions that were lifted to Isabelle/UTP expressions (shown in listings 4.3 and 4.4). This allowed us to formulate lemmas about the invariant interactions strictly on the HOL level, which is quite nice as it makes reasoning easier for those used to Isabelle/HOL syntax and allows usage of more Isabelle/HOL constructs such as `let...in...` for locally binding variables.

For insertion sort in particular, the outer invariant was relatively simple; it essentially establishes the statement from before, that the outer loop keeps track of what is currently sorted (as well as that the outer loop maintains the contents of the list). This information must be carried through in order to discharge the algorithm's postcondition (which in fact becomes trivial, as when the algorithm finishes you have `i = length array` and thus obtain `sorted (take (length xs) xs) = sorted xs`). The trouble comes when the inner invariant is involved, as that requires a bit more proving (though not as much as other proofs for the

---

[1] Strictly speaking, as insertion sort is *stable*, there is a third property: that equal elements in the list maintain the same relative order in the final list. That is mainly useful for types that can be sorted in different ways (e.g. records that can be sorted using individual fields as keys, such as sorting first by family name and then by given name) and not so much for simple one-key sorting so was not considered for this example.

```
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 4.2: 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 4.3: 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 4.4: Insertion Sort Inner Invariant

```
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 4.5: 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›
  using assms unfolding inner_invr_def outer_invr_def Let_def
  apply (erule_tac disjE1)
   apply auto
   apply (metis Cons_nth_drop_Suc Suc_leI drop_0 length_greater_0_conv
    length_take less_imp_le min.absorb2 nth_take sorted.simps)
  apply (drule (1) insert_with_sorted)
    apply auto
   apply (smt One_nat_def diff_Suc_less last_conv_nth le_less_trans length_take
    list.size(3) min.absorb2 not_le_imp_less not_less_iff_gr_or_eq nth_take)
  using take_take[symmetric, where n = j and m = ‹Suc i› and xs = array]
    id_take_nth_drop[where xs = ‹take (Suc i) array› and i = j]
  by (auto simp: min_def)
```

Listing 4.6: Insertion Sort Outer Invariant Step Condition

inner invariant). The lemmas required to satisfy the verification conditions generated for the outer loop are shown in listings 4.5 to 4.7.

As noted already, proofs for the inner invariant are more complicated due to the increased complexity of the inner invariant and the nature of the inner loop code. As the inner loop essentially moves an element down through the sorted portion until the element is in its sorted position, the invariant had to be formulated to describe that formally. This invariant essentially ensures that, for each iteration of the inner loop, the previously-sorted portion remains sorted except for the element being moved down and that element is less than everything above it in the sorted section. The lemmas needed for the verification conditions generated for the inner loop are shown in listings 4.8 and 4.9; there is no lemma for the final state of the inner loop as that is already handled by listing 4.6. You may observe that the proof for the inner invariant step lemma has a lot more body to it than the other lemmas shown here; this is in large part because I did not develop a useful library of simplification lemmas and properties for expressions involving `swap_at`.

Listing 4.9: Insertion Sort Inner 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 4.7: 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 4.8: 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›
  using assms unfolding inner_invr_def Let_def
  apply clarsimp
  apply (safe; (simp add: swap_at_def; fail)?)
proof goal_cases
  case 1
  then show ?case by (simp add: swap_at_def Multiset.mset_swap)
next
  assume 2: ‹0 < j›
    ‹array!j < array!(j - Suc 0)›
    ‹i < length array›
    ‹j ≤ i›
    ‹mset array = mset old_array›
    ‹sorted (take j array @ drop (Suc j) (take (Suc i) array))›
    ‹∀x ∈ set (drop (Suc j) (take (Suc i) array)). array!j < x›
  define xs₁ where ‹xs₁ = take j array›
  define xs₂ where ‹xs₂ = drop (Suc j) (take (Suc i) array)›
  define x where ‹x = array ! j›
  obtain xs₁' y where xs_last: ‹xs₁ = xs₁' @ [y]›
    unfolding xs₁_def using 2
```

```
    by (metis Suc_pred diff_le_self le_less_trans take_hd_drop)
  have xs_butlast: ‹xs₁' = take (j - Suc 0) array›
    by (smt 2(3) 2(4) Suc_pred append1_eq_conv assms(2) diff_le_self
   le_less_trans take_hd_drop xs₁_def xs_last)
  have y: ‹y = array ! (j - Suc 0)›
    by (metis (no_types, lifting) 2(3) 2(4) Cons_nth_drop_Suc One_nat_def
   Suc_pred assms(2) diff_le_self le_less_trans list.sel(1) nth_append_length
   take_hd_drop xs₁_def xs_butlast xs_last)
  have xs₁'_is_aaker: ‹xs₁' = take (j - Suc 0) (swap_at j array)›
    by (simp add: swap_at_def xs_butlast)
  have y_concat_xs₂: ‹y # xs₂ = drop j (take (Suc i) (swap_at j array))›
    using ‹j > 0›
    apply (auto simp: swap_at_def drop_take list_update_swap)
    by (smt 2(3) 2(4) Cons_nth_drop_Suc Suc_diff_Suc drop_take drop_update_cancel
    le_less_trans length_list_update lessI nth_list_update_eq take_Suc_Cons xs₂
   _def y)
  from 2 show ‹sorted (take (j - Suc 0) (swap_at j array) @ drop j (take (Suc i)
   (swap_at j array)))›
    by (fold xs₁_def xs₂_def xs_butlast xs₁'_is_aaker y_concat_xs₂) (simp add:
   xs_last)
  {
    fix x
    assume ‹x ∈ set (drop j (take (Suc i) (swap_at j array)))›
    show ‹swap_at j array!(j - Suc 0) < x›
      by (smt 2(2) 2(3) 2(4) 2(7) One_nat_def ‹x ∈ set (drop j (take (Suc i) (
   swap_at j array)))› diff_le_self le_less_trans length_list_update
   nth_list_update_eq set_ConsD swap_at_def xs₂_def y y_concat_xs₂)[2]
  }
qed
```

4.2 Quicksort

Developed by none other than Tony Hoare in the early 1960s [45], quicksort has become an ubiquitous sorting algorithm implemented in many forms and present in the standard libraries of languages such as C, C++, and Java. Algorithm 4.3 is based on the Lomuto partitioning scheme [7, 18], which is less efficient than Hoare's original method and modern improvements but is probably the easiest to reason about. It is not exactly the same as Lomuto's because some initial values were adjusted due to Isabelle/HOL typing constraints, as well as removal of a minor optimization that is not strictly required to reduce the amount of proving needed.

---

[2] Another usage of SMT

Algorithm 4.3 Lomuto-style Quicksort

```
function QUICKSORT($A, lo, hi$)
    if $lo < hi$ then
        $p \leftarrow$ PARTITION($A, lo, hi$)
        QUICKSORT($A, lo, p - 1$)
        QUICKSORT($A, p + 1, hi$)
    end if
end function

function PARTITION($A, lo, hi$)
    $pivot \leftarrow A[hi]$
    $i \leftarrow lo$
    $j \leftarrow lo$
    for $j \leftarrow lo, hi - 1$ do                                        ▷ inclusive
        if $A[j] < pivot$ then
            SWAP($i, j, A$)
            $i \leftarrow i + 1$
        end if
    end for
    SWAP($i, hi, A$)                       ▷ $pivot$ is always less than or equal to $A[i]$
    return $i$
end function
```

```
pivot :== &A(hi)_a;;
i :== lo;;
j :== lo;;
(while &j <_u hi do
    (if_u &A(&j)_a <_u &pivot then
        A :== swap_u (&i) (&j) (&A);;
        i :== (&i + 1)
    else II);;
    j :== (&j + 1)
od);;
A :== swap_u (&i) hi (&A)
```

Listing 4.10: Quicksort Partition in Isabelle/UTP

Unfortunately, the quicksort proof in this thesis is restricted to a proof of the body of the partition function due to time constraints; the (anti)framing reasoning already existing in Isabelle/UTP was not enough and we did not develop good reasoning, nor further reasoning about lenses with framing, by the time of writing of this thesis. We also did not get far in handling recursion and the necessary invariants. Even so, the partitioning algorithm in its own right has a good amount of algorithmic complexity and the proof is thus detailed below.

4.2.1 Proof Setup

As with insertion sort, the quicksort partition requires usage of an auxiliary variable to represent the list before algorithm execution. On top of that, there are some additional expressions to work with: `lo` and `hi`. As these 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.

```
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 4.11: Proof of Quicksort Partition Correctness

```
1  definition ‹qs_partition_invr A oldA lo hi i j pivot ≡
2    mset (slice lo (Suc hi) A) = mset (slice lo (Suc hi) oldA)
3  ∧ take lo A = take lo oldA
4  ∧ drop (Suc hi) A = drop (Suc hi) oldA
5  ∧ lo ≤ i
6  ∧ i ≤ j
7  ∧ j ≤ hi
8  ∧ hi < length A
9  ∧ (∀x ∈ set (slice lo i A). x ≤ pivot)
10 ∧ (∀x ∈ set (slice i j A). pivot ≤ x)
11 ∧ pivot = A!hi
12 ›
13 abbreviation ‹qs_partition_invr_u ≡ sepop qs_partition_invr›
```

Listing 4.12: Quicksort Partition Invariant

4.2.2 Invariant

While the partition invariant in listing 4.12 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 4.13, 4.14, 4.16, and 4.18 were what was developed to resolve the generated verification conditions for the quicksort partition loop. Note in particular listings 4.14 and 4.16; 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), and in fact the `if`-statement annotation mentioned in section 3.4.1 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 simplication of expressions involving `swap` and `slice`, which resulted in cleaner proofs for the invariant-related lemmas and can be found in appendix B; other subproofs were also extracted and proved separately (listings 4.15 and 4.17).

```
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 4.13: Quicksort Partition Invariant Initial Condition

```
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›
  using assms unfolding qs_partition_invr_def swap_def
  apply auto
  subgoal
    using Multiset.mset_swap[of ‹j - lo› ‹slice lo (Suc hi) A› ‹i - lo›]
    by (simp add: slice_update_extract)
  subgoal for x
    by (cases ‹i = j›) (auto simp: slice_suc2_eq)
  subgoal for x
    apply (cases ‹i = j›)
     apply (auto simp: slice_set_conv_nth nth_list_update)
     apply fastforce
    by (metis Suc_leD less_antisym less_trans)
  done
```

Listing 4.14: 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 4.15: 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 4.16: 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 4.17: 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 4.18: Quicksort Partition Invariant Final Condition

Chapter 5

Conclusions

Despite USIMPL's incomplete state, the case studies in chapter 4 show that it can already be used for verification of algorithms that do not include procedure calls or scoping and that the goal of developing a modular library of reusable VC proofs does appear to be achievable. Thus, the contributions described in this thesis serve as a worthwhile foundation for the theorem-based proving component of the Orca project. To reiterate, those contributions include extensions to the Isabelle/UTP implementation of Hoare and He's UTP in the proof assistant Isabelle with features of the Simpl language presented by Schirmer as well as development of additional algebraic laws for Isabelle/UTP language constructs. It would also be inappropriate to leave out a mention of my main contribution to the development of USIMPL, the SP VCG.

5.1 Lessons Learned

There are a few things that can be observed from the work that was done from this thesis that can be used to direct future progress. For example, due to Isabelle's reliance on strictly conservative proofs and a lack of methodology for easily generating helper simplification lemmas based on combinations of definitions, libraries for helper lemmas beyond the auxiliary lemmas needed to prove VCs ended up being a more significant component than expected when developing the case studies. This is most obvious when viewing some of the insertion sort auxiliary lemmas, which were developed without the intent of other helpers; once a helper-library approach was adopted for quicksort proving, the auxiliary lemmas became cleaner and easier to prove. Such an approach does require thinking about the proofs in terms of transformations between equivalent representations, which can be a pain, but that is an unavoidable aspect of using a simplification-based term-rewriting tool like Isabelle.

A less obvious but still important aspect is that, for loops and other control flow structures with complex behavior, development of invariants and assertions to continue the flow of proving can be difficult and tedious. This is most obviously shown by the need to carry through trivial information about index variables and how their values relate to each other, but also by the requirement for carry-through of basic conditions such as sorting operations maintaining the contents of their list. Development of formalized pre- and postconditions can also be tricky, and the specific level of detail required depending on how users want to use the algorithms in relation to others can be hard to determine. This can be observed with insertion sort as well; as mentioned in section 4.1.1, insertion sort is stable, and so one could formalize that in an additional component of the postcondition if desired (which could be a complicated procedure). Unfortunately, doing so would increase the complexity of the proof beyond just the specification, as the intermediate components of the proof would then have to show that stability is maintained during the inner and outer loops. One possible methodology for that would be to extract the stability statement as a predicate and then build up another set of helper/auxiliary lemmas that show how stability is maintained given various list operations; as with the auxiliary lemmas for VCs, such an approach is quite open-ended and auxiliary development would be continuous as more algorithms that utilize reasoning about stable sorting are verified.

Now that some of the issues with our approach have been covered, the following sections go into a bit more detail on where to go with the knowledge gained from development of USIMPL and what may eventually be doable with it.

5.2 Connections to the Wider World

In all, USIMPL seems like a viable tool for software verification; with the development of more automated methods (further discussed in section 5.3), it would be a useful component for application to systems that require a high degree of reliability, ideally usable even by those without much experience in formal methods. The requirement for development of additional lemmas to discharge generated VCs, as well as the requirement for development of invariants for loops/recursion and the need to formalize preconditions and postconditions, does pose a significant problem for this, particularly due to some of the issues brought up in section 5.1, as many conventional programmers these days do not have experience writing functional code or formulating their algorithms in an explicitly mathematical representation; however, building libraries of common conditions for algorithms used in the wild (discussed previously) may help alleviate such issues to a degree, and future invariant work is again detailed in section 5.3.

As previously discussed in section 2.6.2, Dafny provides another method of doing this by directly integrating pre-/postconditions and invariants into the language with which programs are written; however, Dafny presents a higher-level language and as such lower-level elements of C (such as byte-level memory accesses and the like) would have to be modeled anyway. That fact, combined with the fact that Dafny is not itself verified, makes Dafny not as suitable for the kind of work we are trying to accomplish with USIMPL and ultimately Orca as, while we still have to model such lower-level properties, we do so in a verified manner and thus, assuming the Isabelle kernel is correct, we can trust our results to also be correct.

5.3 Future Work

As mentioned in sections 1.2, 3.3, and 4.2 and above, scoping support, recursive function calls, and a heap-style memory model all remain to be properly implemented in USIMPL. We also plan on proceeding with testing and, if necessary, improving handling of total correctness, abrupt termination, and side-affecting expressions (previously mentioned in chapter 3). The scoping support may be handled either by development of proper Hoare rules for framing or perhaps by implementation of separation logic (last mentioned in section 3.3). We also plan on fleshing out the SP proofs we currently have to further match the lemmas existing in Isabelle/UTP for WP reasoning and, when better-refined, to convert the current Eisbach VCG methods into SML code to improve performance. Additional algebraic laws are also in development, as mentioned in the corresponding extension section.

Sections 4.1.2 and 4.2.2 also show how important invariants are even for seemingly simple algorithms; most tools still rely on manual/user-provided invariants and annotations, so development of tools for automatic generation of invariants when possible would greatly enhance usability. A very primitive example of such behavior is how LLVM's scalar evolution analyses can determine loop trip counts when the bounds are loop invariant [70]; more complex analysis is provided by the CoRnucopia of ABstractions (CRAB)[1] library for abstract interpretation [19, 20] of computer programs, which is used by the SeaHorn[2] static analysis tool [42, 43], but even that is quite limited in its ability to generate algebraic invariants.

The ability to generate at least some invariants would also be useful when working with legacy software, especially with the development of verified source-to-source compilers (*transpilers*, previously mentioned in section 3.4.1) to enable automation of the initial steps of verification (i.e. getting the code in a verifiable form). The current plan is for the transpiler(s) to be integrated with USIMPL, and as such they would most likely be written in SML, with the current target language being C as that is what the Linux kernel and many other large codebases for systems software are written in (and other higher-level languages may include features that are harder to represent in USIMPL).

---

[1] `https://github.com/seahorn/crab`

[2] `https://seahorn.github.io/`

Bibliography

[1] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994. ISSN 0304-3975. doi: 10.1016/0304-3975(94)90010-8.

[2] Paul E. Ammann, Paul E. Black, and Wei Ding. Model checkers in software testing. Technical report, NIST-IR 6777, National Institute of Standards and Technology, 2002.

[3] Apple Inc. About the security content of security update 2017-001 - Apple Support, November 2017. URL `https://support.apple.com/en-us/HT208315`. Accessed November 30, 2017.

[4] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *4th International Symposium on Formal Methods for Components and Objects, Revised Lectures*, FMCO 2005, pages 364–387. Springer, November 2005.

[5] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices: International Workshop, CASSIS 2004, Revised Selected Papers*, volume 3362 of *Lecture Notes on Computer Science*, pages 49–69. Springer Berlin Heidelberg, Berlin, Heidelberg, March 2004. ISBN 978-3-540-30569-9. doi: 10.1007/978-3-540-30569-9_3.

[6] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. *Handbook of Satisfiability*, 185:825–885, 2009.

[7] Jon Bentley. *Programming Pearls*. ACM, New York, NY, USA, 1986. ISBN 0-201-50019-1.

[8] Gilles Bernot, Marie-Claude Gaudel, and Bruno Marre. A formal approach to software testing. In *Algebraic Methodology and Software Technology*, AMAST '91, pages 234–253, May 1991.

[9] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer-Verlag Berlin Heidelberg, first edition, 2004. doi: 10.1007/978-3-662-07964-5.

[10] Per Bjesse. What is formal verification? *ACM SIGDA Newsletter*, 35(24), December 2005. ISSN 0163-5743. doi: 10.1145/1113792.1113794.

[11] Michael Blair, Sally Obenski, and Paula Bridickas. Patriot missile defense: Software problem led to system failure at dhahran, saudi arabia. Technical report, United States General Accounting Office: Information Management and Technology Division, February 1992. URL `http://www.gao.gov/products/IMTEC-92-26`.

[12] Max Böhm and Ewald Speckenmeyer. A fast parallel SAT-solver — efficient workload balancing. *Annals of Mathematics and Artificial Intelligence*, 17(2):381–400, 1996. ISSN 1573-7470. doi: 10.1007/BF02127976.

[13] Achim D. Brucker and Burkhart Wolff. On theorem prover-based testing. *Formal Aspects of Computing*, 25(5):683–721, 2013. ISSN 0934-5043. doi: 10.1007/s00165-012-0222-y. URL `https://www.brucker.ch/bibliography/abstract/brucker.ea-theorem-prover-2012`.

[14] Ricky W. Butler. What is formal methods?, April 2016. URL `https://shemesh.larc.nasa.gov/fm/fm-what.html`. Accessed November 1, 2017.

[15] Chin-Liang Chang and Richard Char-Tung Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, 1969.

[16] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Progress on the state explosion problem in model checking. In Reinhard Wilhelm, editor, *Informatics: 10 Years Back, 10 Years Ahead*, pages 176–194. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001. ISBN 978-3-540-44577-7. doi: 10.1007/3-540-44577-3_12.

[17] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, New York, NY, USA, May 1971. ACM. doi: 10.1145/800157.805047.

[18] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, chapter Quicksort, pages 170–190. MIT Press and McGraw-Hill, third edition, 2009. ISBN 0-262-03384-4.

[19] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252, New York, NY, USA, January 1977. ACM. doi: 10.1145/512950.512973.

[20] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '79, pages 269–282, New York, NY, USA, January 1979. ACM. doi: 10.1145/567752.567778.

[21] Judith Crow and Ben Di Vito. Formalizing space shuttle software requirements: Four case studies. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 7(3):296–332, July 1998.

[22] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, July 1962. ISSN 0001-0782. doi: 10.1145/368273.368557.

[23] Leonardo de Moura, Bruno Dutertre, and Natarajan Shankar. A tutorial on satisfiability modulo theories. In *Computer-Aided Verification (CAV'2007)*, volume 4590 of *Lecture Notes in Computer Science*, pages 20–36. Springer-Verlag, July 2007.

[24] Edsger Wybe Dijkstra. *A Discipline of Programming*. Prentice-Hall Series in Automatic Computation. Prentice-Hall, Englewood Cliffs, NJ, October 1976. ISBN 978-0-132-15871-8.

[25] Bruno Dutertre. Yices 2.2. In Armin Biere and Roderick Bloem, editors, *Computer-Aided Verification (CAV'2014)*, volume 8559 of *Lecture Notes in Computer Science*, pages 737–744. Springer, July 2014.

[26] Bruno Dutertre and Leonardo de Moura. The YICES SMT solver. Technical Report 2, SRI International, 2006. URL `http://yices.csl.sri.com/papers/tool-paper.pdf`.

[27] Bruno Dutertre and Maria Sorea. Modeling and verification of a fault-tolerant real-time startup protocol using calendar automata. In Yassine Lakhnech and Sergio Yovine, editors, *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems (FORMATS 2004)/Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT 2004) Proceedings*, volume 3253 of *Lecture Notes in Computer Science*, pages 199–214, Grenoble, France, September 2004. Springer Berlin Heidelberg. ISBN 978-3-540-30206-3. doi: 10.1007/978-3-540-30206-3_15.

[28] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing: 6th International Conference (SAT 2003), Selected Revised Papers*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518, Santa Margherita Ligure, Italy, 2003. Springer Berlin Heidelberg. ISBN 978-3-540-24605-3. doi: 10.1007/978-3-540-24605-3_37.

[29] Robert W. Floyd. Assigning meanings to programs. In *Proceedings of the Symposia in Applied Mathematics*, volume 19, pages 19–32, Providence, Rhode Island, 1967. American Mathematical Society.

[30] Simon Foster and Jim Woodcock. Unifying theories of programming in Isabelle. In Zhiming Liu, Jim Woodcock, and Huibiao Zhu, editors, *Unifying Theories of Programming and Formal Engineering Methods: International Training School on Software Engineering, Held at ICTAC 2013*, volume 8050 of *Lecture Notes in Computer Science*, pages 109–155. Springer Berlin Heidelberg, August 2013. ISBN 978-3-642-39721-9. doi: 10.1007/978-3-642-39721-9_3.

[31] Simon Foster, Frank Zeyda, and Jim Woodcock. Isabelle/UTP: A mechanised theory engineering framework. In David Naumann, editor, *Unifying Theories of Programming: 5th International Symposium, UTP 2014, Singapore, May 13, 2014, Revised Selected Papers*, volume 8963 of *Lecture Notes in Computer Science*, pages 21–41. Springer International Publishing, Cham, 2015. ISBN 978-3-319-14806-9. doi: 10.1007/978-3-319-14806-9_2.

[32] Simon Foster, Frank Zeyda, and Jim Woodcock. Unifying heterogeneous state-spaces with lenses. In *International Colloquium on Theoretical Aspects of Computing*, pages 295–314. Springer International Publishing, October 2016.

[33] Gordon Fraser, Franz Wotawa, and Paul E. Ammann. Testing with model checkers: A survey. *Software Testing, Verification and Reliability*, 19(3):215–261, December 2008. ISSN 1099-1689. doi: 10.1002/stvr.402.

[34] Jon William Freeman. *Improvements to Propositional Satisfiability Search Algorithms*. PhD thesis, University of Pennsylvania, 1995.

[35] Marie-Claude Gaudel. Testing can be formal, too. In Peter D. Mosses, Mogens Nielsen, and Michael I. Schwartzbach, editors, *Theory and Practice of Software Development: 6th International Joint Conference*, TAPSOFT '95, pages 82–96, Berlin, Heidelberg, May 1995. Springer Berlin Heidelberg. ISBN 978-3-540-49233-7. doi: 10.1007/3-540-59293-8_188.

[36] Gerhard Gentzen. Untersuchungen über das logische Schließen. I. *Mathematische Zeitschrift*, 39(1):176–210, 1935.

[37] Joseph Goguen and Grant Malcolm. *Algebraic Semantics of Imperative Programs*. MIT press, 1996.

[38] Carla P. Gomes, Henry Kautz, Ashish Sabharwal, and Bart Selman. Satisfiability solvers. In Frank van Harmelen, Vladimir Lifschitz, and Bruce Porter, editors, *Handbook of Knowledge Representation*, volume 3 of *Foundations of Artificial Intelligence*, chapter 2, pages 89–134. Elsevier, 2008. doi: 10.1016/S1574-6526(07)03002-7.

[39] Michael John Caldwell Gordon. HOL: A proof generating system for higher-order logic. In Graham Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 73–128. Springer US, Boston, MA, 1988. ISBN 978-1-4613-2007-4. doi: 10.1007/978-1-4613-2007-4_3.

[40] Michael John Caldwell Gordon. Background reading on Hoare logic, April 2012.

[41] Michael John Caldwell Gordon and Hélène Collavizza. Forward with Hoare. In A. W. Roscoe, Cliff B. Jones, and Kenneth R. Wood, editors, *Reflections on the Work of C.A.R. Hoare*, pages 101–121. Springer London, London, 2010. ISBN 978-1-84882-912-1. doi: 10.1007/978-1-84882-912-1_5.

[42] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. The SeaHorn verification framework. In *International Conference on Computer-Aided Verification*, pages 343–361. Springer, 2015.

[43] Arie Gurfinkel, Temesghen Kahsai, and Jorge A. Navas. SeaHorn: A framework for verifying C programs (competition contribution). In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 447–450. Springer, 2015.

[44] Marijn J. H. Heule and Hans Van Maaren. *Look-ahead based SAT solvers*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 155–184. IOS Press, Netherlands, 1 edition, 2009. ISBN 9781586039295. doi: 10.3233/978-1-58603-929-5-155.

[45] Charles Antony Richard Hoare. Quicksort. *The Computer Journal*, 5(1):10–16, January 1962. doi: 10.1093/comjnl/5.1.10.

[46] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969. ISSN 0001-0782. doi: 10.1145/363235.363259.

[47] Charles Antony Richard Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice Hall Englewood Cliffs, 1998. ISBN 0-13-458761-8.

[48] Charles Antony Richard Hoare, Ian James Hayes, He Jifeng, Carroll C. Morgan, Andrew William Roscoe, Jeff W. Sanders, Ib Holm Sørensen, J. Michael Spivey, and Bernard A. Sufrin. Laws of programming. *Communications of the ACM*, 30(8):672–686, August 1987. ISSN 0001-0782. doi: 10.1145/27651.27653.

[49] Jieh Hsiang, Hélène Kirchner, Pierre Lescanne, and Michaël Rusinowitch. The term rewriting approach to automated theorem proving. *The Journal of Logic Programming*, 14(1–2):71–99, October 1992. ISSN 0743-1066. doi: 10.1016/0743-1066(92)90047-7.

[50] Stanisław Jaśkowski. On the rules of suppositions in formal logic, 1934.

[51] Gilles Kahn. Natural semantics. In Franz J. Brandenburg, Guy Vidal-Naquet, and Martin Wirsing, editors, *4th Annual Symposium on Theoretical Aspects of Computer Science*, STACS 87, pages 22–39, Berlin, February 1987. Springer-Verlag. ISBN 978-3-540-47419-7. doi: 10.1007/BFb0039592.

[52] Ioannis T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *14th International Symposium on Formal Methods*, FM 2006, pages 268–283, Berlin, Heidelberg, August 2006. Springer Berlin Heidelberg. ISBN 978-3-540-37216-5. doi: 10.1007/11813040_19.

[53] Ioannis T. Kassios. *A Theory of Object Oriented Refinement*. PhD thesis, University of Toronto, 2006.

[54] Ioannis T. Kassios. The dynamic frames theory. *Formal Aspects of Computing*, 23(3): 267–288, 2011. ISSN 1433-299X. doi: 10.1007/s00165-010-0152-5.

[55] Hadi Katebi, Karem A. Sakallah, and João Paulo Marques Silva. Empirical study of the anatomy of modern SAT solvers. *Theory and Applications of Satisfiability Testing (SAT 2011)*, pages 343–356, 2011.

[56] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-752-3. doi: 10.1145/1629575.1629596. URL `https://sel4.systems/`.

[57] Gerwin Klein, Tobias Nipkow, Larry Paulson, and René Thiemann. Archive of formal proofs, November 2017. URL `https://www.isa-afp.org/submitting.html`. Accessed November 12, 2017.

[58] Donald Ervin Knuth. *The Art of Computer Programming*, volume 3: Sorting and Searching, chapter 5.2.1: Sorting by Insertion, pages 80–105. Addison-Wesley, second edition, 1998. ISBN 0-201-89685-0.

[59] Jean-Louis Lassez, V. L. Nguyen, and Elizabeth A. Sonenberg. Fixed point theorems and semantics: A folk tale. *Information Processing Letters*, 14(3):112–116, 1982. ISSN 0020-0190. doi: 10.1016/0020-0190(82)90065-5.

[60] K. Rustan M. Leino. Ecstatic: An object-oriented programming language with an axiomatic semantics. In *The Fourth International Workshop on Foundations of Object-Oriented Languages*, January 1997.

[61] K. Rustan M. Leino. This is Boogie 2, June 2008. URL `https://www.microsoft.com/en-us/research/publication/this-is-boogie-2-2/`.

[62] Xavier Leroy et al. The CompCert verified compiler: Documentation and user's manual. Technical report, INRIA Paris-Rocquencourt, 2012.

[63] Nancy G. Leveson and Clark S. Turner. An investigation of the Therac-25 accidents. *Computer*, 26(7):18–41, July 1993. ISSN 0018-9162. doi: 10.1109/MC.1993.274940.

[64] Donald W. Loveland. *Automated Theorem Proving: A Logical Basis*, volume 6 of *Fundamental Studies in Computer Science*. North-Holland Publishing, 1978.

[65] Daniel Matichuk, Makarius Wenzel, and Toby Murray. An Isabelle proof method language. In Gerwin Klein and Ruben Gamboa, editors, *Interactive Theorem Proving: 5th International Conference, Held as Part of the Vienna Summer of Logic*, ITP 2014, VSL 2014, pages 390–405, Cham, July 2014. Springer International Publishing. ISBN 978-3-319-08970-6. doi: 10.1007/978-3-319-08970-6_25.

[66] Daniel Matichuk, Toby Murray, and Makarius Wenzel. Eisbach: A proof method language for Isabelle. *Journal of Automated Reasoning*, 56(3):261–282, March 2016. ISSN 1573-0670. doi: 10.1007/s10817-015-9360-2.

[67] John McCarthy and Patrick J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. *Readings in Artificial Intelligence*, pages 431–450, 1969.

[68] Robin Milner. Logic for computable functions: Description of a machine implementation. Technical report, Stanford University, Stanford, CA, USA, 1972.

[69] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT press, 1997.

[70] n/a. LLVM: llvm::ScalarEvolution class reference, November 2017. URL `https://llvm.org/doxygen/classllvm_1_1ScalarEvolution.html`. Accessed November 13, 2017.

[71] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, 2006.

[72] Tobias Nipkow. *Programming and Proving in Isabelle/HOL*, October 2017. URL `https://isabelle.in.tum.de/dist/Isabelle2017/doc/prog-prove.pdf`.

[73] Tobias Nipkow, Markus M. Wenzel, and Laurence C. Paulson. Session HOL (Isabelle2017: October 2017), October 2017. URL `http://isabelle.in.tum.de/dist/library/HOL/HOL/index.html`. Accessed November 19, 2017.

[74] Marcel Oliveira, Ana Cavalcanti, and Jim Woodcock. A UTP semantics for Circus. *Formal Aspects of Computing*, 21(1–2):3–32, February 2009. ISSN 1433-299X. doi: 10.1007/s00165-007-0052-5.

[75] Marcel Vinicius Medeiros Oliveira. *Formal Derivation of State-Rich Reactive Programs Using Circus*. PhD thesis, University of York, Department of Computer Science, 2005.

[76] Marcel Vinicius Medeiros Oliveira, Ana Cavalcanti, and Jim Woodcock. A denotational semantics for Circus. *Electronic Notes in Theoretical Computer Science*, 187:107–123, July 2007. ISSN 1571-0661. doi: 10.1016/j.entcs.2006.08.047.

[77] Susan Owicki. A consistent and complete deductive system for the verification of parallel programs. In *Proceedings of the Eighth Annual ACM Symposium on Theory of Computing*, STOC '76, pages 73–86, New York, NY, USA, 1976. ACM. doi: 10.1145/800113.803634.

[78] Sam Owre and Natarajan Shankar. Writing PVS proof strategies. In *Design and Application of Strategies/Tactics in Higher Order Logics*, STRATA 2003, pages 1–15, 2003. URL `http://pvs.csl.sri.com/papers/strata03/strata03.pdf`.

[79] Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: A prototype verification system. In *International Conference on Automated Deduction*, pages 748–752. Springer, 1992.

[80] Sam Owre, S. Rajan, John M. Rushby, Natarajan Shankar, and Mandayam K. Srivas. PVS: Combining specification, proof checking, and model checking. In Rajeev Alur and Thomas A. Henzinger, editors, *8th International Conference on Computer Aided Verification (CAV '96), Proceedings*, volume 1102 of *Lecture Notes in Computer Science*, pages 411–414. Springer Berlin Heidelberg, July 1996. ISBN 978-3-540-68599-9. doi: 10.1007/3-540-61474-5_91.

[81] Sam Owre, John M. Rushby, Natarajan Shankar, and David W. J. Stringer-Calvert. Pvs: An experience report. In Dieter Hutter, Werner Stephan, Paolo Traverso, and Markus Ullmann, editors, *International Workshop on Current Trends in Applied Formal Methods*, FM-Trends 98, pages 338–345, Berlin, Heidelberg, October 1998. Springer Berlin Heidelberg. ISBN 978-3-540-48257-4. doi: 10.1007/3-540-48257-1_24.

[82] Christos H. Papadimitriou. On selecting a satisfying truth assignment. In *Proceedings of the 32nd Annual Symposium of Foundations of Computer Science*, pages 163–169. IEEE, October 1991. doi: 10.1109/SFCS.1991.185365.

[83] Radek Pelánek. Fighting state space explosion: Review and evaluation. In Darren Cofer and Alessandro Fantechi, editors, *Formal Methods for Industrial Critical Systems: 13th International Workshop, FMICS 2008, L'Aquila, Italy, September 15-16, 2008, Revised Selected Papers*, volume 5596 of *Lecture Notes in Computer Science*, pages 37–52. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. ISBN 978-3-642-03240-0. doi: 10.1007/978-3-642-03240-0_7.

[84] Gordon D. Plotkin. *A Structural Approach to Operational Semantics.* University of Aarhus, 1981.

[85] Gordon D. Plotkin. The origins of structural operational semantics. *The Journal of Logic and Algebraic Programming*, 60:3–15, 2004. doi: 10.1016/j.jlap.2004.03.009.

[86] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74. IEEE, 2002.

[87] Norbert Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL*. PhD thesis, Technical University of Munich, 2006.

[88] Dana S. Scott. Outline of a mathematical theory of computation. Technical report, Oxford University Computing Laboratory, Programming Research Group, 1970.

[89] Dana S. Scott. A type-theoretical alternative to ISWIM, CUCH, OWHY. *Theoretical Computer Science*, 121(1):411–440, 1993. ISSN 0304-3975. doi: 10.1016/0304-3975(93)90095-B. Annotated version of the 1969 manuscript.

[90] Bart Selman, Henry A. Kautz, and Bram Cohen. Local search strategies for satisfiability testing. *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge*, 26:521–532, October 1993.

[91] Stewart Shapiro. Classical logic II: Higher order logic. In Lou Goble, editor, *The Blackwell Guide to Philosophical Logic*. Blackwell, 2001. ISBN 0-631-20693-0.

[92] Synopsys, Inc. Heartbleed bug, April 2014. URL `http://heartbleed.com/`. Accessed November 30, 2017.

[93] Andrew S. Tanenbaum, Jorrit N. Herder, and Herbert Bos. Can we make operating systems reliable and secure? *Computer*, 39(5):44–51, May 2006. ISSN 0018-9162. doi: 10.1109/MC.2006.156.

[94] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann, 2010.

[95] Adriaan van Wijngaarden, Barry James Mailloux, John Edward Lancelot Peck, Cornelis Hermanus Antonius Koster, Charles Hodgson Lindsey, Michel Sintzoff, L. G. L. T. Meertens, and R. G. Fisker, editors. *Revised Report on the Algorithmic Language ALGOL 68*. IFIP Working Group 2.1 on Algorithmic Languages and Calculi, 1968.

[96] Yakir Vizel, Georg Weissenbacher, and Sharad Malik. Boolean satisfiability solvers and their applications in model checking. *Proceedings of the IEEE*, 103(11):2021–2035, November 2015. ISSN 0018-9219. doi: 10.1109/JPROC.2015.2455034.

[97] Eric G. Wagner. Algebraic semantics. In Samson Abramsky, Dov M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 3, pages 323–393. Oxford University Press, Oxford, UK, 1994. ISBN 0-19-853762-X.

[98] Markus M. Wenzel. *Isabelle/Isar—A Versatile Environment for Human-Readable Formal Proof Documents*. PhD thesis, Technische Universität München, Universitätsbibliothek, 2002.

[99] Martin Wildmoser and Tobias Nipkow. Certifying machine code safety: Shallow versus deep embedding. In Konrad Slind, Annette Bunker, and Ganesh Gopalakrishnan, editors, *17th International Conference on Theorem Proving in Higher Order Logics*, TPHOLs 2004, pages 133–142. Springer, September 2004.

[100] Glynn Winskel. *The Formal Semantics of Programming Languages*. MIT Press, 1993.

[101] Jim Woodcock and Simon Foster. UTP by example: Designs. In Jonathan P. Bowen, Zhiming Liu, and Zili Zhang, editors, *Engineering Trustworthy Software Systems: Second International School, March 28–April 2, Tutorial Lectures*, SETSS 2016, pages 16–50. Springer International Publishing, Cham, 2016. ISBN 978-3-319-56841-6. doi: 10.1007/978-3-319-56841-6_2.

[102] Qiwen Xu, Willem-Paul de Roever, and Jifeng He. The rely-guarantee method for verifying shared variable concurrent programs. *Formal Aspects of Computing*, 9(2): 149–174, March 1997. ISSN 1433-299X. doi: 10.1007/BF01211617.

Appendices

Appendix A

Main Extension Proofs

This chapter contains the theory data from the Orca Isabelle/UTP extension that I used for my case studies as well as the SP proofs; a full session (including the theories from Isabelle/UTP used) can be found on Google Drive for now, with the `ROOT` file containing the path to the examples for the case studies as well as some other examples.

A.1 Algebraic Laws of Programming

In this section we introduce the semantic rules (algebraic laws) related to the different statements of Simpl. Our ultimate plan is to use these rules in order to optimize a given program written in our variant of Simpl before any deductive proof verification activity or formal testing.

theory *Algebraic-Laws*
  imports *../../../../Isabelle−UTP/utp/utp-urel-laws*
begin

named-theorems *symbolic-exec* **and** *symbolic-exec-assign-uop* **and** *symbolic-exec-assign-bop* **and** *symbolic-exec-assign-trop* **and** *symbolic-exec-assign-qtop* **and** *symbolic-exec-ex*

A.1.1 SKIP Laws

In this section we introduce the algebraic laws of programming related to the SKIP statement.

lemma *pre-skip-post*: $(\lceil b\rceil_{<} \wedge II) = (II \wedge \lceil b\rceil_{>})$
  by *rel-auto*

lemma *skip-var*:
  fixes $x :: (bool \Longrightarrow {}'\alpha)$
  shows $(\$x \wedge II) = (II \wedge \$x')$
  by *rel-auto*

lemma *assign-r-alt-def* [*symbolic-exec*]:
fixes $x :: ('a \Longrightarrow '\alpha)$
shows $x :== v = II[\![\lceil v \rceil_< / \$x]\!]$
by *rel-auto*

lemma *skip-r-alpha-eq*:
$II = (\$\Sigma' =_u \$\Sigma)$
by *rel-auto*

lemma *skip-r-refine-orig*:
$(p \Rightarrow p) \sqsubseteq II$
by *pred-blast*

lemma *skip-r-eq*[*simp*]: $[\![II]\!]_e\ (a,\ b) \longleftrightarrow a = b$
by *rel-auto*

lemma *skip-refine-join*:
$(p \Rightarrow q) \sqsubseteq II \longleftrightarrow `((p \sqcup II) \Rightarrow q)`$
by *pred-auto*

lemma *skip-refine-rel*:
$`(II \Rightarrow (p \Rightarrow q))` \Longrightarrow (p \Rightarrow q) \sqsubseteq II$
by *pred-auto*

lemma *skip-r-refine-pred*:
$`(p \Rightarrow q)` \Longrightarrow (\lceil p \rceil_< \Rightarrow \lceil q \rceil_>) \sqsubseteq II$
by *rel-auto*

A.1.2 Assignment Laws

In this section we introduce the algebraic laws of programming related to the assignment statement.

lemma $\&v[\![expr/v]\!] = [v \mapsto_s expr] \dagger \&v$ **..**

lemma *usubst-cancel*[*usubst*,*symbolic-exec*]:
assumes *1*:*weak-lens v*
shows $(\&v)[\![expr/v]\!] = expr$
using *1*
by *transfer′ rel-auto*

lemma *usubst-cancel-r*[*usubst*,*symbolic-exec*]:
assumes *1*:*weak-lens v*
shows $(\$v)[\![\lceil expr \rceil_< / \$v]\!] = \lceil expr \rceil_<$
using *1*

by *rel-auto*

lemma *assign-test*[*symbolic-exec*]:
assumes *1*:*mwb-lens x*
shows $(x :== «u» ;; x :== «v») = (x :== «v»)$
using *1*
by (*simp add*: *assigns-comp subst-upd-comp subst-lit usubst-upd-idem*)

lemma *assign-r-comp*[*symbolic-exec*]:
$(x :== u ;; P) = P[\![\lceil u\rceil_{<}/\$x]\!]$
by (*simp add*: *assigns-r-comp usubst*)

lemma *assign-twice*[*symbolic-exec*]:
assumes *mwb-lens x* **and** $x \sharp f$
shows $(x :== e ;; x :== f) = (x :== f)$
using *assms*
by (*simp add*: *assigns-comp usubst*)

lemma *assign-commute*:
assumes $x \bowtie y$ $x \sharp f$ $y \sharp e$
shows $(x :== e ;; y :== f) = (y :== f ;; x :== e)$
using *assms*
by (*rel-auto*, *simp-all add*: *lens-indep-comm*)

lemma *assign-cond*:
fixes $x :: ('a \Longrightarrow '\alpha)$
assumes $out\alpha \sharp b$
shows $(x :== e ;; (P \triangleleft b \triangleright Q)) =$
$((x :== e ;; P) \triangleleft b[\![\lceil e\rceil_{<}/\$x]\!] \triangleright (x :== e ;; Q))$
by *rel-auto*

lemma *assign-rcond*[*symbolic-exec*]:
fixes $x :: ('a \Longrightarrow '\alpha)$
shows $(x :== e ;; (P \triangleleft b \triangleright_r Q)) = ((x :== e ;; P) \triangleleft (b[\![e/x]\!]) \triangleright_r (x :== e ;; Q))$
by *rel-auto*

lemma *assign-uop1*[*symbolic-exec-assign-uop*]:
assumes *1*: *mwb-lens v*
shows $(v :== e1 ;; v :== (uop\ F\ (\&v))) = (v :== (uop\ F\ e1))$
using *1*
by *rel-auto*

lemma *assign-bop1*[*symbolic-exec-assign-bop*]:
assumes *1*: *mwb-lens v* **and** *2*:$v \sharp e2$
shows $(v :== e1 ;; v :== (bop\ bp\ (\&v)\ e2)) = (v :== (bop\ bp\ e1\ e2))$

```
  using 1 2
  by rel-auto

lemma assign-bop2[symbolic-exec-assign-bop]:
  assumes 1: mwb-lens v and 2:v ♯ e2
  shows (v :== e1 ;; v :== (bop bp e2 (&v))) = (v :== (bop bp e2 e1))
  using 1 2
  by rel-auto

lemma assign-trop1[symbolic-exec-assign-trop]:
  assumes 1: mwb-lens v and 2:v ♯ e2 and 3:v ♯ e3
  shows (v :== e1 ;; v :== (trop tp (&v) e2 e3)) =
        (v :== (trop tp e1 e2 e3))
  using 1 2 3
  by rel-auto

lemma assign-trop2[symbolic-exec-assign-trop]:
  assumes 1: mwb-lens v and 2:v ♯ e2 and 3:v ♯ e3
  shows (v :== e1 ;; v :== (trop tp e2 (&v) e3)) =
        (v :== (trop tp e2 e1 e3))
  using 1 2 3
  by rel-auto

lemma assign-trop3[symbolic-exec-assign-trop]:
  assumes 1: mwb-lens v and 2:v ♯ e2 and 3:v ♯ e3
  shows (v :== e1 ;; v :== (trop tp e2 e3 (&v))) =
        (v :== (trop tp e2 e3 e1))
  using 1 2 3
  by rel-auto

lemma assign-qtop1[symbolic-exec-assign-qtop]:
  assumes 1: mwb-lens v and 2:v ♯ e2 and 3:v ♯ e3 and 4:v ♯ e4
  shows (v :== e1 ;; v :== (qtop tp (&v) e2 e3 e4)) =
        (v :== (qtop tp e1 e2 e3 e4))
  using 1 2 3 4
  by rel-auto

lemma assign-qtop2[symbolic-exec-assign-qtop]:
  assumes 1: mwb-lens v and 2:v ♯ e2 and 3:v ♯ e3 and 4:v ♯ e4
  shows (v :== e1 ;; v :== (qtop tp e2 (&v) e3 e4)) =
        (v :== (qtop tp e2 e1 e3 e4))
  using 1 2 3 4
  by rel-auto

lemma assign-qtop3[symbolic-exec-assign-qtop]:
```

```
  assumes 1: mwb-lens v and 2:v ♯ e2 and 3:v ♯ e3 and 4:v ♯ e4
  shows (v :== e1 ;; v :== (qtop tp e2 e3 (&v) e4)) =
        (v :== (qtop tp e2 e3 e1 e4))
  using 1 2 3 4
  by rel-auto
```

```
lemma assign-qtop4[symbolic-exec-assign-qtop]:
  assumes 1: mwb-lens v and 2:v ♯ e2 and 3:v ♯ e3 and 4:v ♯ e4
  shows (v :== e1 ;; v :== (qtop tp e2 e3 e4 (&v))) =
        (v :== (qtop tp e2 e3 e4 e1))
  using 1 2 3 4
  by rel-auto
```

```
lemma assign-cond-seqr-dist:
  (v :== e ;; (P ◁ b ▷ Q)) = ((v :== e ;; P) ◁ b⟦⌈e⌉_</$v⟧ ▷ (v :== e ;; Q))
  by rel-auto
```

In the sequel we find assignment laws proposed by Hoare

```
lemma assign-vwb-skip:
  assumes 1: vwb-lens v
  shows (v :== &v) = II
  by (simp add: assms skip-r-def usubst-upd-var-id)
```

```
lemma assign-simultaneous:
  assumes 1: vwb-lens v2
  and     2: v1 ⋈ v2
  shows (v1,v2 :== e, (&v2)) = (v1 :== e)
  by (simp add: 1 2 usubst-upd-comm usubst-upd-var-id)
```

```
lemma assign-seq:
  assumes 1: vwb-lens var2
  shows(var1:== expr);; (var2 :== &var2) = (var1:== expr)
  using 1 by rel-auto
```

```
lemma assign-cond-uop[symbolic-exec-assign-uop]:
  assumes 1: weak-lens v
  shows v :== expr ;; (C1 ◁uop F (&v)▷_r C2) =
        (v :== expr ;; C1) ◁uop F expr▷_r (v :== expr ;; C2)
  using 1
  by rel-auto
```

```
lemma assign-cond-bop1[symbolic-exec-assign-bop]:
  assumes 1: weak-lens v and 2: v ♯ exp2
  shows (v :== expr ;; (C1 ◁(bop bp (&v) exp2)▷_r C2)) =
        ((v :== expr ;; C1) ◁(bop bp expr exp2)▷_r (v :== expr ;; C2))
```

```
  using 1 2
  by rel-auto

lemma assign-cond-bop2[symbolic-exec-assign-bop]:
  assumes 1: weak-lens v and 2: v ♯ exp2
  shows (v :== exp1 ;; (C1 ◁(bop bp exp2 (&v))▷ᵣ C2)) =
        ((v :== exp1 ;; C1) ◁(bop bp exp2 exp1)▷ᵣ (v :== exp1 ;; C2))
  using 1 2
  by rel-auto

lemma assign-cond-trop1[symbolic-exec-assign-trop]:
  assumes 1: weak-lens v and 2: v ♯ exp2 and 3: v ♯ exp3
  shows (v :== expr ;; (C1 ◁(trop tp (&v) exp2 exp3)▷ᵣ C2)) =
        ((v :== expr ;; C1) ◁(trop tp expr exp2 exp3)▷ᵣ (v :== expr ;; C2))
  using 1 2 3
  by rel-auto

lemma assign-cond-trop2[symbolic-exec-assign-trop]:
  assumes 1: weak-lens v and 2: v ♯ exp2 and 3: v ♯ exp3
  shows (v :== exp1 ;; (C1 ◁(trop tp exp2 (&v) exp3)▷ᵣ C2)) =
        ((v :== exp1 ;; C1) ◁(trop tp exp2 exp1 exp3)▷ᵣ (v :== exp1 ;; C2))
  using 1 2 3
  by rel-auto

lemma assign-cond-trop3[symbolic-exec-assign-trop]:
  assumes 1: weak-lens v and 2: v ♯ exp2 and 3: v ♯ exp3
  shows (v :== exp1 ;; (C1 ◁(trop bp exp2 exp3 (&v))▷ᵣ C2)) =
        ((v :== exp1 ;; C1) ◁(trop bp exp2 exp3 exp1)▷ᵣ (v :== exp1 ;; C2))
  using 1 2 3
  by rel-auto

lemma assign-cond-qtop1[symbolic-exec-assign-qtop]:
  assumes 1: weak-lens v and 2: v ♯ exp2 and 3: v ♯ exp3 and 4: v ♯ exp4
  shows (v :== exp1 ;; (C1 ◁(qtop tp (&v) exp2 exp3 exp4)▷ᵣ C2)) =
        ((v :== exp1 ;; C1) ◁(qtop tp exp1 exp2 exp3  exp4)▷ᵣ (v :== exp1 ;; C2))
  using 1 2 3 4
  by rel-auto

lemma assign-cond-qtop2[symbolic-exec-assign-qtop]:
  assumes 1: weak-lens v and 2: v ♯ exp2 and 3: v ♯ exp3 and 4: v ♯ exp4
  shows (v :== exp1 ;; (C1 ◁(qtop tp exp2 (&v) exp3 exp4)▷ᵣ C2)) =
        ((v :== exp1 ;; C1) ◁(qtop tp exp2 exp1 exp3 exp4)▷ᵣ (v :== exp1 ;; C2))
  using 1 2 3 4
  by rel-auto
```

lemma *assign-cond-qtop3*[*symbolic-exec-assign-qtop*]:
assumes *1*: *weak-lens* v **and** *2*: $v \sharp exp2$ **and** *3*: $v \sharp exp3$ **and** *4*: $v \sharp exp4$
shows $(v :== exp1\ ;;\ (C1 \triangleleft(qtop\ bp\ exp2\ exp3\ (\&v)\ exp4)\triangleright_r C2)) =$
$((v :== exp1\ ;;\ C1) \triangleleft(qtop\ bp\ exp2\ exp3\ exp1\ \ exp4)\triangleright_r (v :== exp1\ ;;\ C2))$
using *1 2 3 4*
by *rel-auto*

lemma *assign-cond-qtop4*[*symbolic-exec-assign-qtop*]:
assumes *1*: *weak-lens* v **and** *2*: $v \sharp exp2$ **and** *3*: $v \sharp exp3$ **and** *4*: $v \sharp exp4$
shows $(v :== exp1\ ;;\ (C1 \triangleleft(qtop\ bp\ exp2\ exp3\ exp4\ (\&v))\triangleright_r C2)) =$
$((v :== exp1\ ;;\ C1) \triangleleft(qtop\ bp\ exp2\ exp3\ \ exp4\ exp1)\triangleright_r (v :== exp1\ ;;\ C2))$
using *1 2 3 4*
by *rel-auto*

lemma *assign-cond-If* [*symbolic-exec*]:
$((v :== exp1) \triangleleft bexp \triangleright_r (v :== exp2)) =$
$(v :== (trop\ If\ bexp\ exp1\ exp2))$
by *rel-auto*

lemma *assign-cond-If-uop*[*symbolic-exec-assign-uop*]:
assumes *1*: *mwb-lens* v
shows $(v :== E;;$
$((v :== uop\ F\ (\&v)) \triangleleft uop\ F\ (\&v)\triangleright_r (v :== uop\ G\ (\&v)))) =$
$(v :== trop\ If\ (uop\ F\ E)\ (uop\ F\ E)\ (uop\ G\ E))$
using *1*
proof (*rel-simp*, *transfer*)
fix $a :: 'a$ **and** $b :: 'a$ **and** $va :: bool \Longrightarrow 'a$ **and** $Fa :: bool \Rightarrow bool$ **and** $Ea :: 'a \Rightarrow bool$ **and** $Ga :: bool \Rightarrow bool$
have $Fa\ (Ea\ a) \longrightarrow (Fa\ (Ea\ a) \wedge b = put_{va}\ a\ (Fa\ (Ea\ a)) \vee \neg\ Fa\ (Ea\ a) \wedge b = put_{va}\ a\ (Ga\ (Ea\ a))) \wedge b = put_{va}\ a\ (Fa\ (Ea\ a) \vee \neg\ Fa\ (Ea\ a) \wedge Ga\ (Ea\ a)) \vee \neg\ b = put_{va}\ a\ (Fa\ (Ea\ a) \vee \neg\ Fa\ (Ea\ a) \wedge Ga\ (Ea\ a)) \wedge (\neg\ Fa\ (Ea\ a) \vee \neg\ b = put_{va}\ a\ (Fa\ (Ea\ a)))$
by *presburger*
then have $\neg\ ((\neg\ Fa\ (Ea\ a) \vee \neg\ b = put_{va}\ a\ (Fa\ (Ea\ a))) \wedge (Fa\ (Ea\ a) \vee \neg\ b = put_{va}\ a\ (Ga\ (Ea\ a)))) = (b = put_{va}\ a\ (Fa\ (Ea\ a) \vee \neg\ Fa\ (Ea\ a) \wedge Ga\ (Ea\ a)))$
by *fastforce*
then show $(Fa\ (Ea\ a) \wedge b = put_{va}\ a\ (Fa\ (Ea\ a)) \vee \neg\ Fa\ (Ea\ a) \wedge b = put_{va}\ a\ (Ga\ (Ea\ a))) = (b = put_{va}\ a\ (Fa\ (Ea\ a) \vee \neg\ Fa\ (Ea\ a) \wedge Ga\ (Ea\ a)))$
by *meson*
qed

lemma *assign-cond-If-bop*[*symbolic-exec-assign-bop*]:
assumes *1*: *mwb-lens* v **and** *2*: $v \sharp expr$
shows $((v :== E);;$
$((v :== (bop\ F\ expr\ (\&v))) \triangleleft bop\ F\ expr\ (\&v)\triangleright_r (v :== (bop\ G\ expr\ (\&v))))) =$
$(v :== (trop\ If\ (bop\ F\ expr\ E)\ (bop\ F\ expr\ E)\ (bop\ G\ expr\ E)))$

using *1 2*

proof (*rel-simp*, *transfer*)

fix $a :: 'a$ **and** $b :: 'a$ **and** $va :: bool \Longrightarrow 'a$ **and** $Fa :: bool \Rightarrow bool$ **and** $Ea :: 'a \Rightarrow bool$ **and** $Ga :: bool \Rightarrow bool$

have $Fa\ (Ea\ a) \longrightarrow (Fa\ (Ea\ a) \wedge b = put_{va}\ a\ (Fa\ (Ea\ a)) \vee \neg\ Fa\ (Ea\ a) \wedge b = put_{va}\ a\ (Ga\ (Ea\ a))) \wedge b = put_{va}\ a\ (Fa\ (Ea\ a) \vee \neg\ Fa\ (Ea\ a) \wedge Ga\ (Ea\ a)) \vee \neg\ b = put_{va}\ a\ (Fa\ (Ea\ a) \vee \neg\ Fa\ (Ea\ a) \wedge Ga\ (Ea\ a)) \wedge (\neg\ Fa\ (Ea\ a) \vee \neg\ b = put_{va}\ a\ (Fa\ (Ea\ a)))$

by *presburger*

then have $\neg\ ((\neg\ Fa\ (Ea\ a) \vee \neg\ b = put_{va}\ a\ (Fa\ (Ea\ a))) \wedge (Fa\ (Ea\ a) \vee \neg\ b = put_{va}\ a\ (Ga\ (Ea\ a)))) = (b = put_{va}\ a\ (Fa\ (Ea\ a) \vee \neg\ Fa\ (Ea\ a) \wedge Ga\ (Ea\ a)))$

by *fastforce*

then show $(Fa\ (Ea\ a) \wedge b = put_{va}\ a\ (Fa\ (Ea\ a)) \vee \neg\ Fa\ (Ea\ a) \wedge b = put_{va}\ a\ (Ga\ (Ea\ a))) = (b = put_{va}\ a\ (Fa\ (Ea\ a) \vee \neg\ Fa\ (Ea\ a) \wedge Ga\ (Ea\ a)))$

by *meson*

qed

lemma *assign-cond-If-bop1* [*symbolic-exec-assign-bop*]:

assumes *1*: *mwb-lens v* **and** *2*: $v \sharp expr$

shows $((v :== E);;$

$((v :== (bop\ F\ (\&v)\ expr)) \triangleleft bop\ F\ (\&v)\ expr \triangleright_r (v :== (bop\ G\ (\&v)\ expr)))) =$

$(v :== (trop\ If\ (bop\ F\ E\ expr)\ (bop\ F\ E\ expr)\ (bop\ G\ E\ expr)))$

using *1 2*

proof (*rel-simp*, *transfer*)

fix $a :: 'a$ **and** $b :: 'a$ **and** $va :: bool \Longrightarrow 'a$ **and** $Fa :: bool \Rightarrow bool$ **and** $Ea :: 'a \Rightarrow bool$ **and** $Ga :: bool \Rightarrow bool$

have $Fa\ (Ea\ a) \longrightarrow (Fa\ (Ea\ a) \wedge b = put_{va}\ a\ (Fa\ (Ea\ a)) \vee \neg\ Fa\ (Ea\ a) \wedge b = put_{va}\ a\ (Ga\ (Ea\ a))) \wedge b = put_{va}\ a\ (Fa\ (Ea\ a) \vee \neg\ Fa\ (Ea\ a) \wedge Ga\ (Ea\ a)) \vee \neg\ b = put_{va}\ a\ (Fa\ (Ea\ a) \vee \neg\ Fa\ (Ea\ a) \wedge Ga\ (Ea\ a)) \wedge (\neg\ Fa\ (Ea\ a) \vee \neg\ b = put_{va}\ a\ (Fa\ (Ea\ a)))$

by *presburger*

then have $\neg\ ((\neg\ Fa\ (Ea\ a) \vee \neg\ b = put_{va}\ a\ (Fa\ (Ea\ a))) \wedge (Fa\ (Ea\ a) \vee \neg\ b = put_{va}\ a\ (Ga\ (Ea\ a)))) = (b = put_{va}\ a\ (Fa\ (Ea\ a) \vee \neg\ Fa\ (Ea\ a) \wedge Ga\ (Ea\ a)))$

by *fastforce*

then show $(Fa\ (Ea\ a) \wedge b = put_{va}\ a\ (Fa\ (Ea\ a)) \vee \neg\ Fa\ (Ea\ a) \wedge b = put_{va}\ a\ (Ga\ (Ea\ a))) = (b = put_{va}\ a\ (Fa\ (Ea\ a) \vee \neg\ Fa\ (Ea\ a) \wedge Ga\ (Ea\ a)))$

by *meson*

qed

lemma *assign-cond-If-bop2* [*symbolic-exec-assign-bop*]:

assumes *1*: *mwb-lens v* **and** *2*: $v \sharp exp1$ **and** *3*: $v \sharp exp2$

shows $((v :== E);;$

$((v :== (bop\ F\ (\&v)\ exp1)) \triangleleft bop\ F\ (\&v)\ exp1 \triangleright_r (v :== (bop\ G\ (\&v)\ exp2)))) =$

$(v :== (trop\ If\ (bop\ F\ E\ exp1)\ (bop\ F\ E\ exp1)\ (bop\ G\ E\ exp2)))$

using *1 2 3*

proof (*rel-simp*, *transfer*)

fix $a :: 'a$ **and** $b :: 'a$ **and** $va :: bool \Longrightarrow 'a$ **and** $Fa :: bool \Rightarrow bool$ **and** $Ea :: 'a \Rightarrow bool$ **and** Ga

:: *bool* ⇒ *bool*
have *Fa* (*Ea a*) ⟶ (*Fa* (*Ea a*) ∧ *b* = *put*$_{va}$ *a* (*Fa* (*Ea a*)) ∨ ¬ *Fa* (*Ea a*) ∧ *b* = *put*$_{va}$ *a* (*Ga* (*Ea a*))) ∧ *b* = *put*$_{va}$ *a* (*Fa* (*Ea a*) ∨ ¬ *Fa* (*Ea a*) ∧ *Ga* (*Ea a*)) ∨ ¬ *b* = *put*$_{va}$ *a* (*Fa* (*Ea a*) ∨ ¬ *Fa* (*Ea a*) ∧ *Ga* (*Ea a*)) ∧ (¬ *Fa* (*Ea a*) ∨ ¬ *b* = *put*$_{va}$ *a* (*Fa* (*Ea a*)))
by *presburger*
then have ¬ ((¬ *Fa* (*Ea a*) ∨ ¬ *b* = *put*$_{va}$ *a* (*Fa* (*Ea a*))) ∧ (*Fa* (*Ea a*) ∨ ¬ *b* = *put*$_{va}$ *a* (*Ga* (*Ea a*)))) = (*b* = *put*$_{va}$ *a* (*Fa* (*Ea a*) ∨ ¬ *Fa* (*Ea a*) ∧ *Ga* (*Ea a*)))
by *fastforce*
then show (*Fa* (*Ea a*) ∧ *b* = *put*$_{va}$ *a* (*Fa* (*Ea a*)) ∨ ¬ *Fa* (*Ea a*) ∧ *b* = *put*$_{va}$ *a* (*Ga* (*Ea a*))) = (*b* = *put*$_{va}$ *a* (*Fa* (*Ea a*) ∨ ¬ *Fa* (*Ea a*) ∧ *Ga* (*Ea a*)))
by *meson*
qed

lemma *assign-cond-If-bop4* [*symbolic-exec-assign-bop*]:
assumes *1*: *mwb-lens v* **and** *2*: *v* ♯ *exp1* **and** *3*: *v* ♯ *exp2*
shows ((*v* :== *E*);;
((*v* :== (*bop F* (&*v*) *exp1*)) ◁*bop F* (&*v*) *exp1*▷$_r$ (*v* :== (*bop G exp2* (&*v*))))) =
(*v* :== (*trop If* (*bop F E exp1*) (*bop F E exp1*) (*bop G exp2 E*)))
using *1 2 3*
proof (*rel-simp*, *transfer*)
fix *a* :: ′*a* **and** *b* :: ′*a* **and** *va* :: *bool* ⟹ ′*a* **and** *Fa* :: *bool* ⇒ *bool* **and** *Ea* :: ′*a* ⇒ *bool* **and** *Ga* :: *bool* ⇒ *bool*
have *Fa* (*Ea a*) ⟶ (*Fa* (*Ea a*) ∧ *b* = *put*$_{va}$ *a* (*Fa* (*Ea a*)) ∨ ¬ *Fa* (*Ea a*) ∧ *b* = *put*$_{va}$ *a* (*Ga* (*Ea a*))) ∧ *b* = *put*$_{va}$ *a* (*Fa* (*Ea a*) ∨ ¬ *Fa* (*Ea a*) ∧ *Ga* (*Ea a*)) ∨ ¬ *b* = *put*$_{va}$ *a* (*Fa* (*Ea a*) ∨ ¬ *Fa* (*Ea a*) ∧ *Ga* (*Ea a*)) ∧ (¬ *Fa* (*Ea a*) ∨ ¬ *b* = *put*$_{va}$ *a* (*Fa* (*Ea a*)))
by *presburger*
then have ¬ ((¬ *Fa* (*Ea a*) ∨ ¬ *b* = *put*$_{va}$ *a* (*Fa* (*Ea a*))) ∧ (*Fa* (*Ea a*) ∨ ¬ *b* = *put*$_{va}$ *a* (*Ga* (*Ea a*)))) = (*b* = *put*$_{va}$ *a* (*Fa* (*Ea a*) ∨ ¬ *Fa* (*Ea a*) ∧ *Ga* (*Ea a*)))
by *fastforce*
then show (*Fa* (*Ea a*) ∧ *b* = *put*$_{va}$ *a* (*Fa* (*Ea a*)) ∨ ¬ *Fa* (*Ea a*) ∧ *b* = *put*$_{va}$ *a* (*Ga* (*Ea a*))) = (*b* = *put*$_{va}$ *a* (*Fa* (*Ea a*) ∨ ¬ *Fa* (*Ea a*) ∧ *Ga* (*Ea a*)))
by *meson*
qed

lemma *assign-cond-If-bop5* [*symbolic-exec-assign-bop*]:
assumes *1*: *mwb-lens v* **and** *2*: *v* ♯ *exp1* **and** *3*: *v* ♯ *exp2*
shows ((*v* :== *E*);;
((*v* :== (*bop F exp1* (&*v*))) ◁*bop F exp1* (&*v*)▷$_r$ (*v* :== (*bop G* (&*v*) *exp2*)))) =
(*v* :== (*trop If* (*bop F exp1 E*) (*bop F exp1 E*) (*bop G E exp2*)))
using *1 2 3*
proof (*rel-simp*, *transfer*)
fix *a* :: ′*a* **and** *b* :: ′*a* **and** *va* :: *bool* ⟹ ′*a* **and** *Fa* :: *bool* ⇒ *bool* **and** *Ea* :: ′*a* ⇒ *bool* **and** *Ga* :: *bool* ⇒ *bool*
have *Fa* (*Ea a*) ⟶ (*Fa* (*Ea a*) ∧ *b* = *put*$_{va}$ *a* (*Fa* (*Ea a*)) ∨ ¬ *Fa* (*Ea a*) ∧ *b* = *put*$_{va}$ *a* (*Ga* (*Ea a*))) ∧ *b* = *put*$_{va}$ *a* (*Fa* (*Ea a*) ∨ ¬ *Fa* (*Ea a*) ∧ *Ga* (*Ea a*)) ∨ ¬ *b* = *put*$_{va}$ *a* (*Fa* (*Ea a*) ∨ ¬

$Fa\ (Ea\ a) \wedge Ga\ (Ea\ a)) \wedge (\neg\ Fa\ (Ea\ a) \vee \neg\ b = put_{va}\ a\ (Fa\ (Ea\ a)))$

by *presburger*

then have $\neg\ ((\neg\ Fa\ (Ea\ a) \vee \neg\ b = put_{va}\ a\ (Fa\ (Ea\ a))) \wedge (Fa\ (Ea\ a) \vee \neg\ b = put_{va}\ a\ (Ga\ (Ea\ a)))) = (b = put_{va}\ a\ (Fa\ (Ea\ a) \vee \neg\ Fa\ (Ea\ a) \wedge Ga\ (Ea\ a)))$

by *fastforce*

then show $(Fa\ (Ea\ a) \wedge b = put_{va}\ a\ (Fa\ (Ea\ a)) \vee \neg\ Fa\ (Ea\ a) \wedge b = put_{va}\ a\ (Ga\ (Ea\ a))) = (b = put_{va}\ a\ (Fa\ (Ea\ a) \vee \neg\ Fa\ (Ea\ a) \wedge Ga\ (Ea\ a)))$

by *meson*

qed

lemma *assign-cond-If-bop6*[*symbolic-exec-assign-bop*]:

assumes *1*: *mwb-lens* v **and** *2*: $v \sharp exp1$ **and** *3*: $v \sharp exp2$

shows $((v :== E);;$

$((v :== (bop\ F\ exp1\ (\&v))) \triangleleft bop\ F\ exp1\ (\&v) \triangleright_r (v :== (bop\ G\ exp2\ (\&v))))) =$

$(v :== (trop\ If\ (bop\ F\ exp1\ E)\ (bop\ F\ exp1\ E)\ (bop\ G\ exp2\ E)))$

using *1 2 3*

proof (*rel-simp*, *transfer*)

fix $a :: 'a$ **and** $b :: 'a$ **and** $va :: bool \Longrightarrow 'a$ **and** $Fa :: bool \Rightarrow bool$ **and** $Ea :: 'a \Rightarrow bool$ **and** $Ga :: bool \Rightarrow bool$

have $Fa\ (Ea\ a) \longrightarrow (Fa\ (Ea\ a) \wedge b = put_{va}\ a\ (Fa\ (Ea\ a)) \vee \neg\ Fa\ (Ea\ a) \wedge b = put_{va}\ a\ (Ga\ (Ea\ a))) \wedge b = put_{va}\ a\ (Fa\ (Ea\ a) \vee \neg\ Fa\ (Ea\ a) \wedge Ga\ (Ea\ a)) \vee \neg\ b = put_{va}\ a\ (Fa\ (Ea\ a) \vee \neg\ Fa\ (Ea\ a) \wedge Ga\ (Ea\ a)) \wedge (\neg\ Fa\ (Ea\ a) \vee \neg\ b = put_{va}\ a\ (Fa\ (Ea\ a)))$

by *presburger*

then have $\neg\ ((\neg\ Fa\ (Ea\ a) \vee \neg\ b = put_{va}\ a\ (Fa\ (Ea\ a))) \wedge (Fa\ (Ea\ a) \vee \neg\ b = put_{va}\ a\ (Ga\ (Ea\ a)))) = (b = put_{va}\ a\ (Fa\ (Ea\ a) \vee \neg\ Fa\ (Ea\ a) \wedge Ga\ (Ea\ a)))$

by *fastforce*

then show $(Fa\ (Ea\ a) \wedge b = put_{va}\ a\ (Fa\ (Ea\ a)) \vee \neg\ Fa\ (Ea\ a) \wedge b = put_{va}\ a\ (Ga\ (Ea\ a))) = (b = put_{va}\ a\ (Fa\ (Ea\ a) \vee \neg\ Fa\ (Ea\ a) \wedge Ga\ (Ea\ a)))$

by *meson*

qed

lemma *assign-cond-If-trop*[*symbolic-exec-assign-trop*]:

assumes *1*: *mwb-lens* v **and** *2*: $v \sharp exp1$ **and** *3*: $v \sharp exp2$

shows $((v :== E);;$

$((v :== (trop\ F\ exp1\ exp2\ (\&v))) \triangleleft trop\ F\ exp1\ exp2\ (\&v) \triangleright_r (v :== (trop\ G\ exp1\ exp2\ (\&v))))) =$

$(v :== (trop\ If\ (trop\ F\ exp1\ exp2\ E)\ (trop\ F\ exp1\ exp2\ E)\ (trop\ G\ exp1\ exp2\ E)))$

using *1 2 3*

proof (*rel-simp*, *transfer*)

fix $a :: 'a$ **and** $b :: 'a$ **and** $va :: bool \Longrightarrow 'a$ **and** $Fa :: bool \Rightarrow bool$ **and** $Ea :: 'a \Rightarrow bool$ **and** $Ga :: bool \Rightarrow bool$

have $Fa\ (Ea\ a) \longrightarrow (Fa\ (Ea\ a) \wedge b = put_{va}\ a\ (Fa\ (Ea\ a)) \vee \neg\ Fa\ (Ea\ a) \wedge b = put_{va}\ a\ (Ga\ (Ea\ a))) \wedge b = put_{va}\ a\ (Fa\ (Ea\ a) \vee \neg\ Fa\ (Ea\ a) \wedge Ga\ (Ea\ a)) \vee \neg\ b = put_{va}\ a\ (Fa\ (Ea\ a) \vee \neg\ Fa\ (Ea\ a) \wedge Ga\ (Ea\ a)) \wedge (\neg\ Fa\ (Ea\ a) \vee \neg\ b = put_{va}\ a\ (Fa\ (Ea\ a)))$

by *presburger*

then have $\neg ((\neg Fa\ (Ea\ a) \vee \neg b = put_{va}\ a\ (Fa\ (Ea\ a))) \wedge (Fa\ (Ea\ a) \vee \neg b = put_{va}\ a\ (Ga\ (Ea\ a)))) = (b = put_{va}\ a\ (Fa\ (Ea\ a) \vee \neg Fa\ (Ea\ a) \wedge Ga\ (Ea\ a)))$
  by *fastforce*
then show $(Fa\ (Ea\ a) \wedge b = put_{va}\ a\ (Fa\ (Ea\ a)) \vee \neg Fa\ (Ea\ a) \wedge b = put_{va}\ a\ (Ga\ (Ea\ a))) = (b = put_{va}\ a\ (Fa\ (Ea\ a) \vee \neg Fa\ (Ea\ a) \wedge Ga\ (Ea\ a)))$
  by *meson*
qed

lemma *assign-cond-If-trop1* [*symbolic-exec-assign-trop*]:
assumes *1*: *mwb-lens* v **and** *2*: $v \sharp exp1$ **and** *3*: $v \sharp exp2$
shows $((v :== E);;$
$((v :== (trop\ F\ exp1\ (\&v)\ exp2)) \triangleleft trop\ F\ exp1\ (\&v)\ exp2 \triangleright_r (v :== (trop\ G\ exp1\ (\&v)\ exp2)))) =$
$(v :== (trop\ If\ (trop\ F\ exp1\ E\ exp2)\ (trop\ F\ exp1\ E\ exp2)\ (trop\ G\ exp1\ E\ exp2)))$
using *1 2 3*
proof (*rel-simp*, *transfer*)
fix $a :: 'a$ **and** $b :: 'a$ **and** $va :: bool \Longrightarrow 'a$ **and** $Fa :: bool \Rightarrow bool$ **and** $Ea :: 'a \Rightarrow bool$ **and** $Ga :: bool \Rightarrow bool$
have $Fa\ (Ea\ a) \longrightarrow (Fa\ (Ea\ a) \wedge b = put_{va}\ a\ (Fa\ (Ea\ a)) \vee \neg Fa\ (Ea\ a) \wedge b = put_{va}\ a\ (Ga\ (Ea\ a))) \wedge b = put_{va}\ a\ (Fa\ (Ea\ a) \vee \neg Fa\ (Ea\ a) \wedge Ga\ (Ea\ a)) \vee \neg b = put_{va}\ a\ (Fa\ (Ea\ a) \vee \neg Fa\ (Ea\ a) \wedge Ga\ (Ea\ a)) \wedge (\neg Fa\ (Ea\ a) \vee \neg b = put_{va}\ a\ (Fa\ (Ea\ a)))$
  by *presburger*
then have $\neg ((\neg Fa\ (Ea\ a) \vee \neg b = put_{va}\ a\ (Fa\ (Ea\ a))) \wedge (Fa\ (Ea\ a) \vee \neg b = put_{va}\ a\ (Ga\ (Ea\ a)))) = (b = put_{va}\ a\ (Fa\ (Ea\ a) \vee \neg Fa\ (Ea\ a) \wedge Ga\ (Ea\ a)))$
  by *fastforce*
then show $(Fa\ (Ea\ a) \wedge b = put_{va}\ a\ (Fa\ (Ea\ a)) \vee \neg Fa\ (Ea\ a) \wedge b = put_{va}\ a\ (Ga\ (Ea\ a))) = (b = put_{va}\ a\ (Fa\ (Ea\ a) \vee \neg Fa\ (Ea\ a) \wedge Ga\ (Ea\ a)))$
  by *meson*
qed

lemma *assign-cond-If-trop2* [*symbolic-exec-assign-trop*]:
assumes *1*: *mwb-lens* v **and** *2*: $v \sharp exp1$ **and** *3*: $v \sharp exp2$
shows $((v :== E);;$
$((v :== (trop\ F\ (\&v)\ exp1\ exp2)) \triangleleft trop\ F\ (\&v)\ exp1\ exp2 \triangleright_r (v :== (trop\ G\ (\&v)\ exp1\ exp2)))) =$
$(v :== (trop\ If\ (trop\ F\ E\ exp1\ exp2)\ (trop\ F\ E\ exp1\ exp2)\ (trop\ G\ E\ exp1\ exp2)))$
using *1 2 3*
proof (*rel-simp*, *transfer*)
fix $a :: 'a$ **and** $b :: 'a$ **and** $va :: bool \Longrightarrow 'a$ **and** $Fa :: bool \Rightarrow bool$ **and** $Ea :: 'a \Rightarrow bool$ **and** $Ga :: bool \Rightarrow bool$
have $Fa\ (Ea\ a) \longrightarrow (Fa\ (Ea\ a) \wedge b = put_{va}\ a\ (Fa\ (Ea\ a)) \vee \neg Fa\ (Ea\ a) \wedge b = put_{va}\ a\ (Ga\ (Ea\ a))) \wedge b = put_{va}\ a\ (Fa\ (Ea\ a) \vee \neg Fa\ (Ea\ a) \wedge Ga\ (Ea\ a)) \vee \neg b = put_{va}\ a\ (Fa\ (Ea\ a) \vee \neg Fa\ (Ea\ a) \wedge Ga\ (Ea\ a)) \wedge (\neg Fa\ (Ea\ a) \vee \neg b = put_{va}\ a\ (Fa\ (Ea\ a)))$
  by *presburger*
then have $\neg ((\neg Fa\ (Ea\ a) \vee \neg b = put_{va}\ a\ (Fa\ (Ea\ a))) \wedge (Fa\ (Ea\ a) \vee \neg b = put_{va}\ a\ (Ga$

(Ea a)))) = (b = put_{va} a (Fa (Ea a) ∨ ¬ Fa (Ea a) ∧ Ga (Ea a)))
  by *fastforce*
 then show (Fa (Ea a) ∧ b = put_{va} a (Fa (Ea a)) ∨ ¬ Fa (Ea a) ∧ b = put_{va} a (Ga (Ea a))) = (b = put_{va} a (Fa (Ea a) ∨ ¬ Fa (Ea a) ∧ Ga (Ea a)))
  by *meson*
qed

lemma *assign-cond-If-trop3* [*symbolic-exec-assign-trop*]:
 assumes *1*: *mwb-lens v* **and** *2*: v ♯ *exp1* **and** *3*: v ♯ *exp2* **and** *4*: v ♯ *exp3* **and** *5*: v ♯ *exp4*
 shows ((v :== E);;
   ((v :== (trop F exp1 exp2 (&v))) ◁trop F exp1 exp2 (&v)$\triangleright_r$ (v :== (trop G exp3 exp4 (&v))))) =
  (v :== (trop If (trop F exp1 exp2 E) (trop F exp1 exp2 E) (trop G exp3 exp4 E)))
 using *1 2 3 4 5*
proof (*rel-simp*, *transfer*)
 fix a :: 'a **and** b :: 'a **and** va :: bool ⟹ 'a **and** Fa :: bool ⇒ bool **and** Ea :: 'a ⇒ bool **and** Ga :: bool ⇒ bool
 have Fa (Ea a) ⟶ (Fa (Ea a) ∧ b = put_{va} a (Fa (Ea a)) ∨ ¬ Fa (Ea a) ∧ b = put_{va} a (Ga (Ea a))) ∧ b = put_{va} a (Fa (Ea a) ∨ ¬ Fa (Ea a) ∧ Ga (Ea a)) ∨ ¬ b = put_{va} a (Fa (Ea a) ∨ ¬ Fa (Ea a) ∧ Ga (Ea a)) ∧ (¬ Fa (Ea a) ∨ ¬ b = put_{va} a (Fa (Ea a)))
  by *presburger*
 then have ¬ ((¬ Fa (Ea a) ∨ ¬ b = put_{va} a (Fa (Ea a))) ∧ (Fa (Ea a) ∨ ¬ b = put_{va} a (Ga (Ea a)))) = (b = put_{va} a (Fa (Ea a) ∨ ¬ Fa (Ea a) ∧ Ga (Ea a)))
  by *fastforce*
 then show (Fa (Ea a) ∧ b = put_{va} a (Fa (Ea a)) ∨ ¬ Fa (Ea a) ∧ b = put_{va} a (Ga (Ea a))) = (b = put_{va} a (Fa (Ea a) ∨ ¬ Fa (Ea a) ∧ Ga (Ea a)))
  by *meson*
qed

lemma *assign-cond-If-trop4* [*symbolic-exec-assign-trop*]:
 assumes *1*: *mwb-lens v* **and** *2*: v ♯ *exp1* **and** *3*: v ♯ *exp2* **and** *4*: v ♯ *exp3* **and** *5*: v ♯ *exp4*
 shows ((v :== E);;
  ((v :== (trop F exp1 (&v) exp2)) ◁trop F exp1 (&v) exp2$\triangleright_r$ (v :== (trop G exp3 (&v) exp4)))) =
  (v :== (trop If (trop F exp1 E exp2) (trop F exp1 E exp2) (trop G exp3 E exp4)))
 using *1 2 3 4 5*
proof (*rel-simp*, *transfer*)
 fix a :: 'a **and** b :: 'a **and** va :: bool ⟹ 'a **and** Fa :: bool ⇒ bool **and** Ea :: 'a ⇒ bool **and** Ga :: bool ⇒ bool
 have Fa (Ea a) ⟶ (Fa (Ea a) ∧ b = put_{va} a (Fa (Ea a)) ∨ ¬ Fa (Ea a) ∧ b = put_{va} a (Ga (Ea a))) ∧ b = put_{va} a (Fa (Ea a) ∨ ¬ Fa (Ea a) ∧ Ga (Ea a)) ∨ ¬ b = put_{va} a (Fa (Ea a) ∨ ¬ Fa (Ea a) ∧ Ga (Ea a)) ∧ (¬ Fa (Ea a) ∨ ¬ b = put_{va} a (Fa (Ea a)))
  by *presburger*
 then have ¬ ((¬ Fa (Ea a) ∨ ¬ b = put_{va} a (Fa (Ea a))) ∧ (Fa (Ea a) ∨ ¬ b = put_{va} a (Ga (Ea a)))) = (b = put_{va} a (Fa (Ea a) ∨ ¬ Fa (Ea a) ∧ Ga (Ea a)))

by *fastforce*
then show $(Fa\ (Ea\ a) \wedge b = put_{va}\ a\ (Fa\ (Ea\ a)) \vee \neg\ Fa\ (Ea\ a) \wedge b = put_{va}\ a\ (Ga\ (Ea\ a))) = (b = put_{va}\ a\ (Fa\ (Ea\ a) \vee \neg\ Fa\ (Ea\ a) \wedge Ga\ (Ea\ a)))$
by *meson*
qed

lemma *assign-cond-If-trop5*[*symbolic-exec-assign-trop*]:
assumes *1*: *mwb-lens v* **and** *2*: $v \sharp exp1$ **and** *3*: $v \sharp exp2$ **and** *4*: $v \sharp exp3$ **and** *5*: $v \sharp exp4$
shows $((v :== E);;$
$((v :== (trop\ F\ (\&v)\ exp1\ exp2)) \triangleleft trop\ F\ (\&v)\ exp1\ exp2 \triangleright_r (v :== (trop\ G\ (\&v)\ exp3\ exp4)))) =$
$(v :== (trop\ If\ (trop\ F\ E\ exp1\ exp2)\ (trop\ F\ E\ exp1\ exp2)\ (trop\ G\ E\ exp3\ exp4)))$
using *1 2 3 4 5*
proof (*rel-simp*, *transfer*)
fix $a :: 'a$ **and** $b :: 'a$ **and** $va :: bool \Longrightarrow 'a$ **and** $Fa :: bool \Rightarrow bool$ **and** $Ea :: 'a \Rightarrow bool$ **and** $Ga :: bool \Rightarrow bool$
have $Fa\ (Ea\ a) \longrightarrow (Fa\ (Ea\ a) \wedge b = put_{va}\ a\ (Fa\ (Ea\ a)) \vee \neg\ Fa\ (Ea\ a) \wedge b = put_{va}\ a\ (Ga\ (Ea\ a))) \wedge b = put_{va}\ a\ (Fa\ (Ea\ a) \vee \neg\ Fa\ (Ea\ a) \wedge Ga\ (Ea\ a)) \vee \neg\ b = put_{va}\ a\ (Fa\ (Ea\ a) \vee \neg\ Fa\ (Ea\ a) \wedge Ga\ (Ea\ a)) \wedge (\neg\ Fa\ (Ea\ a) \vee \neg\ b = put_{va}\ a\ (Fa\ (Ea\ a)))$
by *presburger*
then have $\neg\ ((\neg\ Fa\ (Ea\ a) \vee \neg\ b = put_{va}\ a\ (Fa\ (Ea\ a))) \wedge (Fa\ (Ea\ a) \vee \neg\ b = put_{va}\ a\ (Ga\ (Ea\ a)))) = (b = put_{va}\ a\ (Fa\ (Ea\ a) \vee \neg\ Fa\ (Ea\ a) \wedge Ga\ (Ea\ a)))$
by *fastforce*
then show $(Fa\ (Ea\ a) \wedge b = put_{va}\ a\ (Fa\ (Ea\ a)) \vee \neg\ Fa\ (Ea\ a) \wedge b = put_{va}\ a\ (Ga\ (Ea\ a))) = (b = put_{va}\ a\ (Fa\ (Ea\ a) \vee \neg\ Fa\ (Ea\ a) \wedge Ga\ (Ea\ a)))$
by *meson*
qed

A.1.3 Conditional Laws

In this section we introduce the algebraic laws of programming related to the conditional statement.

named-theorems *urel-cond*

lemma *cond-assoc*:
$(P \triangleleft b \triangleright (Q \triangleleft b \triangleright R)) = ((P \triangleleft b \triangleright Q) \triangleleft b \triangleright R)$
by *rel-auto*

lemma *cond-distr*[*urel-cond*]:
$((P \triangleleft b' \triangleright R) \triangleleft b \triangleright (Q \triangleleft b' \triangleright R)) = ((P \triangleleft b \triangleright Q) \triangleleft b' \triangleright R)$
by *rel-auto*

lemma *cond-ueq-distr*[*urel-cond*]:
$((P =_u Q) \triangleleft b \triangleright (R =_u S)) =$

$((P \triangleleft b \triangleright R) =_u (Q \triangleleft b \triangleright S))$
by *rel-auto*

lemma *cond-conj-distr*[*urel-cond*]:
$((P \wedge Q) \triangleleft b \triangleright (P \wedge S)) = (P \wedge (Q \triangleleft b \triangleright S))$
by *rel-auto*

lemma *cond-disj-distr* [*urel-cond*]:
$((P \vee Q) \triangleleft b \triangleright (P \vee S)) = (P \vee (Q \triangleleft b \triangleright S))$
by *rel-auto*

theorem *COND12*[*urel-cond*]:
$((C1 \triangleleft bexp2 \triangleright C3) \triangleleft bexp1 \triangleright (C2 \triangleleft bexp3 \triangleright C3)) =$
$((C1 \triangleleft bexp1 \triangleright C2) \triangleleft (bexp2 \triangleleft bexp1 \triangleright bexp3) \triangleright C3)$
by *rel-auto*

lemma *comp-cond-left-distr*:
$((P \triangleleft b \triangleright_r Q) \;;\; R) = ((P \;;\; R) \triangleleft b \triangleright_r (Q \;;\; R))$
by *rel-auto*

lemma *cond-var-subst-left*[*urel-cond*]:
assumes *vwb-lens x*
shows $(P[\![true/x]\!] \triangleleft \&x \triangleright Q) = (P \triangleleft \&x \triangleright Q)$
using *assms*
apply *rel-auto* **apply** *transfer*
using *vwb-lens.put-eq* **by** *fastforce*

lemma *cond-var-subst-right*[*urel-cond*]:
assumes *vwb-lens x*
shows $(P \triangleleft \&x \triangleright Q[\![false/x]\!]) = (P \triangleleft \&x \triangleright Q)$
using *assms*
apply *pred-auto* **apply** *transfer*
by (*metis* (*full-types*) *vwb-lens.put-eq*)

lemma *cond-var-split*[*urel-cond*]:
vwb-lens x $\Longrightarrow (P[\![true/x]\!] \triangleleft \&x \triangleright P[\![false/x]\!]) = P$
by (*rel-auto*, (*metis* (*full-types*) *vwb-lens.put-eq*)+)

lemma *cond-seq-left-distr*[*urel-comp*]:
$out\alpha \sharp b \Longrightarrow ((P \triangleleft b \triangleright Q) \;;\; R) = ((P \;;\; R) \triangleleft b \triangleright (Q \;;\; R))$
by *rel-auto*

lemma *cond-seq-right-distr*[*urel-comp*]:
$in\alpha \sharp b \Longrightarrow (P \;;\; (Q \triangleleft b \triangleright R)) = ((P \;;\; Q) \triangleleft b \triangleright (P \;;\; R))$

```
  by rel-auto
```

A.1.4 Sequential Laws

In this section we introduce the algebraic laws of programming related to the sequential composition of statements.

```
lemma seqr-exists-left[symbolic-exec]:
  ((∃ $x · P) ;; Q) = (∃ $x · (P ;; Q))
  by rel-auto

lemma seqr-exists-right[symbolic-exec]:
  (P ;; (∃ $x´ · Q)) = (∃ $x´ · (P ;; Q))
  by rel-auto

lemma seqr-left-zero [simp, symbolic-exec-ex]:
  false ;; P = false
  by pred-auto

lemma seqr-right-zero [simp, symbolic-exec-ex]:
  P ;; false = false
  by pred-auto

lemma seqr-or-distr[urel-comp]:
  (P ;; (Q ∨ R)) = ((P ;; Q) ∨ (P ;; R))
  by rel-auto

lemma seqr-unfold:
  (P ;; Q) = (∃ v · P⟦«v»/$Σ´⟧ ∧ Q⟦«v»/$Σ⟧)
  by rel-auto

lemma seqr-middle:
  assumes vwb-lens x
  shows (P ;; Q) = (∃ v · P⟦«v»/$x´⟧ ;; Q⟦«v»/$x⟧)
  using assms
  apply (rel-auto robust)
  apply (rename-tac xa P Q a b y)
  apply (rule-tac x=get_xa y in exI)
  apply (rule-tac x=y in exI)
  apply (simp)
done

lemma seqr-left-one-point[urel-comp]:
  assumes vwb-lens x
  shows ((P ∧ $x´ =_u «v») ;; Q) = (P⟦«v»/$x´⟧ ;; Q⟦«v»/$x⟧)
```

using *assms*
by (*rel-auto*, *metis vwb-lens-wb wb-lens.get-put*)

lemma *seqr-right-one-point*[*urel-comp*]:
assumes *vwb-lens x*
shows $(P \;;;\; (\$x =_u «v» \wedge Q)) = (P\llbracket «v»/\$x' \rrbracket \;;;\; Q\llbracket «v»/\$x \rrbracket)$
using *assms*
by (*rel-auto*, *metis vwb-lens-wb wb-lens.get-put*)

lemma *seqr-insert-ident-left*[*urel-comp*]:
assumes *vwb-lens x* $\$x' \sharp P$ $\$x \sharp Q$
shows $((\$x' =_u \$x \wedge P) \;;;\; Q) = (P \;;;\; Q)$
using *assms*
by (*rel-auto*, *meson vwb-lens-wb wb-lens-weak weak-lens.put-get*)

lemma *seqr-insert-ident-right*[*urel-comp*]:
assumes *vwb-lens x* $\$x' \sharp P$ $\$x \sharp Q$
shows $(P \;;;\; (\$x' =_u \$x \wedge Q)) = (P \;;;\; Q)$
using *assms*
by (*rel-auto*, *metis* (*no-types*, *hide-lams*) *vwb-lens-def wb-lens-def weak-lens.put-get*)

lemma *seq-var-ident-lift*[*urel-comp*]:
assumes *vwb-lens x* $\$x' \sharp P$ $\$x \sharp Q$
shows $((\$x' =_u \$x \wedge P) \;;;\; (\$x' =_u \$x \wedge Q)) = (\$x' =_u \$x \wedge (P \;;;\; Q))$
using *assms*
by (*rel-auto*, *metis* (*no-types*, *lifting*) *vwb-lens-wb wb-lens-weak weak-lens.put-get*)

A.1.5 While laws

In this section we introduce the algebraic laws of programming related to the while statement.

theorem *while-unfold*:
$\mathit{while}\ b\ \mathit{do}\ P\ \mathit{od} = ((P \;;;\; \mathit{while}\ b\ \mathit{do}\ P\ \mathit{od}) \triangleleft b \triangleright_r II)$
proof –
have *m*:*mono* $(\lambda X.\ (P \;;;\; X) \triangleleft b \triangleright_r II)$
by (*auto intro*: *monoI seqr-mono cond-mono*)
have $(\mathit{while}\ b\ \mathit{do}\ P\ \mathit{od}) = (\nu\ X \cdot (P \;;;\; X) \triangleleft b \triangleright_r II)$
by (*simp add*: *while-def*)
also have ... $= ((P \;;;\; (\nu\ X \cdot (P \;;;\; X) \triangleleft b \triangleright_r II)) \triangleleft b \triangleright_r II)$
by (*subst lfp-unfold*, *simp-all add*: *m*)
also have ... $= ((P \;;;\; \mathit{while}\ b\ \mathit{do}\ P\ \mathit{od}) \triangleleft b \triangleright_r II)$
by (*simp add*: *while-def*)
finally show *?thesis* **.**
qed

lemma *while-true*:
  shows (*while true do P od*) = *false*
  apply (*simp add*: *while-def alpha*)
  apply (*rule antisym*)
  apply (*simp-all*)
  apply (*rule lfp-lowerbound*)
  apply (*simp*)
done

lemma *while-false*:
  shows (*while false do P od*) = *II*
proof −
  have (*while false do P od*) = (*P* ;; *while false do P od*) $\triangleleft$ *false* $\triangleright_r$ *II*
    using *while-unfold*[*of - P*] **by** *simp*
  also have ... = *II* **by** (*simp add*: *aext-false*)
  finally show *?thesis* **.**
qed

lemma *while-inv-unfold*:
  (*while b invr p do P od*) = ((*P* ;; *while b invr p do P od*) $\triangleleft$ *b* $\triangleright_r$ *II*)
  unfolding *while-inv-def* **using** *while-unfold*
  by *auto*

theorem *while-bot-unfold*:
  $while_\perp$ *b do P od* = ((*P* ;; $while_\perp$ *b do P od*) $\triangleleft$ *b* $\triangleright_r$ *II*)
proof −
  have *m*:*mono* (λX. (*P* ;; *X*) $\triangleleft$ *b* $\triangleright_r$ *II*)
    by (*auto intro*: *monoI seqr-mono cond-mono*)
  have ($while_\perp$ *b do P od*) = (μ *X* • (*P* ;; *X*) $\triangleleft$ *b* $\triangleright_r$ *II*)
    by (*simp add*: *while-bot-def*)
  also have ... = ((*P* ;; (μ *X* • (*P* ;; *X*) $\triangleleft$ *b* $\triangleright_r$ *II*)) $\triangleleft$ *b* $\triangleright_r$ *II*)
    by (*subst gfp-unfold*, *simp-all add*: *m*)
  also have ... = ((*P* ;; $while_\perp$ *b do P od*) $\triangleleft$ *b* $\triangleright_r$ *II*)
    by (*simp add*: *while-bot-def*)
  finally show *?thesis* **.**
qed

theorem *while-bot-false*: $while_\perp$ *false do P od* = *II*
  by (*simp add*: *while-bot-def mu-const alpha*)

theorem *while-bot-true*: $while_\perp$ *true do P od* = (μ *X* • *P* ;; *X*)
  by (*simp add*: *while-bot-def alpha*)

An infinite loop with a feasible body corresponds to a program error (non-termination).

theorem *while-infinite*: *P* ;; $true_h$ = *true* $\Longrightarrow$ $while_\perp$ *true do P od* = *true*

apply (*simp add*: *while-bot-true*)
apply (*rule antisym*)
apply (*simp*)
apply (*rule gfp-upperbound*)
apply (*simp*)
done

A.1.6 assume and assert laws

lemma *assume-twice*[*urel-comp*]: $(b^{\top} ;; c^{\top}) = (b \wedge c)^{\top}$
by *rel-auto*

lemma *assert-twice*[*urel-comp*]: $(b_{\perp} ;; c_{\perp}) = (b \wedge c)_{\perp}$
by *rel-auto*

A.1.7 Refinement rules

lemma *pre-weak-rel*:
assumes ‘$Pre \Rightarrow I$‘
and $(I \Rightarrow Post) \sqsubseteq P$
shows $(Pre \Rightarrow Post) \sqsubseteq P$
using *assms*
by(*rel-auto*)

lemma *post-str-rel*:
$(p{\Rightarrow}q) \sqsubseteq P \Longrightarrow$ ‘$q{\Rightarrow}r$‘ $\Longrightarrow (p{\Rightarrow}r) \sqsubseteq P$
by *pred-blast*

lemma *cond-refine-rel*:
assumes $(b \wedge p \Rightarrow q) \sqsubseteq C_1$ **and** $(\neg b \wedge p \Rightarrow q) \sqsubseteq C_2$
shows $(p \Rightarrow q) \sqsubseteq (C_1 \triangleleft b \triangleright C_2)$
using *assms* **by** *rel-auto*

lemma *cond-refine-pred*:
assumes $(\lceil b \wedge p \rceil_{<} \Rightarrow \lceil q \rceil_{>}) \sqsubseteq C_1$ **and** $(\lceil \neg b \wedge p \rceil_{<} \Rightarrow \lceil q \rceil_{>}) \sqsubseteq C_2$
shows $(\lceil p \rceil_{<} \Rightarrow \lceil q \rceil_{>}) \sqsubseteq (C_1 \triangleleft \lceil b \rceil_{<} \triangleright C_2)$
using *assms* **by** *rel-auto*

lemma *seq-refine-pred*:
assumes $(\lceil p \rceil_{<} \Rightarrow \lceil s \rceil_{>}) \sqsubseteq f$ **and** $(\lceil s \rceil_{<} \Rightarrow \lceil q \rceil_{>}) \sqsubseteq fa$
shows $(\lceil p \rceil_{<} \Rightarrow \lceil q \rceil_{>}) \sqsubseteq (f ;; fa)$
using *assms* **by** *rel-auto*

lemma *seq-refine-unrest*:
assumes $out\alpha \sharp p$ $in\alpha \sharp q$

assumes $(p \Rightarrow \lceil s \rceil_{>}) \sqsubseteq f$ **and** $(\lceil s \rceil_{<} \Rightarrow q) \sqsubseteq fa$
shows $(p \Rightarrow q) \sqsubseteq (f \;;; fa)$
using *assms* **by** *rel-blast*

lemmas *skip-refine′* = *post-str-rel*[*OF skip-r-refine-orig*]

end

A.2 Relational Hoare Calculus

theory *utp-hoare*
imports *../../AlgebraicLaws/Rel&Des/Algebraic-Laws*
begin

named-theorems *hoare* **and** *hoare-safe*

method *hoare-split* **uses** *hr* =
((*simp add*: *assigns-r-comp usubst unrest*)?, — Eliminate assignments where possible
(*auto*
intro: *hoare intro*!: *hoare-safe hr*
simp add: *assigns-r-comp conj-comm conj-assoc usubst unrest*))[*1*] — Apply Hoare logic laws

method *hoare-auto* **uses** *hr* = (*hoare-split hr*: *hr*; *rel-auto?*)

A.2.1 Hoare triple definition

A Hoare triple is represented by a precondition P, a postcondition Q, and a program C. It says that, whenever P holds on the initial state, Q must hold on the final state after execution of C.

definition *hoare-r* :: $'\alpha\ cond \Rightarrow {'\alpha}\ hrel \Rightarrow {'\alpha}\ cond \Rightarrow bool$ $(\{\!|\text{-}|\!\}\text{-}\{\!|\text{-}|\!\}_u)$ **where**
$\{\!|p|\!\}Q\{\!|r|\!\}_u = ((\lceil p \rceil_{<} \Rightarrow \lceil r \rceil_{>}) \sqsubseteq Q)$

declare *hoare-r-def* [*upred-defs*]

lemma *hoare-true* [*hoare*]: $\{\!|p|\!\}C\{\!|true|\!\}_u$
by *rel-auto*

lemma *hoare-false* [*hoare*]: $\{\!|false|\!\}C\{\!|q|\!\}_u$
by *rel-auto*

A.2.2 Hoare for Consequence

lemma *hoare-r-conseq* [*hoare*]:
assumes ‘$p_1 \Rightarrow p_2$‘ **and** $\{\!|p_2|\!\}C\{\!|q_2|\!\}_u$ **and** ‘$q_2 \Rightarrow q_1$‘

shows $\{\!|p_1|\!\}C\{\!|q_1|\!\}_u$
by (*insert assms*) *rel-auto*

A.2.3 Precondition strengthening

lemma *hoare-pre-str*[*hoare*]:
assumes ‘$p_1 \Rightarrow p_2$‘ **and** $\{\!|p_2|\!\}C\{\!|q|\!\}_u$
shows $\{\!|p_1|\!\}C\{\!|q|\!\}_u$
by (*insert assms*) *rel-auto*

A.2.4 Post-condition weakening

lemma *hoare-post-weak*[*hoare*]:
assumes $\{\!|p|\!\}C\{\!|q_2|\!\}_u$ **and** ‘$q_2 \Rightarrow q_1$‘
shows $\{\!|p|\!\}C\{\!|q_1|\!\}_u$
by (*insert assms*) *rel-auto*

A.2.5 Hoare and assertion logic

lemma *hoare-r-conj* [*hoare*]:
assumes$\{\!|p|\!\}C\{\!|r|\!\}_u$ **and** $\{\!|p|\!\}C\{\!|s|\!\}_u$
shows $\{\!|p|\!\}C\{\!|r \wedge s|\!\}_u$
by (*insert assms*) *rel-auto*

A.2.6 Hoare SKIP

lemma *skip-hoare-r* [*hoare-safe*]: $\{\!|p|\!\}II\{\!|p|\!\}_u$
by *rel-auto*

A.2.7 Hoare for assignment

lemma *assigns-hoare-r* [*hoare-safe*]:
assumes‘$p \Rightarrow \sigma \dagger q$‘
shows $\{\!|p|\!\}\langle\sigma\rangle_a\{\!|q|\!\}_u$
by (*insert assms*) *rel-auto*

lemma *assigns-hoare-r′* [*hoare*]: $\{\!|\sigma \dagger p|\!\}\langle\sigma\rangle_a\{\!|p|\!\}_u$
by *rel-auto*

lemma *assigns-naive-rule*:
assumes $x\sharp e$ **and** *weak-lens x*
shows $\{\!|p|\!\}x :== e\{\!|\&x =_u e|\!\}_u$
using *assms*
by *pred-simp*

lemma *assigns-floyd-r* [*hoare*]:
assumes ‹*vwb-lens x*›
shows ‹$\{\!|p|\!\}x :== e\{\!|\exists\, v \cdot p[\![«v»/x]\!] \wedge \&x =_u e[\![«v»/x]\!]|\!\}_u$›
apply (*insert assms*)
apply *pred-simp*
apply *transfer*
apply (*rule-tac x* = ‹get_x *a*› **in** *exI*)

apply *auto*
done

A.2.8 Hoare for Sequential Composition

lemma *seq-hoare-r*:
assumes$\{\!|p|\!\}C_1\{\!|s|\!\}_u$ **and** $\{\!|s|\!\}C_2\{\!|r|\!\}_u$
shows$\{\!|p|\!\}C_1$;; $C_2\{\!|r|\!\}_u$
by (*insert assms*) *rel-auto*

lemma *seq-hoare-invariant* [*hoare-safe*]:
assumes $\{\!|p|\!\}Q_1\{\!|p|\!\}_u$ **and** $\{\!|p|\!\}Q_2\{\!|p|\!\}_u$
shows $\{\!|p|\!\}Q_1$;; $Q_2\{\!|p|\!\}_u$
using *assms*
by (*auto simp*: *seq-hoare-r*)

lemma *seq-hoare-stronger-pre-1* [*hoare-safe*]:
assumes $\{\!|p \wedge q|\!\}Q_1\{\!|p \wedge q|\!\}_u$ **and** $\{\!|p \wedge q|\!\}Q_2\{\!|q|\!\}_u$
shows $\{\!|p \wedge q|\!\}Q_1$;; $Q_2\{\!|q|\!\}_u$
using *assms*
by (*auto simp*: *seq-hoare-r*)

lemma *seq-hoare-stronger-pre-2* [*hoare-safe*]:
assumes $\{\!|p \wedge q|\!\}Q_1\{\!|p \wedge q|\!\}_u$ **and** $\{\!|p \wedge q|\!\}Q_2\{\!|p|\!\}_u$
shows $\{\!|p \wedge q|\!\}Q_1$;; $Q_2\{\!|p|\!\}_u$
using *assms*
by (*auto simp*: *seq-hoare-r*)

lemma *seq-hoare-inv-r-2* [*hoare*]:
assumes $\{\!|p|\!\}Q_1\{\!|q|\!\}_u$ **and** $\{\!|q|\!\}Q_2\{\!|q|\!\}_u$
shows $\{\!|p|\!\}Q_1$;; $Q_2\{\!|q|\!\}_u$
using *assms*
by (*auto simp*: *seq-hoare-r*)

lemma *seq-hoare-inv-r-3* [*hoare*]:
assumes $\{\!|p|\!\}Q_1\{\!|p|\!\}_u$ **and** $\{\!|p|\!\}Q_2\{\!|q|\!\}_u$
shows $\{\!|p|\!\}Q_1$;; $Q_2\{\!|q|\!\}_u$

using *assms*
by (*auto simp*: *seq-hoare-r*)

A.2.9 Hoare for Conditional

lemma *cond-hoare-r* [*hoare-safe*]:
assumes $\{\!|b \wedge p|\!\}C_1\{\!|q|\!\}_u$ **and** $\{\!|\neg b \wedge p|\!\}C_2\{\!|q|\!\}_u$
shows $\{\!|p|\!\}C_1 \triangleleft\, b \,\triangleright_r\, C_2\{\!|q|\!\}_u$
by (*insert assms*) *rel-auto*

A.2.10 Hoare for assert

lemma *assert-hoare-r* [*hoare-safe*]:
assumes $\{\!|c \wedge p|\!\}II\{\!|q|\!\}_u$ **and** $\{\!|\neg c \wedge p|\!\}true\{\!|q|\!\}_u$
shows $\{\!|p|\!\}c_{\bot}\{\!|q|\!\}_u$
unfolding *rassert-def* **using** *assms cond-hoare-r* [*of c p - q*]
by *auto*

A.2.11 Hoare for assume

lemma *assume-hoare-r* [*hoare-safe*]:
assumes $\{\!|c \wedge p|\!\}II\{\!|q|\!\}_u$ **and** $\{\!|\neg c \wedge p|\!\}false\{\!|q|\!\}_u$
shows $\{\!|p|\!\}c^{\top}\{\!|q|\!\}_u$
unfolding *rassume-def* **using** *assms cond-hoare-r* [*of c p - q*]
by *auto*

A.2.12 Hoare for While-loop

lemma *while-hoare-r* [*hoare-safe*]:
assumes $\{\!|p \wedge b|\!\}C\{\!|p|\!\}_u$
shows $\{\!|p|\!\}while\ b\ do\ C\ od\{\!|\neg b \wedge p|\!\}_u$
using *assms*
apply (*simp add*: *while-def hoare-r-def*)
apply (*rule-tac lfp-lowerbound*)
apply(*rel-auto*)
done

lemma *while-hoare-r′* [*hoare-safe*]:
assumes $\{\!|p \wedge b|\!\}C\{\!|p|\!\}_u$ **and** *‘p ∧ ¬b ⇒ q‘*
shows $\{\!|p|\!\}while\ b\ do\ C\ od\{\!|q|\!\}_u$
using *assms*
by (*metis conj-comm hoare-r-conseq p-imp-p taut-true while-hoare-r*)

lemma *while-invr-hoare-r* [*hoare-safe*]:
assumes $\{\!|p \wedge b|\!\}C\{\!|p|\!\}_u$ **and** *‘pre ⇒ p‘* **and** *‘(¬b ∧ p) ⇒ post‘*

shows $\{\!|pre|\!\}$*while b invr p do C od*$\{\!|post|\!\}_u$
by (*metis assms hoare-r-conseq while-hoare-r while-inv-def*)

end

A.3 Strongest Postcondition

theory *utp-sp*
imports *../../Isabelle−UTP/utp/utp-wp*

begin

named-theorems *sp*

method *sp-tac* = (*simp add*: *sp*)

consts
usp :: $'a \Rightarrow 'b \Rightarrow 'c$ (**infix** *sp 60*)

definition *sp-upred* :: $'\alpha$ *cond* $\Rightarrow$ $('\alpha,\ '\beta)$ *rel* $\Rightarrow$ $'\beta$ *cond* **where**
sp-upred p Q = $\lfloor(\lceil p\rceil_{>}\ ;;\ Q) :: ('\alpha,\ '\beta)\ rel\rfloor_{>}$

adhoc-overloading
usp sp-upred

declare *sp-upred-def* [*upred-defs*]

lemma *sp-false* [*sp*]: *p sp false* = *false*
by (*rel-simp*)

lemma *sp-true* [*sp*]: $q \neq$ *false* $\Longrightarrow$ *q sp true* = *true*
by (*rel-auto*)

lemma *sp-assigns-r* [*sp*]:
vwb-lens x $\Longrightarrow$ (*p sp x* :== *e*) = $(\exists\ v \cdot p⟦«v»/x⟧ \wedge \&x =_u e⟦«v»/x⟧)$
apply (*rel-simp*)
apply *transfer*
apply *auto*
apply (*rule-tac* $x = \langle get_x\ y\rangle$ **in** *exI*)
apply *simp*
apply (*metis vwb-lens.put-eq*)
done

lemma *it-is-post-condition*: $\{\!|p|\!\}C\{\!|p\ sp\ C|\!\}_u$

by *rel-blast*

lemma *it-is-the-strongest-post*: $`p\ sp\ C \Rightarrow Q` \Longrightarrow \{\!|p|\!\}C\{\!|Q|\!\}_u$
by *rel-blast*

lemma *so*: $`p\ sp\ C \Rightarrow Q` = \{\!|p|\!\}C\{\!|Q|\!\}_u$
by *rel-blast*

theorem *sp-hoare-link*: $\{\!|p|\!\}Q\{\!|r|\!\}_u \longleftrightarrow (r \sqsubseteq p\ sp\ Q)$
by *rel-auto*

theorem *sp-eq-intro*: $[\![\bigwedge r.\ r\ sp\ P = r\ sp\ Q]\!] \Longrightarrow P = Q$
by (*rel-auto robust*, *fastforce+*)

lemma *wp-sp-sym*: $`prog\ wp\ (true\ sp\ prog)`$
by *rel-auto*

lemma *it-is-pre-condition*: $\{\!|C\ wp\ Q|\!\}C\{\!|Q|\!\}_u$
by *rel-blast*

lemma *it-is-the-weakest-pre*: $`P \Rightarrow C\ wp\ Q` = \{\!|P|\!\}C\{\!|Q|\!\}_u$
by *rel-blast*

end

A.4 SP VCG

theory *VCG-rel-Floyd*
imports *../../Midend−IVL/Isabelle−UTP−Extended/hoare/HoareLogic/PartialCorrectness/utp-hoare*
begin

The below definition helps in asserting independence for a group of lenses, as otherwise the number of assumptions required increases greatly. Unfortunately, it is not usable with lenses of different types as Isabelle does not allow heterogenous lists; element types must be unifiable.

definition $\langle lens\text{-}indep\text{-}all\ lenses \longleftrightarrow (\forall\, l \in set\ lenses.\ vwb\text{-}lens\ l \wedge eff\text{-}lens\ l) \wedge$
$(\forall\, i\ j.\ i < length\ lenses \wedge j < length\ lenses \wedge$
$i \neq j \longrightarrow lenses!i \bowtie lenses!j)\rangle$
lemma *lens-indep-all-alt*:
$\langle lens\text{-}indep\text{-}all\ lenses \longleftrightarrow (\forall\, l \in set\ lenses.\ vwb\text{-}lens\ l \wedge eff\text{-}lens\ l) \wedge$
$distinct\ lenses \wedge$
$(\forall\, a \in set\ lenses.\ \forall\, b \in set\ lenses.\ a \neq b \longrightarrow a \bowtie b)\rangle$
unfolding *lens-indep-all-def distinct-conv-nth*

apply (*safe*; *simp?*)
apply (*metis lens-indep-quasi-irrefl nth-mem vwb-lens-wb*)
apply (*metis in-set-conv-nth*)
done

named-theorems *hoare-rules*

lemma *assert-hoare-r'*[*hoare-rules*]:
assumes $\langle `p \Rightarrow c` \rangle$
shows $\langle \{\!|p|\!\} c_{\bot} \{\!|p \wedge c|\!\}_u \rangle$
using *assms*
by (*metis assert-hoare-r conj-comm hoare-false refBy-order skip-hoare-r utp-pred-laws.inf.orderE utp-pred-laws.inf-compl-bot-left1*)

lemma *assume-hoare-r'*[*hoare-rules*]:
shows $\langle \{\!|p|\!\} c^{\top} \{\!|p \wedge c|\!\}_u \rangle$
by *rel-simp*

lemma *cond-hoare-r'*:
assumes $\langle \{\!|b \wedge p|\!\} C_1 \{\!|q|\!\}_u \rangle$ **and** $\langle \{\!|\neg b \wedge p|\!\} C_2 \{\!|s|\!\}_u \rangle$
shows $\langle \{\!|p|\!\} \mathit{if}_u\ b\ \mathit{then}\ C_1\ \mathit{else}\ C_2\ \{\!|q \vee s|\!\}_u \rangle$
by (*insert assms*, *rel-auto*)

lemma *cond-assert-hoare-r*[*hoare-rules*]:
assumes $\langle \{\!|b \wedge p|\!\} C_1 \{\!|q|\!\}_u \rangle$
and $\langle \{\!|\neg b \wedge p|\!\} C_2 \{\!|s|\!\}_u \rangle$
and $\langle `q \Rightarrow A` \rangle$
and $\langle `s \Rightarrow A` \rangle$
and $\langle \{\!|A|\!\} P \{\!|A'|\!\}_u \rangle$
shows $\langle \{\!|p|\!\} (\mathit{if}_u\ b\ \mathit{then}\ C_1\ \mathit{else}\ C_2);;\ A_{\bot};;\ P \{\!|A'|\!\}_u \rangle$
apply (*insert assms*)
apply (*rule hoare-post-weak*)
apply (*rule cond-hoare-r' seq-hoare-r*|*assumption*)+
apply (*rule assert-hoare-r'*)
using *impl-disjI* **apply** *blast*
apply (*rule hoare-pre-str*[**where** $p_2 = A$])
apply (*simp add*: *disj-comm impl-alt-def*)
apply *assumption*
apply *pred-auto*
done

lemma *cond-assert-last-hoare-r*[*hoare-rules*]:
assumes $\langle \{\!|b \wedge p|\!\} C_1 \{\!|q|\!\}_u \rangle$
and $\langle \{\!|\neg b \wedge p|\!\} C_2 \{\!|s|\!\}_u \rangle$
and $\langle `q \Rightarrow A` \rangle$

```
    and ‹‘s ⇒ A‘›
  shows ‹⦃p⦄(ifᵤ b then C₁ else C₂);; A⊥⦃A⦄ᵤ›
  apply (insert assms)
  apply (rule hoare-post-weak)
   apply (rule cond-hoare-r' seq-hoare-r|assumption)+
   apply (rule assert-hoare-r')
  using impl-disjI apply blast
  using refBy-order by fastforce

lemma while-invr-hoare-r'[hoare-rules]:
  assumes ‹‘pre ⇒ p‘› and ‹⦃p ∧ b⦄C⦃p'⦄ᵤ› and ‹‘p' ⇒ p‘›
  shows ‹⦃pre⦄while b invr p do C od⦃¬b ∧ p⦄ᵤ›
  by (metis while-inv-def assms hoare-post-weak hoare-pre-str while-hoare-r)

lemma nu-refine-intro[hoare-rules]:
  assumes ‹(C ⇒ S) ⊑ F(C ⇒ S)›
  shows ‹(C ⇒ S) ⊑ ν F›
  using assms
  by (simp add: lfp-lowerbound)

lemma nu-hoare-basic-r[hoare-rules]:
  assumes ‹⋀p. ⦃P⦄p⦃Q⦄ᵤ ⟹ ⦃P⦄F p⦃Q⦄ᵤ›
  shows ‹⦃P⦄ν F⦃Q⦄ᵤ›
  using assms unfolding hoare-r-def
  by (rule nu-refine-intro) auto

definition annot-rec ::
  ‹'a upred ⇒ ((bool, 'a) hexpr ⇒ (bool, 'a) hexpr) ⇒ (bool, 'a) hexpr› where
  ‹annot-rec P F ≡ ν F›

syntax
  -nu-annot :: ‹pttrn ⇒ logic ⇒ logic ⇒ logic› (ν - [-] · - [0, 10] 10)

translations
  ν X [P] · p ⇌ CONST annot-rec P (λX. p)

lemma nu-hoare-r:
  assumes PRE: ‹‘P' ⇒ P‘›
  assumes IH: ‹⋀p. ⦃P⦄p⦃Q⦄ᵤ ⟹ ⦃P⦄F p⦃Q⦄ᵤ›
  shows ‹⦃P'⦄ν F⦃Q⦄ᵤ›
  apply (rule hoare-pre-str[OF PRE])
  using IH
  unfolding hoare-r-def
  by (rule nu-refine-intro) (rule order-refl)
```

lemma *nu-hoare-annot-r*[*hoare-rules*]:
assumes *PRE*: $\langle `P' \Rightarrow P` \rangle$
assumes *IH*: $\langle \bigwedge p.\ \{\!|P|\!\}p\{\!|Q|\!\}_u \Longrightarrow \{\!|P|\!\}F\ p\{\!|Q|\!\}_u \rangle$
shows $\langle \{\!|P'|\!\}$*annot-rec* $P\ F\{\!|Q|\!\}_u\rangle$
using *nu-hoare-r assms* **unfolding** *annot-rec-def* **.**

lemmas [*hoare-rules*] =
cond-hoare-r′ — Needs to come after annotated cond check
assigns-floyd-r
skip-hoare-r
seq-hoare-r

named-theorems *vcg-simps*
lemmas [*vcg-simps*] =
lens-indep.lens-put-irr1
lens-indep.lens-put-irr2
lens-indep-all-alt

named-theorems *hoare-rules-extra* **and** *vcg-dests*

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*+

end

Appendix B

Proof Helpers

This chapter contains helper theories and some addons to Isabelle/UTP to lift more HOL functions to the UTP level.

B.1 Binary Operations

```
theory BitOps
imports
  Main
  ~~/src/HOL/Word/Bits-Bit
begin
```

Bits of BitOperations.thy and MoreWord.thy from the VAMP machine model theories, Copyright 2003-2009 Kara Abdul-Qadar, Matthias Daum, Mark Hillebrand, Dirk Leinenbach, Elena Petrova, Mareike Schmidt, Alexandra Tsyban, and Martin Wildmoser and licensed under the German-Jurisdiction Creative Commons Attribution Non-commercial Share Alike 2.0 License (https://creativecommons.org/licenses/by-nc/2.0/de/legalcode), simplified English version at https://creativecommons.org/licenses/by-nc/2.0/de/deed.en.

The only changes made (by Joshua A. Bockenek in 2017) were spacing adjustments and usage of pretty-printing characters (like ⇒ instead of =>, cartouches, etc.), plus some minor syntactic tweaks that do not affect the semantics. For now the associated lemmas have been left out, but those may be necessary for any proofs involving bit operations. This may eventually be replaced given the reliance on a non-commercial-use license.

B.1.1 Building blocks

```
definition bv-msb :: ‹bit list ⇒ bit› where
  ‹bv-msb w = (if w = [] then 0 else hd w)›

definition bv-extend :: ‹[nat, bit, bit list] ⇒ bit list› where
```

```
  ⟨bv-extend i b w = (replicate (i − length w) b) @ w⟩

fun rem-initial :: ⟨bit ⇒ bit list ⇒ bit list⟩ where
  ⟨rem-initial b [] = []⟩
| ⟨rem-initial b (x # xs) = (if b = x then rem-initial b xs else x # xs)⟩

abbreviation ⟨norm-unsigned ≡ rem-initial 0⟩

primrec norm-signed :: ⟨bit list ⇒ bit list⟩ where
  norm-signed-Nil:  ⟨norm-signed [] = []⟩
| norm-signed-Cons: ⟨norm-signed (b # bs) =
    (case b of 1 ⇒ b # rem-initial b bs
             | 0 ⇒ if norm-unsigned bs = [] then [] else b # norm-unsigned bs)⟩

fun nat-to-bv-helper :: ⟨nat ⇒ bit list ⇒ bit list⟩ where
  Zeroo: ⟨nat-to-bv-helper 0 bs = bs⟩
| Succ:  ⟨nat-to-bv-helper (Suc n) bs =
        (nat-to-bv-helper (Suc n div 2) ((if Suc n mod 2 = 0
                                          then (0::bit)
                                          else (1::bit)) # bs))⟩

definition nat-to-bv :: ⟨nat ⇒ bit list⟩ where
  ⟨nat-to-bv n = nat-to-bv-helper n []⟩

abbreviation ⟨bv-not ≡ map (λx::bit. NOT x)⟩

definition int-to-bv :: ⟨int ⇒ bit list⟩ where
  ⟨int-to-bv n = (if 0 ≤ n
                  then norm-signed (0 # nat-to-bv (nat n))
                  else norm-signed (bv-not (0 # nat-to-bv (nat (−n − 1)))))⟩

primrec bitval :: ⟨bit ⇒ nat⟩ where
  ⟨bitval 0 = 0⟩
| ⟨bitval 1 = 1⟩

definition bv-to-nat :: ⟨bit list ⇒ nat⟩ where
  ⟨bv-to-nat = foldl (%bn b. 2 ∗ bn + bitval b) 0⟩

definition bv-to-int :: ⟨bit list ⇒ int⟩ where
  ⟨bv-to-int w =
    (case bv-msb w of 0 ⇒ int (bv-to-nat w)
                    | 1 ⇒ − int (bv-to-nat (bv-not w) + 1))⟩

— convert int to bv of a desired length
definition int2bvn :: ⟨nat ⇒ int ⇒ bit list⟩ where
```

```
  ‹int2bvn n a = (let v = int-to-bv a in drop (length v − n) (bv-extend n (bv-msb v) v))›
```

— convert nat to bv of a desired length

```
definition nat2bvn :: ‹nat ⇒ nat ⇒ bit list› where
  ‹nat2bvn n a = (let v = nat-to-bv a in drop (length v − n) (bv-extend n (0::bit) v))›
```

B.1.2 Base definitions for AND/OR/XOR

```
definition s-bitop :: ‹(bit ⇒ bit ⇒ bit) ⇒ int ⇒ int ⇒ int› where
  ‹s-bitop f x y ≡ let v = int-to-bv x; w = int-to-bv y in
     bv-to-int (map (λ (a, b). f a b)
                   (zip (bv-extend (length w) (bv-msb v) v)
                        (bv-extend (length v) (bv-msb w) w)))›

definition u-bitop :: ‹(bit ⇒ bit ⇒ bit) ⇒ nat ⇒ nat ⇒ nat› where
  ‹u-bitop f x y ≡ let v = nat-to-bv x; w = nat-to-bv y in
     bv-to-nat (map (λ (a, b). f a b)
                   (zip (bv-extend (length w) (0::bit) v)
                        (bv-extend (length v) (0::bit) w)))›
```

B.1.3 Bit shifting

```
definition ‹s-lsh x w a ≡ bv-to-int ((drop a (int2bvn w x)) @ replicate a 0)›
definition ‹u-lsh x w a ≡ bv-to-nat ((drop a (nat2bvn w x)) @ replicate a 0)›
definition ‹s-rsh x w a ≡ if a > 0
                          then int (bv-to-nat (take (w − a) (int2bvn w x)))
                          else x›
definition ‹u-rsh x w a ≡ bv-to-nat (take (length (nat-to-bv x) − a) (nat-to-bv x))›
```

B.1.4 Negation

This subsection covers both plain bitwise NOT and two's-complement negation (only needed for unsigned/nat values?)

```
definition ‹s-not w x ≡ bv-to-int (bv-not (int2bvn w x))›
definition ‹u-not w x ≡ bv-to-nat (bv-not (nat2bvn w x))›
definition ‹u-neg w x ≡ 1 + u-not w x›

end
```

B.2 Syntax extensions for UTP

```
theory utp-extensions
imports
  BitOps
```

../../Isabelle−UTP/utp/utp
~~/src/HOL/Library/Multiset
begin
recall-syntax — Fixes notation issue with inclusion of HOL libraries.

B.2.1 Notation

We need multisets for concise list invariants for sorting. Also, int/nat conversion is sometimes needed as some loop methods mix array indices and loop variables (which sometimes rely on going below 0 for termination). Bitwise operations and record access/update are included for completeness.

A helper function for record updating.

lift-definition *rec-update-wrapper* :: ‹('a, 'α) uexpr ⇒ ('a ⇒ 'a, 'α) uexpr› **is**
‹λv s -. v s› **.**

syntax
-umset :: ‹('a list, 'α) uexpr ⇒ ('a multiset, 'α) uexpr› ($mset_u$'(-'))
-unat :: ‹(nat, 'α) uexpr ⇒ (int, 'α) uexpr› (int_u'(-'))
-uint :: ‹(int, 'α) uexpr ⇒ (nat, 'α) uexpr› (nat_u'(-'))
-uapply-rec :: ‹('a, 'α) uexpr ⇒ utuple-args ⇒ ('b, 'α) uexpr› (-⦇-⦈$_r$ [999,0] 999)
-uupd-rec :: ‹('a, 'α) uexpr ⇒ (('b ⇒ 'b) ⇒ 'a ⇒ 'a) ⇒ ('b, 'α) uexpr ⇒ ('a, 'α) uexpr› (-/'(-/↦/ -')$_r$ [900,0,0] 900)
-ubs-and :: ‹(int, 'α) uexpr ⇒ (int, 'α) uexpr ⇒ (int, 'α) uexpr› (**infixl** $\wedge_{bs}$ 85)
-ubu-and :: ‹(nat, 'α) uexpr ⇒ (nat, 'α) uexpr ⇒ (nat, 'α) uexpr› (**infixl** $\wedge_{bu}$ 85)
-ubs-or :: ‹(int, 'α) uexpr ⇒ (int, 'α) uexpr ⇒ (int, 'α) uexpr› (**infixl** $\vee_{bs}$ 80)
-ubu-or :: ‹(nat, 'α) uexpr ⇒ (nat, 'α) uexpr ⇒ (nat, 'α) uexpr› (**infixl** $\vee_{bu}$ 80)
-ubs-lsh :: ‹(int, 'α) uexpr ⇒ (nat, 'α) uexpr ⇒ (nat, 'α) uexpr ⇒ (int, 'α) uexpr› (- $\ll_{s'/-}$ - [100,100,101] 100)
-ubu-lsh :: ‹(nat, 'α) uexpr ⇒ (nat, 'α) uexpr ⇒ (nat, 'α) uexpr ⇒ (nat, 'α) uexpr› (- $\ll_{u'/-}$ - [100,100,101] 100)
-ubs-rsh :: ‹(int, 'α) uexpr ⇒ (nat, 'α) uexpr ⇒ (nat, 'α) uexpr ⇒ (int, 'α) uexpr› (- $\gg_{s'/-}$ - [100,100,101] 100)
-ubu-rsh :: ‹(nat, 'α) uexpr ⇒ (nat, 'α) uexpr ⇒ (nat, 'α) uexpr ⇒ (nat, 'α) uexpr› (- $\gg_{u'/-}$ - [100,100,101] 100)
-ubs-not :: ‹(nat, 'α) uexpr ⇒ (int, 'α) uexpr ⇒ (int, 'α) uexpr› ($\neg_{s'/-}$ - [200, 150] 150)
-ubu-not :: ‹(nat, 'α) uexpr ⇒ (nat, 'α) uexpr ⇒ (nat, 'α) uexpr› ($\neg_{u'/-}$ - [200, 150] 150)
-ubu-neg :: ‹(nat, 'α) uexpr ⇒ (nat, 'α) uexpr ⇒ (nat, 'α) uexpr› ($-_{u'/-}$ - [200, 150] 150)

translations
-umset ⇌ *CONST uop CONST mset*
-uint ⇌ *CONST uop CONST int*
-unat ⇌ *CONST uop CONST nat*
f⦇*kf*⦈$_r$ ⇀ *CONST uop kf f*

$f(k \mapsto v)_r \rightharpoonup$ *CONST bop k* (*CONST rec-update-wrapper v*) *f*
-ubs-and ⇌ *CONST bop* (*CONST s-bitop* (*op AND*))
-ubu-and ⇌ *CONST bop* (*CONST u-bitop* (*op AND*))
-ubs-or ⇌ *CONST bop* (*CONST s-bitop* (*op OR*))
-ubu-or ⇌ *CONST bop* (*CONST u-bitop* (*op OR*))
-ubs-lsh ⇌ *CONST trop CONST s-lsh*
-ubu-lsh ⇌ *CONST trop CONST u-lsh*
-ubs-rsh ⇌ *CONST trop CONST s-rsh*
-ubu-rsh ⇌ *CONST trop CONST u-rsh*
-ubs-not ⇌ *CONST bop CONST s-not*
-ubu-not ⇌ *CONST bop CONST u-not*
-ubu-neg ⇌ *CONST bop CONST u-neg*

B.2.2 Extra stuff to work more-arg functions into UTP

lift-definition *qiop* ::
⟨$('a \Rightarrow 'b \Rightarrow 'c \Rightarrow 'd \Rightarrow 'e \Rightarrow 'f) \Rightarrow$
$('a, '\alpha)$ *uexpr* ⇒ $('b, '\alpha)$ *uexpr* ⇒ $('c, '\alpha)$ *uexpr* ⇒ $('d, '\alpha)$ *uexpr* ⇒ $('e, '\alpha)$ *uexpr* ⇒
$('f, '\alpha)$ *uexpr*⟩
is ⟨$\lambda f\ u\ v\ w\ x\ y\ b.\ f\ (u\ b)\ (v\ b)\ (w\ b)\ (x\ b)\ (y\ b)$⟩ **.**
lift-definition *sxop* ::
⟨$('a \Rightarrow 'b \Rightarrow 'c \Rightarrow 'd \Rightarrow 'e \Rightarrow 'f \Rightarrow 'g) \Rightarrow$
$('a, '\alpha)$ *uexpr* ⇒ $('b, '\alpha)$ *uexpr* ⇒ $('c, '\alpha)$ *uexpr* ⇒ $('d, '\alpha)$ *uexpr* ⇒ $('e, '\alpha)$ *uexpr* ⇒
$('f, '\alpha)$ *uexpr* ⇒ $('g, '\alpha)$ *uexpr*⟩
is ⟨$\lambda f\ u\ v\ w\ x\ y\ z\ b.\ f\ (u\ b)\ (v\ b)\ (w\ b)\ (x\ b)\ (y\ b)\ (z\ b)$⟩ **.**
lift-definition *sepop* ::
⟨$('a \Rightarrow 'b \Rightarrow 'c \Rightarrow 'd \Rightarrow 'e \Rightarrow 'f \Rightarrow 'g \Rightarrow 'h) \Rightarrow$
$('a, '\alpha)$ *uexpr* ⇒ $('b, '\alpha)$ *uexpr* ⇒ $('c, '\alpha)$ *uexpr* ⇒ $('d, '\alpha)$ *uexpr* ⇒ $('e, '\alpha)$ *uexpr* ⇒
$('f, '\alpha)$ *uexpr* ⇒ $('g, '\alpha)$ *uexpr* ⇒ $('h, '\alpha)$ *uexpr*⟩
is ⟨$\lambda f\ u\ v\ w\ x\ y\ z\ a\ b.\ f\ (u\ b)\ (v\ b)\ (w\ b)\ (x\ b)\ (y\ b)\ (z\ b)\ (a\ b)$⟩ **.**
update-uexpr-rep-eq-thms — Necessary to get the above utilized by {pred,rel}_{auto,simp}

The below lemmas do not seem useful in general but are included for completeness.

lemma *qiop-ueval* [*ueval*]: ⟨$[\![qiop\ f\ v\ x\ y\ z\ w]\!]_e\ b = f\ ([\![v]\!]_e\ b)\ ([\![x]\!]_e\ b)\ ([\![y]\!]_e\ b)\ ([\![z]\!]_e\ b)\ ([\![w]\!]_e\ b)$⟩
by *transfer simp*

lemma *subst-qiop* [*usubst*]: ⟨$\sigma \dagger qiop\ f\ t\ u\ v\ w\ x = qiop\ f\ (\sigma \dagger t)\ (\sigma \dagger u)\ (\sigma \dagger v)\ (\sigma \dagger w)\ (\sigma \dagger x)$⟩
by *transfer simp*

lemma *unrest-qiop* [*unrest*]: ⟨$[\![x \sharp t;\ x \sharp u;\ x \sharp v;\ x \sharp w;\ x \sharp y]\!] \Longrightarrow x \sharp qiop\ f\ t\ u\ v\ w\ y$⟩
by *transfer simp*

lemma *aext-qiop* [*alpha*]:
⟨$qiop\ f\ t\ u\ v\ w\ x \oplus_p a = qiop\ f\ (t \oplus_p a)\ (u \oplus_p a)\ (v \oplus_p a)\ (w \oplus_p a)\ (x \oplus_p a)$⟩
by *pred-auto*

lemma *lit-qiop-simp* [*lit-simps*]:
‹*«i x y z u t» = qiop i «x» «y» «z» «u» «t»*›
by *transfer simp*

lemma *sxop-ueval* [*ueval*]: $\langle [\![sxop\ f\ v\ x\ y\ z\ w\ t]\!]_e\ b = f\ ([\![v]\!]_e b)\ ([\![x]\!]_e b)\ ([\![y]\!]_e b)\ ([\![z]\!]_e b)\ ([\![w]\!]_e b)\ ([\![t]\!]_e b)\rangle$
by *transfer simp*

lemma *subst-sxop* [*usubst*]:
$\langle \sigma \dagger sxop\ f\ t\ u\ v\ w\ x\ y = sxop\ f\ (\sigma \dagger t)\ (\sigma \dagger u)\ (\sigma \dagger v)\ (\sigma \dagger w)\ (\sigma \dagger x)\ (\sigma \dagger y)\rangle$
by *transfer simp*

lemma *unrest-sxop* [*unrest*]: $\langle [\![x \sharp t;\ x \sharp u;\ x \sharp v;\ x \sharp w;\ x \sharp y;\ x \sharp z]\!] \Longrightarrow x \sharp sxop\ f\ t\ u\ v\ w\ y\ z\rangle$
by *transfer simp*

lemma *aext-sxop* [*alpha*]:
$\langle sxop\ f\ t\ u\ v\ w\ x\ y \oplus_p a = sxop\ f\ (t \oplus_p a)\ (u \oplus_p a)\ (v \oplus_p a)\ (w \oplus_p a)\ (x \oplus_p a)\ (y \oplus_p a)\rangle$
by *pred-auto*

lemma *lit-sxop-simp* [*lit-simps*]:
‹*«i x y z u t v» = sxop i «x» «y» «z» «u» «t» «v»*›
by *transfer simp*

end

B.3 VCG Helpers

theory *vcg-helpers*
imports *utp-extensions*
begin

lemma *disjE1*:
assumes $\langle P \vee Q\rangle$
and $\langle P \Longrightarrow R\rangle$
and $\langle \neg P \Longrightarrow Q \Longrightarrow R\rangle$
shows $\langle R\rangle$
using *assms* **by** *blast*

lemma *insert-with-sorted*:
assumes $\langle sorted\ (xs_1\ @\ xs_2)\rangle$
and $\langle \forall y \in set\ xs_2.\ x < y\rangle$
and $\langle x \geq last\ xs_1\rangle$
and $\langle xs_1 \neq []\rangle$
shows $\langle sorted\ (xs_1\ @\ x\ \#\ xs_2)\rangle$

```
  using assms
  by (auto simp: sorted-append sorted-Cons) (smt One-nat-def Suc-pred diff-Suc-less
      dual-order.trans in-set-conv-nth last-conv-nth leD length-greater-0-conv not-less-eq-eq
      sorted-nth-mono)
```

B.3.1 Swap

The below definition provides an easy-to-understand swap-elements-at-i-and-(i-1) function.

```
definition ‹swap-at i xs = xs[i := xs!(i−1), i−1 := xs!i]›
abbreviation ‹swap-at_u ≡ bop swap-at›
```

The below definition provides a more general swap function.

```
definition ‹swap i j xs = xs[i := xs!j, j := xs!i]›
abbreviation ‹swap_u ≡ trop swap›

lemma mset-swap[simp]:
  assumes ‹i < length xs›
      and ‹j < length xs›
  shows ‹mset (swap i j xs) = mset xs›
  using assms unfolding swap-def
  by (simp add: mset-swap)

lemma set-swap[simp]:
  assumes ‹i < length xs›
      and ‹j < length xs›
    shows ‹set (swap i j xs) = set xs›
  using assms unfolding swap-def
  by simp

lemma swap-commute:
  ‹swap i j xs = swap j i xs›
  unfolding swap-def
  by (cases ‹i = j›) (auto simp: list-update-swap)

lemma swap-id[simp]:
  assumes ‹i < length xs›
  shows ‹swap i i xs = xs›
  using assms unfolding swap-def
  by simp

lemma drop-swap[simp]:
  assumes ‹i < n›
      and ‹j < n›
  shows ‹drop n (swap i j xs) = drop n xs›
```

```
  using assms unfolding swap-def
  by simp

lemma take-swap[simp]:
  assumes ‹n ≤ i›
      and ‹n ≤ j›
  shows ‹take n (swap i j xs) = take n xs›
  using assms unfolding swap-def
  by simp

lemma swap-length-id[simp]:
  assumes ‹i < length xs›
      and ‹j < length xs›
  shows ‹length (swap i j xs) = length xs›
  using assms unfolding swap-def
  by simp

lemma swap-nth1[simp]:
  assumes ‹i < length xs›
      and ‹j < length xs›
  shows ‹swap i j xs ! i = xs ! j›
  using assms unfolding swap-def
  by (simp add: nth-list-update)

lemma swap-nth2[simp]:
  assumes ‹i < length xs›
      and ‹j < length xs›
  shows ‹swap i j xs ! j = xs ! i›
  using assms unfolding swap-def
  by (simp add: nth-list-update)
```

B.3.2 Slice

```
definition ‹slice l u A ≡ drop l (take u A)›
abbreviation ‹slice_u ≡ trop slice›

lemma slice-empty[simp]:
  assumes ‹i ≥ j›
  shows ‹slice i j xs = []›
  using assms unfolding slice-def
  by simp

lemma slice-nonempty[simp]:
  assumes ‹i < j›
      and ‹i < length xs›
```

```
  shows ‹slice i j xs ≠ []›
  using assms unfolding slice-def
  by simp

lemma slice-suc2-eq:
  assumes ‹j < length xs›
      and ‹i ≤ j›
    shows ‹slice i (Suc j) xs = slice i j xs @ [xs!j]›
  using assms unfolding slice-def
  by (metis diff-is-0-eq drop-0 drop-append length-take less-imp-le min.absorb2
      take-Suc-conv-app-nth)

lemma slice-update-outofbounds-upper[simp]:
  assumes ‹j ≤ k›
  shows ‹slice i j (xs[k := l]) = slice i j xs›
  using assms unfolding slice-def
  by simp

lemma slice-update2-outofbounds-lower[simp]:
  assumes ‹k < i›
  shows ‹slice i j (xs[k := l]) = slice i j xs›
  using assms unfolding slice-def
  by (simp add: drop-take)

lemma drop-set-conv-nth:
  ‹set (drop i xs) = {xs!k | k. i ≤ k ∧ k < length xs}›
  apply (induction xs rule: rev-induct)
   apply (auto simp: nth-append)
  by (metis (no-types, lifting) Suc-pred cancel-comm-monoid-add-class.diff-cancel diff-is-0-eq
      drop-Cons' drop-Nil in-set-conv-nth length-drop length-pos-if-in-set lessI less-Suc0
      less-not-refl)

lemma take-set-conv-nth:
  ‹set (take i xs) = {xs!k | k. k < min i (length xs)}›
  apply (induction i)
   apply auto
   apply (smt in-set-conv-nth le-eq-less-or-eq length-take less-Suc-eq less-le-trans min.absorb2
      not-less-eq-eq nth-take take-all)
  using in-set-conv-nth by fastforce

lemma slice-set-conv-nth:
  ‹set (slice i j xs) = {xs!k | k. i ≤ k ∧ k < j ∧ k < length xs}›
  unfolding slice-def
  by (auto simp: drop-set-conv-nth take-set-conv-nth) force
```

```
lemma slice-update-extract:
  assumes ‹lo ≤ i›
    and ‹i < hi›
  shows ‹slice lo hi (A[i := x]) = (slice lo hi A)[i−lo := x]›
  using assms unfolding slice-def
  by (simp add: drop-update-swap take-update-swap)

lemma slice-length[simp]:
  assumes ‹lo ≤ hi›
      and ‹hi ≤ length xs›
  shows ‹length (slice lo hi xs) = hi − lo›
  using assms unfolding slice-def
  by simp

lemma nth-slice-offset[simp]:
  assumes ‹i < hi − lo›
      and ‹lo ≤ hi›
      and ‹hi ≤ length xs›
  shows ‹(slice lo hi xs)!i = xs!(i + lo)›
  using assms unfolding slice-def
  by (simp add: add.commute min.absorb2)

lemma slice-merge[simp]:
  assumes ‹lo ≤ i›
      and ‹i ≤ hi›
      and ‹hi < length xs›
  shows ‹slice lo i xs @ slice i hi xs = slice lo hi xs›
  using assms unfolding slice-def
  by (smt append-take-drop-id diff-is-0-eq drop-0 drop-append length-take less-or-eq-imp-le
      min.absorb2 take-take)

lemma element-in-set-of-slice: — May not be useful
  assumes ‹lo ≤ i›
      and ‹i < hi›
      and ‹i < length xs›
  shows ‹xs!i ∈ set (slice lo hi xs)›
  using assms
  by (auto simp: slice-set-conv-nth)

lemma take-slice[simp]: ‹take n (slice lo hi xs) = slice lo (min (n + lo) hi) xs›
  unfolding slice-def by (simp add: take-drop)

lemma drop-slice[simp]:
  ‹drop n (slice lo hi xs) = slice (n + lo) hi xs›
  unfolding slice-def by simp
```

B.3.3 Swap and Slice together

lemma *slice-swap-extract*:
  assumes ‹$lo \leq i$›
    and ‹$lo \leq j$›
    and ‹$i < hi$›
    and ‹$j < hi$›
    and ‹$i < \mathit{length}\ xs \vee j < \mathit{length}\ xs$› — Not sure why it doesn't need both at once.
  shows ‹$\mathit{slice}\ lo\ hi\ (\mathit{swap}\ i\ j\ xs) = \mathit{swap}\ (i - lo)\ (j - lo)\ (\mathit{slice}\ lo\ hi\ xs)$›
  using *assms* **unfolding** *slice-def swap-def*
  by (*smt append-take-drop-id drop-update-swap le-cases length-take min.absorb2 not-less nth-append order-trans take-all take-update-swap*)

lemma *mset-slice-swap*[*simp*]:
  assumes ‹$lo \leq i$›
    and ‹$lo \leq j$›
    and ‹$i < hi$›
    and ‹$j < hi$›
    and ‹$i < \mathit{length}\ xs$›
    and ‹$j < \mathit{length}\ xs$›
  shows ‹$\mathit{mset}\ (\mathit{slice}\ lo\ hi\ (\mathit{swap}\ i\ j\ xs)) = \mathit{mset}\ (\mathit{slice}\ lo\ hi\ xs)$›
  using *assms*
  apply (*simp add*: *slice-swap-extract*)
  unfolding *slice-def*
  by *simp*

lemma *set-slice-swap*[*simp*]:
  assumes ‹$lo \leq i$›
    and ‹$lo \leq j$›
    and ‹$i < hi$›
    and ‹$j < hi$›
    and ‹$i < \mathit{length}\ xs$›
    and ‹$j < \mathit{length}\ xs$›
  shows ‹$\mathit{set}\ (\mathit{slice}\ lo\ hi\ (\mathit{swap}\ i\ j\ xs)) = \mathit{set}\ (\mathit{slice}\ lo\ hi\ xs)$›
  using *assms*
  apply (*simp add*: *slice-swap-extract*)
  unfolding *slice-def*
  by *simp*

lemma *set-slice-swap-greaterthan*: — Not all that useful. Could be more general.
  fixes *xs* :: ‹-::*linorder list*›
  assumes ‹$\forall x \in \mathit{set}\ (\mathit{slice}\ i\ hi\ xs).\ x \geq xs!hi$›
    and ‹$i \leq hi$›
    and ‹$hi < \mathit{length}\ xs$›

shows $\langle \forall x \in \mathit{set}\ (\mathit{slice}\ i\ (\mathit{Suc}\ \mathit{hi})\ (\mathit{swap}\ i\ \mathit{hi}\ \mathit{xs})).\ x \geq \mathit{xs}!\mathit{hi}\rangle$
using *assms*
apply *auto* — First application cannot use $[\![?j < \#_u(?\mathit{xs});\ ?i \leq ?j]\!] \Longrightarrow \mathit{slice}\ ?i\ (\mathit{Suc}\ ?j)\ ?\mathit{xs} = \mathit{slice}\ ?i\ ?j\ ?\mathit{xs}\ @\ [?\mathit{xs}(?j)_a]$ or we get an extra goal.
by (*auto simp*: *slice-suc2-eq*)

lemma *slice-swap1* [*simp*]:
assumes $\langle i < \mathit{lo}\rangle$
and $\langle j < \mathit{lo}\rangle$
shows $\langle \mathit{slice}\ \mathit{lo}\ \mathit{hi}\ (\mathit{swap}\ i\ j\ \mathit{xs}) = \mathit{slice}\ \mathit{lo}\ \mathit{hi}\ \mathit{xs}\rangle$
using *assms* **unfolding** *slice-def swap-def*
by (*simp add*: *drop-take*)

lemma *slice-swap2* [*simp*]:
assumes $\langle \mathit{hi} \leq i\rangle$
and $\langle \mathit{hi} \leq j\rangle$
shows $\langle \mathit{slice}\ \mathit{lo}\ \mathit{hi}\ (\mathit{swap}\ i\ j\ \mathit{xs}) = \mathit{slice}\ \mathit{lo}\ \mathit{hi}\ \mathit{xs}\rangle$
using *assms* **unfolding** *slice-def swap-def*
by *simp*

B.3.4 Sorting pivots

definition $\langle \mathit{pivot\text{-}invr}\ i\ A \equiv \forall x \in \mathit{set}\ (\mathit{take}\ i\ A).\ \forall y \in \mathit{set}\ (\mathit{drop}\ i\ A).\ x \leq y\rangle$
abbreviation $\langle \mathit{pivot\text{-}invr}_u \equiv \mathit{bop}\ \mathit{pivot\text{-}invr}\rangle$

B.3.5 Miscellaneous

lemma *upred-taut-refl*: $\langle `A \Rightarrow A`\rangle$
by *pred-simp*

Minor helper for blocks in partial correctness (currently unused.)

abbreviation $\langle \mathit{stet}\ v\ s \equiv «[\![v]\!]_e\ s»\rangle$

end