# A Constructive Formalisation of Hoare Logic within the Interactive Theorem Prover Agda

Project Report
Word Count XXXX
Fraser L. Brooks 1680975
Supervisor: Vincent Rahli

Submitted in partial fulfillment
of the requirements for the degree of
Master of Science
(Computer Science)

at the
University of Birmingham
School of Computer Science
July 2021

## Abstract

Program correctness is a perennial problem for software engineers and computer scientists alike. Many methods exist for establishing the correctness of a program and broadly speaking these methods fall into one of two paradigms; a program can be tested for correctness or the correctness can be 'proved' outright. Due to the sheer complexity of software engineering, testing has reigned supreme in industry as formal techniques for proving correctness, while numerous, have lagged behind practice. However, with the advent of higher-order-logic theorem provers and dependently typed programming languages, both operating under the scope of the Curry-Howard correspondence, the gap between practice and theory is shrinking.

Hoare logic is a formal system in which one can reason rigorously about — and *prove* — the correctness of programs while Agda is both a dependently typed programming language *and* an interactive theorem prover in accordance with the Curry-Howard correspondence. Combining the two, this work sets out to formalise the salient rules from Hoare logic within Agda and in doing so, provide a novel library with which a user could reason and prove correct simple imperative-style programs.

This formalisation was achieved via a deep embedding of both a simple imperative language, dubbed *'Mini-Imp,'* and of a propositional calculus used in the reasoning about programs in the guise of Mini-Imp's expression language. Agda record interfaces were also used to seperate out the concerns of proving program correctness and proving trivial results within the expression language — such as conjunction elimination or the distributivity of multiplication over addition.

The final result is an Agda library that is fit for the purpose of reasoning about and proving correct simple imperative-style programs using the implemented Hoare logic rules. A limitation of the work is the simplicity of the Mini-Imp language and corresponding lack of more sophisticated logical rules meaning there is no facility for reasoning about more complex language constructs like procedures, arrays or pointers. However, more powerful logics such as 'separation logic' could bridge this gap and owing to the expressive power of HOL, with time, there is no reason why the current library couldn't be expanded to encompass separation logic too.

# Contents

# 1 Preliminaries & Literature Review

## 1.1 Programming Language Semantics

"Programming began as an art"

- David Gries, *The Science of Programming*

Around the late 60's – early 70's, in response to the verbose and inelegant languages of the time — some of which, sadly, are still in use today — computer scientists were experimenting with different ways of giving semantics to programming languages. The desire being partly to assist in the development of more elegent laguages but also partly to get a better mathematical grip on the process of computer programming and, in doing so, make a science out of the art.

Early approaches to language specification and semantics fell into what would become the category of *operational* semantics[1] that describe a language in terms of how it is to be executed. Thus demonstrating that the most salient interpretation of a program at the time was as a set of instructions destined for execution by a machine[2] rather than as a syntactic representation of the mathematical object known as an *algorithm*. This lead to languages being designed with the machines that would run them, and the programmers that would use them, in mind. This led to a state of affairs wherein reasoning about the correctness of programms was much harder than it needed to be and was seen as not worth the effort.

As Dijkstra remarked at the time, the balance needed 'redressing' thus leading to a couple of seminal papers, first by Hoare[7] and then himself[3]—the latter, in part, inspired by the former. Hoare introduced *axiomatic* semantics as a means to understanding computer programs via the assertions that can be said to be true before and after execution[3]. Then Dijkstra introduced a *predicate transformer* semantics identifying language constructs with functions between preconditions and postconditions — and thus, the balance between practical power and mathematical elegence and rigour began to find more equitable ground.

[1]In 1968 an operational semantics was given for Algol 68. Even earlier, in 1960, the lambda calculus — the semantics of which is commonly understood as operational — was evoked in giving semantics to the Lisp programming language.

[2]And this meant the *specific* and often competing machines of the time.

[3]Regardless of how that execution is performed!

### 1.1.1 Axiomatic Semantics via Hoare Logic

In 1967, as an alternative to operational semantics, Floyd[5] produced his seminal paper 'Assigning Meanings to Programs' in which a program is given semantics via attatchment of propositions to the connections in a flow chart with nodes as commands. In Floyd's deductive system, whenever a command (a node) is reached via a connection whose associated proposition is true, then, if execution of the program leaves that node, it will leave through a connection whose associated proposition is also true.

> "A *semantic definition* of a particular set of command [program] types, then, is a rule for constructing, for any command $\mathcal{c}$ of one of these types, a *verification condition* $\mathcal{V}_c(\mathcal{P}; \mathcal{Q})$ on the antecedents and consequents of $\mathcal{c}$". - Floyd[5]

The principle idea is that rather than define a program (however large or small) by the way it should be executed, a program can be defined by the antecedents upon the state space that must be true before execution — hereafter referred to as *preconditions* — and the associated consequents upon the state space that can be guaranteed to be true after execution — hereafter referred to as *postconditions* — thus freeing the semantics from concerns of the *how* in favour of the *what.*

These 'antecedents/consequents upon the state space' are first-order logic predicates or propositions and the state space is taken most generally to be a set of pairs of identifiers and values; again the formulation here shields us from implementation details such as whether these 'identifiers' identify memory addresses within a machine or Post-it Notes on a wall.

Later then, in 1969, Hoare[7] built upon and expanded Floyd's work[4], applying the system to text rather than to flow charts, creating a *deductive system* for reasoning about the correctness of programs as we would more naturally recognise them. Central to Hoare's system is the notion of a *Hoare triple* which is a reformulation of Floyd's verification condition $\mathcal{V}_c(\mathcal{P}; \mathcal{Q})$. A Hoare triple associates a precondition, or a state, before execution of a particular program with a resultant postcondition, or state, after execution.[5]

[4]Thus Hoare logic is sometimes referred to as Floyd-Hoare Logic

[5]NB Here, as in much of the literature, preconditions and postconditions and the actual subsets of the state space that they describe are used interchangeably. i.e. $\textsc{False} = \emptyset$ and $\textsc{True} = \mathcal{S}$ where $\mathcal{S}$ is the whole state space.

A Hoare triple is of the form '$\{\!\{\ \mathcal{P}\ \}\!\}\ \mathcal{Q}\ \{\!\{\ \mathcal{R}\ \}\!\}$' which can be read as ...

- If the notation is to denote *partial correctness*:
  - If execution of the program $\mathcal{Q}$ begins in a state satisfying $\mathcal{P}$: then $\mathcal{R}$ will be true of the resultant state *so long as* $\mathcal{Q}$ terminates.
- If the notation is to denote *total correctness*:
  - As above but termination of $\mathcal{Q}$ is also guaranteed.

A note on notation: Hoare's original notation was $\mathcal{P}\ \{\!\{\ \mathcal{Q}\ \}\!\}\ \mathcal{R}$ to denote *partial correctness* but the notation above is now more common. Confusingly, some use $\{\!\{\ \mathcal{P}\ \}\!\}\ \mathcal{Q}\ \{\!\{\ \mathcal{R}\ \}\!\}$ to denote *total* correctness and the other form for partial correctness. In general there seems to be no standard notation, with the $\{\!\{\ \mathcal{P}\ \}\!\}\ \mathcal{Q}\ \{\!\{\ \mathcal{R}\ \}\!\}$ form oft used for the form of correctness most salient for a given work; as such, in this report, the $\{\!\{\ \mathcal{P}\ \}\!\}\ \mathcal{Q}\ \{\!\{\ \mathcal{R}\ \}\!\}$ denotes partial correctness.

The utility of the Hoare triple notation is then immediately demonstrated by giving the Hoare triple that characterises the statement/command that assigns a value to a variable:

Given the expression $f$ and assignment statement '$x := f$':

$$\{\!\{\ \mathcal{P}_0\ \}\!\}\ x := f\ \{\!\{\ \mathcal{P}\ \}\!\}$$

... where $\mathcal{P}_0$ is formed by substituting $f$ for $x$ in $\mathcal{P}$ ($\mathcal{P}_0 = \mathcal{P}[f/x]$).

Note that in general $\{\!\{\ \mathcal{P}\ \}\!\}\ \mathcal{Q}\ \{\!\{\ \mathcal{R}\ \}\!\}$ is a predicate within the predicate calculus (see 5) that can either be true or false, depending on the arguments supplied. The triple given above, however, is actually the first and only *axiom*[6] in Hoare's system as it is true for all possible $\mathcal{P}$, $f$, and $x$. [7]

$$\text{D0 - Axiom of Assignment: } \vdash \{\!\{\ \mathcal{P}[f/x]\ \}\!\}\ x := f\ \{\!\{\ \mathcal{P}\ \}\!\} \tag{1}$$

[6]In actuality, it is an axiom *schema* describing an infinite set of axioms all sharing a common form.

[7]A fact that is proved constructively (along with the inference rules 2, 3, and 4, described on page 7) as part of this formalisation.

This first example not only demonstrates the utility and elegence of the Hoare triple but also shines a light on two of the stumbling blocks of Hoare logic. The first of these is the substitution of the programming language expression $f$ into the predicate $\mathcal{P}$ thus indicating an interplay between the expression language and the assertion language — the assertion language, being, the language from which preconditions and postconditions are to be formed. In theory, and in practice, this interplay isn't a problem so long as the assertion language is more expressive than the program's expression language. So long as this condition is met there will always be *some* sensible, well-defined way of substituting an expression into an assertion and because there is never a need to substitute in the opposite direction, no further complications arise.[8]

The second stumbling block is that at first, to many, the reasoning appears to be happening in the wrong direction. From starting condition $\mathcal{P}$, we substitute to get $\mathcal{P}_0$, that is, we move from postcondition to precondition when to many programmers, reasoning in the direction of execution feels much more natural. The axioms that match the standard programmer's intuition are:

$$\begin{array}{ll} (1.) & \vdash \{\!\!\{\ \mathcal{P}\ \}\!\!\}\ \chi := f\ \{\!\!\{\ \mathcal{P}[\chi/f]\ \}\!\!\} \\ \text{or} \ldots & \\ (2.) & \vdash \{\!\!\{\ \mathcal{P}\ \}\!\!\}\ \chi := f\ \{\!\!\{\ \mathcal{P}[f/\chi]\ \}\!\!\} \end{array}$$

...but both of these are erroneous. The first gives the false consequent $\vdash \{\!\!\{\ \chi = 1\ \}\!\!\}\ \chi := 0\ \{\!\!\{\ \chi = 1\ \}\!\!\}$ as a direct consequence of the fact that $(\chi = 1)[0/(x)] = (\chi = 1)$; as 0 doesn't occur in '$\chi = 1$'. The second gives the false consequent of $\vdash \{\!\!\{\ \chi = y\ \}\!\!\}\ \chi := z\ \{\!\!\{\ z = y\ \}\!\!\}$ via substituting $\chi$ for $z$.

So in fact, the reasoning is in the right direction, that is, *backwards*. This is in line with the radical reformulation of programming that was being proposed at the time by Hoare, and later Dijkstra, and then most lucidly expatiated upon in Gries' monograph[6], 'The Science of Programming.' This reformulation was to frame programming as a *goal-oriented* activity and to construct programs alongside a proof of correctness, starting with the desired postcondition — the desired output — and working backwards towards the necessary precondtion/input.[9]

[8]Within this formalisation however, this stumbling block does present a challenge as it forces upon us a number of considerations. See subsection 2.3

[9]As a result, proofs of correctness constructed using the Agda library produced by this project are also constructed backwards. See XXXXXXX

So we've seen the characterisation of the assignment command as an axiom. In Hoare's original paper, the following inference rules were aslo given, from which proofs of correctness could be developed, starting with the fairly intuitive rules of consequence:

$$\text{D1 - Rules of Consequence:} \tag{2}$$

$$\begin{array}{lllllll} \text{If} & \vdash \{\!\!\{\ \mathcal{P}\ \}\!\!\}\ \mathcal{Q}\ \{\!\!\{\ \mathcal{R}\ \}\!\!\} & \text{and} & \vdash \mathcal{R} \Rightarrow \mathcal{S} & \text{then} & \vdash \{\!\!\{\ \mathcal{P}\ \}\!\!\}\ \mathcal{Q}\ \{\!\!\{\ \mathcal{S}\ \}\!\!\} \\ \text{If} & \vdash \{\!\!\{\ \mathcal{P}\ \}\!\!\}\ \mathcal{Q}\ \{\!\!\{\ \mathcal{R}\ \}\!\!\} & \text{and} & \vdash \mathcal{S} \Rightarrow \mathcal{P} & \text{then} & \vdash \{\!\!\{\ \mathcal{S}\ \}\!\!\}\ \mathcal{Q}\ \{\!\!\{\ \mathcal{R}\ \}\!\!\} \end{array}$$

Next up is the rule of composition which is the rule that allows us to chain Hoare triples together to build up larger proofs of correctness for programs from the proofs of correctness of these programs' constituent parts.

$$\text{D2 - Rule of Composition:} \tag{3}$$

$$\text{If } \vdash \{\!\!\{\ \mathcal{P}\ \}\!\!\}\ \mathcal{Q}_1\ \{\!\!\{\ \mathcal{R}_1\ \}\!\!\} \quad \text{and } \vdash \{\!\!\{\ \mathcal{P}\ \}\!\!\}\ \mathcal{Q}_1\ \{\!\!\{\ \mathcal{R}_2\ \}\!\!\} \text{ then } \vdash \{\!\!\{\ \mathcal{P}\ \}\!\!\}\ \mathcal{Q}_1\ ; \mathcal{Q}_2\ \{\!\!\{\ \mathcal{R}\ \}\!\!\}$$

Finally the most interesting rule, the rule of iteration:

$$\text{D3 - Rule of Iteration:} \tag{4}$$

$$\text{If } \vdash \{\!\!\{\ \mathcal{P} \wedge \mathcal{B}\ \}\!\!\}\ \mathcal{S}\ \{\!\!\{\ \mathcal{P}\ \}\!\!\} \text{ then } \vdash \{\!\!\{\ \mathcal{P}\ \}\!\!\}\ \texttt{WHILE}\ \mathcal{B}\ \texttt{DO}\ \mathcal{S}\ \{\!\!\{\ \neg\mathcal{B} \wedge \mathcal{P}\ \}\!\!\}$$

The insights on display here are that *if* a loop terminates, then we can be sure that the condition $\mathcal{B}$ of the loop is now false, and that if we have a condition $\mathcal{P}$ that we know isn't changed by the running of the body of the loop so long as it is ran when the loop condition $\mathcal{B}$ is also true ($\vdash \{\!\!\{\ \mathcal{P} \wedge \mathcal{B}\ \}\!\!\}\ \mathcal{S}\ \{\!\!\{\ \mathcal{P}\ \}\!\!\}$), then we can also be sure that $\mathcal{P}$ is true *after* the loop terminates. And thus the, now well known, notion of a a loop *invariant* has been introduced.

This was one of the first contributions of the *theory* of programming to the practice, viz, that when designing a loop, we should start with the desired postcondition $\mathcal{R}$ and search for a $\mathcal{P}$ and $\mathcal{B}$ fitting the schema above — i.e. A $\mathcal{P}$ and $\mathcal{B}$ such that $\mathcal{P} \wedge \neg\mathcal{B} \Rightarrow \mathcal{R}$ — at which point we'll have the condition of the loop *and* its precondition and all that shall be left to do is fill in the body of the loop; which we'll be able to do safely by making sure that $\mathcal{P}$ is left invariant, and execution moves towards the falsity of $\mathcal{B}$.

### 1.1.2 Predicate Transformer Semantics via Dijkstra's *Weakest Precondition*

"Program testing can be used to show the presence of bugs, but never to show their absence!"

- Edsger W. Dijkstra, 1970

Very early on in the field of computing science, Dijkstra, among others, was also interested in putting programming on surer mathematical footing; moving away from the notion of programming as a 'chaotic contribution' of 'thousands of ingenious tricks' as it was put in his talk 'Some meditations on Advanced Programming'[2], at the 1962 IFIP conference.

Despite some disagreements between industy and academics — or rather, the theoretically inclined academics — as the 60's progressed and the programs being developed grew in scope, it became apparent that Dijkstra and similar detractors of the status quo were right and that there were serious problems with programming. Hoare's paper had spawned a field of research on axiomatic definitions of programming languages, and many papers were born from this, but the utility of the approach was still subject to doubt. Axiomatic definitions provided a way to reason about programs but didn't so much present a way to *develop* them.

Then, in 1975, building upon Hoare's paper, Dijkstra carried the notion of an axiomatic semantics further in his very influential paper 'Guarded Commands and Non-Determinism'[3] — followed up by the monograph 'A Discipline of Programming'[4] — in which he introduced the notion of a *predicate transformer semantics.*[10]

Whereas Hoare logic characterises programming constructs in terms of logical assertions upon the state space, the idea behind predicate transformer semantics is to characterise a programming construct in terms of a 'predicate transformer', that is, a function that transforms one predicate into another.[11] Thus was born Dijkstra's, now well known, *Weakest Precondition*; usually denoted $wp(\mathcal{S}, \mathcal{R})$ for a command $\mathcal{S}$ and postcondition $\mathcal{R}$.

[10]As well as introducing the notion of guarded commands still very much present in languages today

[11]Being a *function*, that makes predicate transformer semantics a form of *Denotational Semantics*; that is, semantics defined via reference to mathematical objects.

It's useful to keep the two views in mind, that the weakest precondition is both a predicate but also a means of *characterising* a particular programming construct. It is defined for a command $\mathcal{S}$ and a predicate $\mathcal{R}$ that describes the desired result of executing command $\mathcal{S}$ — that is, $\mathcal{R}$ is the desired post-condition — as the predicate, $wp(\mathcal{S}, \mathcal{R})$, that represents/captures:

> "the set of *all* states such that execution of $\mathcal{S}$ begun in any one of them is guaranteed to terminate in a finite amount of time in a state satisfying $\mathcal{R}$." - Gries[6]

NB The term 'weakest' here, means the least restrictive predicate, or, if we consider assertions as the subset of the state space they describe, then 'weakest' can be taken to mean the predicate with the highest cardinality. The guarantee of termination means we're reasoning about *total* correctness. There is a closely associated notion of a weakest *liberal* precondition, defined identically, but without the guarantee of termination,[12]and denoted $wlp(\mathcal{S}, \mathcal{R})$.

The relation to Hoare logic should now be clear and indeed, $wp(\mathcal{S}, \mathcal{R})$ can be defined in terms of Hoare logic: we can say that for a particular command $\mathcal{S}$, and a postcondition $\mathcal{R}$, such that $\mathcal{R}$ is the desired result of executing $\mathcal{S}$, then the weakest precondition is a predicate $wp(\mathcal{S}, \mathcal{R})$, such that for any precondition $\mathcal{P}$, we have $\{\!\{ \mathcal{P} \}\!\}\ \mathcal{S}\ \{\!\{ \mathcal{R} \}\!\}$ if and only if $\mathcal{P} \Rightarrow wp(\mathcal{S}, \mathcal{R})$.

$$\begin{array}{rcl} \mathcal{P}\ \{\!\{ \mathcal{S} \}\!\}\ \mathcal{R} & = & \mathcal{P} \Rightarrow wp(\mathcal{S}, \mathcal{R}) \\ \{\!\{ \mathcal{P} \}\!\}\ \mathcal{S}\ \{\!\{ \mathcal{R} \}\!\} & = & \mathcal{P} \Rightarrow wlp(\mathcal{S}, \mathcal{R}) \end{array} \tag{5}$$

This shows, as remarked previously, that a Hoare triple is just a statement within the underlying predicate calculus and proving a Hoare triple — i.e. a *program* — correct, is reduced to the task of proving a *first-order formula*!

The contribution of predicate transformer semantics, as a reformulation of Hoare logic, is that it lay the ground work for a new (at the time) paradigm of programming, a *science of programming*[6], such that for the first time, programmers could use theory to develop programs *alongside* a proof of correctness, rather than resorting to 'ingenious' but 'chaotic' tricks.

[12]And as such, is the more relevant predicate transformer for this work as constructive or formal proofs of termination is a field unto itself that is not treated in this formalisation.

As a slight diversion then, what does defining a language in terms of $wp$ look like? The simplest commands that can be characterised by their weakest preconditions are the $\mathtt{skip}$ command, that does nothing, characterised by '$wp(\mathtt{skip}, \mathcal{R}) = \mathcal{R}$', and the $\mathtt{abort}$ command — that aborts computation and signifies failure, characterised by '$wp(\mathtt{abort}, \mathcal{R}) = \mathtt{False}$.'

A more interesting example that should be familiar is the assignment command as a reformulation of the axiom of assignment from Hoare logic:

$$wp(x := f, \mathcal{R}) \quad = \quad \mathcal{R}[f/x] \tag{6}$$

A, more interesting still, example would be a characterisation of a simple 'IF...THEN...ELSE' command:

$$\begin{aligned} wp(\texttt{IF } \mathcal{B} \texttt{ THEN } \mathcal{S}_1 \texttt{ ELSE } \mathcal{S}_2 \ , \mathcal{R}) \quad = & \\ & \mathcal{B} \Rightarrow wp(\mathcal{S}_1, \mathcal{R}) \\ \wedge \quad & \neg\mathcal{B} \Rightarrow wp(\mathcal{S}_2, \mathcal{R}) \end{aligned} \tag{7}$$

A weakest precondition for a WHILE/iteration command becomes a little more involved as it has to be defined inductively, similar to the above definition, only in a way that guarantees progress torwards termination. As such, it is not repeated here; the definitions above have been given only for pedagogical reasons to situate the Hoare logic calculus and the rules formalised in this work fully within their understood context.[13]

Thankfully for Hoare logic and this formalisation, it is rarely necessary to formulate/compute the weakest precondition itself. In so far as our concerns are to prove the correctness of programs, it is enough to show that for a given precondition $\mathcal{P}$: $\mathcal{P} \Rightarrow wp(\mathcal{S}, \mathcal{R})$. Indeed, for the Hoare logic inference rules 2,3, and 4, given in the previous section, proofs that they do actually imply what they claim are given in [4]/[6]. For instance, see theorem (11.6) in [6], or the proof of the 'Fundamental Invariance Theorem' in [4] for a proof that any $\mathcal{P}$ that satisfies a more general, non-deterministic, version of the Rule of Iteration from the previous section, does in fact imply the weakest precondition of the WHILE/iterative command as given in those same works.

[13]NB That the definitions given here differ significantly from those in [3]/[4] wherein the language that is defined is non-deterministic, a fact that to many a programmer might sound alarming but in actuality makes for a much 'cleaner' language.

## 1.2 Agda as an Interactive Theorem Prover

"Beware of bugs in the above code; I have only proved it correct, not tried it."

- Donald Knuth, 1977

With Hoare logic and programming language semantics covered, the other prerequisite to understanding this report's title is to briefly explain the phrases 'Constructive Formalisation' and 'Interactive Theorem Prover.'

### 1.2.1 Formal Proof

First up, the word 'formalisation', as in, a *formal* proof. What is a formal proof? Well, according to *Merriam-Webster's* online dictionary, a proof is:

> "the cogency of evidence that compels acceptance by the mind of a truth or a fact"

What exactly this 'evidence' should be is left unspecified. In a *formal* proof, this evidence is situated within some logical system. It is a string of symbols or sentences that form a *well-formed formula* within a formally defined language — read, a language that has been described by precise and unabmiguous rules — each of which has a precise and unambiguous meaning and is either an *axiom* within the logical system, an *assumption*, or follows from one of the logical system's inference rules. Put very simply then, a formal proof is just a very assiduous, unambiguous, sometimes tedious, proof.

### 1.2.2 Constructive Mathematics

Constructive mathematics, or constructive logic, refers to mathematical or logical reasoning within the *constructivism* philosophy of mathematics. It is often characterised as classical mathematics or logic, only without the *Law of Excluded Middle* and the *Axiom of Choice*.[14] The law of the excluded middle, sometimes called the *principle* or *aiom* of the excluded middle by constructivists to emphasise the optionality, is the axiom stating that every

[14]Necessarily without the Axiom of Choice as the Axiom of Choice implies the Law of Excluded Middle within a constructive setting.

proposition is either true or false; that is, $\forall \mathcal{P}. \mathcal{P} \vee \neg \mathcal{P}$. At first glance it seems an obvious, even banal, tool to allow oneself; indeed it is a very useful principle in logic upon which many famous proofs rely. So why reject it?

The beginnings of the constructivist philosophy can be traced back to early 20th century thought led by Brouwer. The main concern of constructivism is in how one asserts that a mathematical object does or does not exist. The problem with LEM is that it allows one to assert the existence of mathematical objects without actually specifying *what* they are, that is, without *constructing* them. Consider the following classical proof:

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

Theorem: There are irrational numbers $a$ and $b$ such that $a^b$ is rational.

```
Proof:
Let $c = \sqrt{2}^{\sqrt{2}}$ and let $P(x) =$ "$x$ is rational".
Via LEM, either $P(c)$ or $\neg P(c)$.
If $P(c)$:
            let $a = b = \sqrt{2}$,
            then we have $a^b = c$,
            therefore $P(a^b)$ via $P(c)$.
If $\neg P(c)$:
            let $a = c = \sqrt{2}^{\sqrt{2}}$,
            let $b = \sqrt{2}$,
            then we have:
            $a^b = \left(\sqrt{2}^{\sqrt{2}}\right)^{\sqrt{2}} = \sqrt{2}^{\sqrt{2}\sqrt{2}} = \sqrt{2}^2 = 2$
            therefore $P(a^b)$ via $P(2)$.
So the theorem holds for both $P(c)$ and $\neg P(c)$, and so...    QED■
```

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

It's not that constructivists doubt the validity of this proof. The objection is that the object under question — some irrational numbers $a$ and $b$ such that $a^b$ is rational — hasn't actually been given. We don't know which of the two cases, $P(c)$ or $\neg P(c)$, is true.

So, rather than characterise constructive mathematics as classical mathematics without the law of the excluded middle, it perhaps can be better described positively as an approach to mathematics with stricter provability requirements wherein a thing can be said to exist only *after* constructing it.

With all that said, it is not necessary to get caught up in the philosophical arguments or the nuances of constructivism for this present work. The primary motivation here is that often constructive treatments of classical results can often prove stronger and more illuminating.

Also of our concern, is that with the rise of the computer, constructive mathematics has come into its own. A constructive proof can be read as an *algorithmic* proof — read, checkable or usable by a computer. As an example, consider the proof given just now, if we wished to do something algoritmically with the $a^b$-object that was proved to exist, the proof given there wouldn't be much use to the machine or the programmer; a computer cannot procede in two minds at once! Thus the utility of constructive mathematics has never been clearer and this leads to a contribution that computer scientists have given the world of mathematics; the subject of the next section.

### 1.2.3 Interactive Theorem Provers

Interactive Theorem Provers, or proof assistants, are broadly characterised as software systems used as an aid in the development of proofs. They can all perhaps trace their origins to a paper by Knuth and Bendix[9] from 1970 in which an algorithm was devised which was capable of deducing new laws or theorems from some given ones; some laws of elementary group theory serving as an exmple within the paper. The formal development of this algorithm in the authors words was 'primarily a precise statement of what hundreds of mathematicians have been doing for many decades.'

Of course, the space of interactive theorem provers has come a long way since then and there are now a plethora of such software systems to choose from. The principle idea behind all of them is that mathematical objects can be represented as *data*, sometimes, also referred to as a *word*, inside a machine – imagine a 'String' in your favourite programming language — and mathematical operations or laws can be implemented as operations upon that data/word. These operations might be referred to as *reductions*. For example, we can encode in data, or as *words*, the natural numbers à la Peano as follows:

`A natural number is inductively defined as either:`

1. The word/data: "zero"

2. Or the word/data: "suc $\chi$" where $\chi$ is some other natural number.

Addition over natural numbers can then be quite simply defined on a case by case basis of the inputs:

$$\texttt{x + y} \overset{\text{def}}{=} \begin{cases} 1.\ \texttt{x} = 0 : & \text{“zero + y”} = \text{“y”} \\ 2.\ \texttt{x} \neq 0 : & \text{“suc } \chi + \text{y”} = \text{“suc (} \chi + \text{y)”} \end{cases}$$

So we might say that the *word* “suc $\chi$ + y” *reduces* to the *word* “suc ($\chi$ + y)”. With such a definition, it can be *algorithmically* verified that $1 + 1 \equiv 2$, with $\equiv$ serving as *definitional* equality, as once both terms on either side of the operator are maximally reduced, they *are* the same. The two constructions above could be built upon further, for instance, the commutativity of addition could be proved, multiplication could be defined in terms of the definition of addition etc. In fact, there are projects operating today aiming to formalise large quantities of mathematics in this manner within particular interactive theorem provers. For instance, the UniMath project is a huge library of mathematics formalised within the Coq interactive theorem prover.

Within these theorem provers then, proving that $\mathcal{P} \Rightarrow \mathcal{Q}$, amounts to a search for a sequence of reductions/rules that transform the data encoding $\mathcal{P}$, into some data encoding $\mathcal{Q}$. This search is done through some degree of cooperation between the user and the machine, in some cases happening automatically, and in others requiring human intervention and ingenuity. The algorithmic nature of theorem provers following this pattern is what gives this realm of mathematics a great synergy with constructive mathematics, although it should be noted that not all interactive theorem provers have to operate under the scope of constructive mathematics.

With that said, Agda[12] is the interactive theorem prover used within this formalisation, chosen as it happens to be a constructive system by default which means that the principle of the excluded middle, if desired, would need to be *postulated* as an axiom.[15] This amounts to inserting it as a rule by which we can reduce words but at the cost of our proof having a computational meaning, because, as was mentioned in the previous section, PEM has no compuational meaning. So this restriction is what allows Aga to also claim itself a *programming language* and operate under the Curry-Howard correspondence, also referred to as the *propositions as types* paradigm in which the type signature of a program becomes a proposition and an implementation of that program becomes a proof of that proposition.

[15]A temptation that has been resisted in this formalisation.

## 1.3 Modern Literature Review

With most of the preliminaries out of the way, and most of them being historic, it is worth examining the picture of program correctness as it appears today. The field has come a long way and despite the methodolgy of program development proposed by Hoare, Dijkstra, and Gries, wherein a program is developed alongside its proof of correctness, struggling to catch on in the mainstream as programs of the time continued to swell in complexity, Hoare logic has been expanded upon giving rise most notably to *Separation Logic* which has become another success story of the theoretical making its way into industry in the form of the *Infer* tool which is now in use in a plethora of tech companies.

Separation logic originated from some papers [XX][XX] that extended Hoare logic to also facilitate reasoning about memory and pointers thus allowing one to prove correct much more complicated and sophisticated programs rather than being limited to local variables as you are in Hoare logic. Indeed, the original aims of this work were to formalise not only Hoare logic but separation logic as well but this proved too much work for the time allotted.

Also of note is the work from a group of researchers towards a *Verified Software Toolchain*[1] aiming to have a modular tool or toolchain that can statically analyse and make observations about a source-language and produce *machine-checked* proofs that guarantee the complete correctness not only of the source-language but also of the compiled program operating within a particular operating system.

Of note is that the two systems above, Infer and VST, work via two possible relaxations of the foundational Halting Problem which of course implies that we can never expect to have a program that for *all possible* programs catches *all possible* bugs. The obvious workaround to this constraint is to relax one of those two constraints, with Infer opting to allow some false negatives — and not catch all bugs — while VST diminshes the first constraint by constraining the programmer from constructing all possible programs.

TODO: THIS PAGE NEEDS MORE WORK

# 2 Specification of the Formalisation

When formalising within a symbolic system, a lot of details that are normally swept under the rug in typical expositions need to be considered. With the preliminaries out the way then, this section details the design decisions that were made with regards to these unavoidable details, the justifications for those decisions, and finally the scope of, and overarching plan for, the formalisation at hand.

These decisions include: the choice between a deep or shallow embedding of the expression language and the programming language, the choice of programming language to model and the encoding of that language as embedded within Agda, and finally the choice of inference rules to be formalised for use within proofs of program correctness.

## 2.1 Shallow vs. Deep Embedding

For Hoare logic to be formalised within Agda, a simple imperative language for the Hoare logic assertions to apply to needs to be constructed and formalised first. This language itself, will actually comprise of *two* languages, the language defining the commands of the language (`WHILE_DO_` etc. . . ) and the expression/assertion language defining both the expressions that appear *within* those commands and the propositional assertions for reasoning.

Given that Agda is a programming language then, the task is to embed one language within another; a task that is actually rather common. So called *Domain Specific Languages*, as opposed to *General Purpose Languages*, are programming languages designed with a specific use case in mind. DSLs can be implimented via standalone syntax and semantics with their own compilation techniques but often they are instead *embedded* within a host language making use of that language's syntax, semantics, and compilation techniques and thus saving a lot of work for the implementer.

The choice one has to make when embedding a language within another is between a *shallow embedding* or a *deep embedding.* The two approaches are closely related with the principal difference being that in a shallow embedding, only the semantics are captured, whereas in a deep embedding the syntax itself is embedded along with some evaluation function, sometimes called an *observation function*; This function then essentially gives an operational semantics to the syntax of the embedded language, while in a shallow embedding the semantics is in terms of the host language's semantics.

As an illustrative example consider a simple expression language of arithmetic expressions with integer constants and addition. A deep embedding might have the form:

```
data Expr : Set where
  Val : Integer → Expr
  Add : Expr → Expr → Expr

eval : Expr → Integer
eval (Val n) = n
eval (Add x y) = eval x + eval y
```

With observation function *eval*. Meanwhile, a shallow embedding of the same language may have the form:

```
Expr = Integer

val : Integer → Expr
val n = n

add : Expr → Expr → Expr
add x y = x + y

eval : Expr → Integer
eval  =  id
```

Both approaches have their advantages. A deep embedding allows for the easy modification of evaluation functions without having to change the language itself. In fact, with a deep embedding, multiple evaluation functions can be given which amounts to being able to give multiple *non-compositional* (operational) semantics for the language; whether or not that counts as an advantage or disadvantage will depend on the context.

Meanwhile, a shallowly embedded language can only be given semantics compositionally through the host language but the pay-off for this is that it makes it much easier to change the embedded language as a small change will not necessitate a change in the evaluation function and all its dependents.

In this work then, a choice of embedding strategy needed to be made for the two languages, viz, the imperative language to be reasoned about and the expression language within that language. Ultimately the choice was made in favour of a *deep embedding* for *both*.

The primary justification for the imperative language to be a deep embedding is that it was decided that one of the primary intentions of this work, beyond formalising Hoare logic in Agda, would be to have a system wherein a small snippet of, say, a C program, could easily be translated into Agda, perhaps even automatically. Then a proof of its correctness could be constructed giving great confidence in the correctness of this snippet within its original program. This made it desirable that the embedded imperative language closely mirror a simple *real-world* language lest the user lose confidence that the program proved correct in Agda correctly captures the one they started with.

The other consideration is that a deep embedding of the imperative language allows for the giving of an operational semantics for that language in the form of its evaluation function. In that sense it could be said that the present work, beyond just formalising some Hoare logic, is giving both an axiomatic semantics *and* an operational semantics to a simple imperative language and showing that both are consistent with one another; thus expanding the scope of the formalisation.

The decision to have the assertion language deeply embedded, however, is a little harder to justify. As this is the language that will also form the logical assertions of the predicate calculus underlying the Hoare logic, it may seem wasteful to re-encode this first-order logic within the higher order logic underpinning Agda.

Agda already has an extensive standard library covering many of the definitions and theorems that may be needed while reasoning within the Hoare logic calculus. For instance, in Hoare logic, it is often necessary to prove $\mathcal{P} \Rightarrow \mathcal{Q}$ for some $\mathcal{P}$ and $\mathcal{Q}$ — e.g. when the user has $\{\!\{\ \mathcal{A}\ \}\!\}\ \mathcal{S}_1\ \{\!\{\ \mathcal{P}\ \}\!\}$ and $\{\!\{\ \mathcal{Q}\ \}\!\}\ \mathcal{S}_2\ \{\!\{\ \mathcal{B}\ \}\!\}$ and wants to derive $\{\!\{\ \mathcal{A}\ \}\!\}\ \mathcal{S}_1\ ;\mathcal{S}_2\ \{\!\{\ \mathcal{B}\ \}\!\}$ — and if, say, this $\mathcal{P}$ is some conjunction involving $\mathcal{Q}$, then what is required is a mechanism/proof corresponding to *conjunction elimination*. A deep embedding prevents a user from simply using the Agda standard library equivalent of conjunction elimination — the projections out of the product/sigma type — and instead necessitates re-proving this fact for the semantics of the embedded language itself.

Despite this drawback, the desire for this work to not only be a theoretical achievement but also to have potential practical use in future as a means of checking the correctness of *real* programs influenced the decision in favour of a deep embedding. This is in keeping with the spirit of Hoare's original paper in which the theory was being developed with a purpose in mind. The problem with relying on Agda's standard library is that the treatments therein don't necessarily capture real programming — most notably in the case of integers where the standard library definition obviously corresponds to the mathematical, *'true'*-integer that is unbounded in both directions, whereas in most programming languages, certainly the ones this work is aiming to reason about, an integer is bounded by dint of it being implemented by the compiler as, usually, either a 32 or 16-bit word.

Hoare was very particular in his original paper to only introduce arithmetic axioms that are true regardless of whether or not one was reasoning with the traditional infinite set of integers or within the programmer's finite sets of 'integers' and regardless of the choice of overflow strategy. AKDNGDNGLDSKNGLAK apdgmPASDMG APODGMPD DAPOGMDG PADGOKKPO AGDKG. AOAFKJ ASIFJL ASLIFJ LASFJ LASFJ LKJLAFK ALSKF ALSKF FSA.

In keeping with this spirit, a deep embedding, it was reasoned, would force the user of this library to think consciously about the inferences that are being used and whether or not they really hold within the 'real' world. The intention being that if a proof of correctness depended, say, on a particular overflow strategy, this fact would be rendered explicit on the way to constructing that proof.

The end result of these twin choices of embedding strategy is that only the Hoare logic caluculus itself takes place within Agda's higher order logic with the rest of the formalisation busying itself *within* the deep embedding of one of the two languages. Despite this, there will be very little, if any, reason to use Agda's standard library and higher expressive power within the process of proving correct a simple program within this library as the rules provided should be sufficient for programs within the scope of the library's capabilities.

## 2.2 Proof Obligation Interfaces

As was mentioned in the previous section, the decision to go with a deep embedding for the expression/assertion language brings with it a major drawback. While some users may wish to rigorously prove all aspects of a program correct, ideally, the user shouldn't be *forced* to re-prove simple, obvious, and banal lemmas when proving a program correct in the library.

This led to the decision to build into the formalisation/library a pair of interfaces using Agda's record types — a generalisation of the dependent product type. These two interfaces being:

- **Data-Interface**: abstract out the reperesentation of the identifiers, values, and operations thereupon.
- **State-Interface**: abstract out the representation of the state space.

The intention of these interfaces would be twofold. First, to allow the user to forestall — perhaps indefinitely — the obligaton of proving simple or obvious lemmas when proving a program correct within the library. And secondly, to separate out the concerns and hide implementation details that are adjunct to Hoare logic but not the main concern in any proof of correctness constructed therein. For example, while reasoning within the Hoare-logic calculus it is undesirable to have knowledge or use of the fact that the state space is represented within the formalisation in a particular way, say, as a list of pairs of identifiers and variables, as no proof should depend on the exact choice of representation.

A sketch of the interfaces as currently defined in the library are given in figure 1. A user of this library would be able to add any needed lemmas to these interfaces as required for any proof of correctness being worked upon. The user that is after a total formalisation then, can go on to prove correct the added inference rules or axioms, within a given instantiation of the interface. Such an instantiation is bound by the definition of the interface to properly identify its arithmetic and overflow strategy thus forcing the explicit consideration on the part of the user of such matters. Alternatively, the user interested only in the mechanics of the Hoare logic calculus can forgo instantiating the interface indefinitely and still construct a proof of correctness.

**Figure 1:** Sketch of Data-Interface and State-Interface

```
DATA-INTERFACE:
    data:                       functions:
    Id    :  Set                WFF                  :  Val → Set
    Val   :  Set                to𝔹Val               :  (v : Val)
    variables:                                            → WFF v
    𝓍     :  Val                                          → Bool
    𝓎     :  Val                arith. rules from [7]:
    𝓏     :  Val                A1                   :  x+y≡y+x
                                ⋮
    constants:
    ⓪     :  Val                A9                   :  x*①≡x
    ①     :  Val                ARITHMETIC-STRATEGY  :  ...
    ②     :  Val                OVERFLOW-STRATEGY    :  ...
    operations:                 propositional rules:
    _||_  :  ...                DeMorgan₁            :  ...
    _&&_  :  ...                DeMorgan₂            :  ...
    ⋮                           ConjunctionElim_left :  ...
                                ⋮
    _+_   :  ...
    _*_   :  ...                NegationElim         :  ...

STATE-INTERFACE:
    definition of state space:
    𝓢     :  Set
    empty/initial state:
    ●     :  𝓢
    state operations:
    updateState          :  Id → Val → 𝓢 → 𝓢
    getIdVal             :  Id → 𝓢 → Maybe Val
    dropValue            :  Id → 𝓢 → 𝓢
    state space lemmas as needed:
    ⋮
```

NB that the interfaces have been instantiated in full as part of this work with $\textsc{Id} \overset{\text{def}}{=} \mathbb{N}$ and $\textsc{Val} \overset{\text{def}}{=} (\mathbb{Z} \texttt{ x } \textsc{Bool})$ — where $\mathbb{N}$, $\mathbb{Z}$, and $\textsc{Bool}$ are the Agda standard library versions. Implicit casting is then assumed between Ints and Bools within the Val type and the state space is the type of Lists of pairs of $\textsc{Id}$'s and $\textsc{Val}$'s — i.e. $\mathcal{S} \overset{\text{def}}{=} (\textsc{List}\ (\textsc{Id} \texttt{ x } \textsc{Val}))$.

## 2.3 The Exppresion and/or Assertion Language

The specification of the expression language is straightforward with the intent being that it closely mirror the array of operands available in most imperative languages. It is described, as it appears in the formalisation, via the following context-free grammar given in Backus-Naur form:

$$\begin{array}{lcl}
\texttt{<Exp>} & ::= & \texttt{<Exp> <Op}_2\texttt{> <Exp> | <Op}_1\texttt{> <Exp> | <terminal>} \\
\texttt{<Op}_2\texttt{>} & ::= & \texttt{\&\&}_o \texttt{ | ||}_o \texttt{ | ==}_o \texttt{ | } \leq_o \texttt{ | ... | +}_o \texttt{ | -}_o \texttt{ | ... | \%}_o \\
\texttt{<Op}_1\texttt{>} & ::= & \texttt{++}_o \texttt{ | -}_o \texttt{ | } \neg_o \texttt{ | -}_o \\
\texttt{<terminal>} & ::= & \texttt{const <num> | var <id> | } \mathit{true} \texttt{ | } \mathit{false} \\
\texttt{<num>} & ::= & \texttt{1 | 2 | 3 | ...} \\
\texttt{<id>} & ::= & \texttt{x | } y \texttt{ | z | ...}
\end{array}$$

Evaluation of these expressions is defined with respect to the operator instantiations abstracted away behind the data-interface. If assertions are to be precisely expressions, however, the language given above may seem to allow for some unusual assertions that may raise an eyebrow. What do the assertions `(2 + 1)` or (x * 5) mean? The problem is that, as mentioned on page 6, expressions need to *at least* be a subset of assertions to allow for the substitution of the former into the latter.

An incredibly baroque solution to this problem would be enforcing a type system that distinguishes between Boolean variables and Integer variables within the deep embedding. The alternative, much simpler, solution that has been opted for here is to assume implicit casting between Ints and Bools within both the expressions and the assertions, drastically simplifying both.

Following C, C++, and most other languages with implicit casting, any non-zero integer is taken to have truth-value *true*, and zero, a truth-value of *false*. Thus the expressions `(2 + 4)` and `(4 * 0)` are valid assertions having constant values *true* and *false* respectively.

The next complication involves the handling of stuck expressions. Is the assertion (`x` == (`y / 0`)) to be read as *false*? Is it even an Assertion? The assertions in Hoare Logic (or assertions in general) are understood to be boolean-valued functions over the state space, but with the present treatment some assertions are only partial functions of the state space as the truth value of any assertion with a variable is undefined in all states in which that variable is not defined; as is any assertion that contains a division by zero error.

This is a problem often brushed aside casually - if mentioned at all - in typical expositions of the subject and it is easy to see why — any sensible programmer will avoid writing code where the non-zero-ness of a divisor is not obvious and a variable that is undefined will immidiately make itself apparent. Unfortunately, in a constructive formalisation such as this one, sweeping things under the table is not an option nor desirable so the complication must be addressed.

Semantically, this problem is resolved by Dijkstra in [4] by introducing a predicate into the expression/assertion language of the form $\mathcal{D}(\mathcal{E})$ which returns true when the given state lies within the domain of the expression $\mathcal{E}$. The weakest precondition of the assignment mechanism is then rewritten as:

$$wp(\mathtt{x} \coloneqq \mathcal{E}, \mathcal{R}) = \{\mathcal{D}(\mathcal{E})\ \mathtt{cand}\ \mathtt{(sub}\ \mathcal{E}\ \mathtt{x}\ \mathcal{R}\mathtt{)}\}$$

…with `cand` being the conditional boolean `&&` that only evaluates the second argument if necessary. In essence, the semantics are changed so that any stuck assertion will be rendered as *false*. From the perspective of Hoare logic — from outside the deep embedding — this solution seems reasonable as with Hoare logic being a *deductive system*, it is only whether or not assertions are true that is of concern, not the conditions under which they fail to be so.

With that said, this only answers the question of how stuck expressions are to be treated *semantically*, not how to handle the issue *syntactically* within this formalisation. Perhaps $\mathcal{D}(\mathcal{E})$ could be added to the expression language as the *well-formed-ness* of an expression in a given state can be defined inductively and checked mechanically. However, making this change would also change the semantics of the imperative language that is also to be embedded.

It is obviously undesirable to have '$(\mathtt{x} == (\mathtt{y}\ /\ 0)) \overset{\text{def}}{=} \mathit{false}$' within the semantics of the *programming* language that users of this library are to form the programs they want to prove correct as no sensible language should allow '`IF ( ¬ (X == (Y / 0))) ...`' to evaluate[16] - not to mention the fact that this would be a deviation from the intention outlined previously for the formalised imperative language to mirror real world languages.

So the desired state of affairs is to have stuck expressions be undefined within the programming language but equate them to false without. The solution used to achieve this was to modify the data interface so that all

[16]What on earth would it even evaluate to?

operations — and by extension the expression `eval` function — had the option of failing via wrapping the output of each in the MAYBE type.

With this decision made, the definition of a *well-formed-formula* could be given simply in terms of evaluation. i.e. an expression/assertion is a well-formed formula if and only if it can be evaluated successfully:

```
WFF : Assertion → S → Set
WFF a s = Is-just (evalExp a s)
```

Wth that in place, an assertion *proper* for the sake of Hoare logic can be represented as follows:

```
Assert : ∀ s A → Set
Assert s A = Σ (WFF A s) (T ∘ toTruthValue)
- Alternative, condensed syntax:
_⊨_ : ∀ s A → Set
s ⊨ A = Assert s A
```

That is, to assert an expression/assertion is to prove it a WFF such that this WFF has truth value *true*. This allows a definition to be given of what it means for one assertion to imply another:

```
_⇒_ : Assertion → Assertion → Set
P ⇒ Q = (s : S) → s ⊨ P → s ⊨ Q
```

Followed finally by an inference example showcasing how assertions are to be embedded and manipulated within the library:

```
a₁ ≝ χ == 2 ∧ y == 1                      a₂ ≝ χ == 2
private a₁ : Assertion                    private a₂ : Assertion
a₁ = ((val χ) == (const ②))               a₂ = (val χ) == (const ②)
       ∧
     ((val y) == (const ①))
```

```
inferenceExample : a₁ ⇒ a₂
inferenceExample s ⊨χ&y = let x = getIdVal χ s ==ᵥ (just ②) in
                          let y = getIdVal y s ==ᵥ (just ①) in
                          ConjunctionElim_left x y ⊨χ&y
```

## 2.4 The 'Mini-Imp' Programming Language

The design and embedding of the imperative language is far simpler than that of the expression language. A simple while-language coined 'Mini-Imp' was devised containing only the programming constructs that are present in [7] and [4] only without non-determinism present in the iterative (`WHILE_DO_`) and alternative (`IF_THEN_ELSE_`) commands; again, this is in keeping with the intention for the language to closely mirror simple real-world languages. The programming constructs themselves are defined as state transformers ($S\Delta$) with a program being a non-empty sequence of these state transformers:

```
data SΔ : Set where
  skip : SΔ
  WHILE_DO_ : Exp → Program → SΔ
  IF_THEN_ELSE_ : Exp → Program → Program → SΔ
  _:=_ : Id → Exp → SΔ

data Program : Set where
  - Terminator:
  _; : SΔ → Program
  - Separator:
  _;_ : SΔ → Program → Program
```

The overloaded terminator/separator construct allows for the terse and familiar encoding of programs but does, however, necessitate a third function for program composition which as it turns out is simply list concatenation:

```
- Program composition
_THEN_ : Program → Program → Program
(c ;) THEN b = c ; b
(c ; b₁) THEN b₂ = c ; (b₁ THEN b₂)
- Commutativity of program composition
THEN-comm : ∀ c₁ c₂ c₃ →
  c₁ THEN (c₂ THEN c₃) ≡ (c₁ THEN c₂) THEN c₃
THEN-comm (sΔ ;) c₂ c₃ = refl
THEN-comm (sΔ ; c₁) c₂ c₃
  rewrite THEN-comm c₁ c₂ c₃ = refl
```

With both the expression language and Mini-Imp defined, see figure 2 for some examples of full programs encoded within the Agda library.

**Figure 2:** Some simple programs defined with Mini-Imp; ripe for reasoning!

```
- Euclids Algorithm for GCD
gcd : (X Y : Exp) → Program
gcd X Y =
    x := X ;
    y := Y ;
  (WHILE (not ( val x == val y ))
    DO (IF ( val x > val y )
       THEN (
         x := val x - val y ;)
       ELSE (
         y := val y - val x ;) ;) );
```

```
- Multiply X and Y, and store in z
- without using multiplication op.
- ((11.4) in TSOP,Gries)
add* : (X Y : Exp) → Program
add* X Y =
      x := X ;
      y := Y ;
      z := const ⓪ ;
  (WHILE
    ( ( val y > const ⓪ ∧ even ⟨ val y ⟩ )
        ∨ ( odd ⟨ val y ⟩ )
    )
      DO (IF ( even ⟨ val y ⟩ )
         THEN (
           y := val y / const ② ;
           x := val x + val x ;)
         ELSE (
           y := val y - const ① ;
           z := val z + val x ;) ;) );
```

The end result of these two deep embeddings then, is that programs can be encoded directly within Agda (see figure 2) in a manner that is imminently intelligible; something that cannot often be said of Agda syntax.

## 2.5 The Rules to be Implemented

With the Mini-Imp langauage specified a rough sketch of the rules to be formalised can be given with the apparatus supporting these Agda definitions to be expounded upon in the next section. First the axiom of assignment:

```
D0-Axiom-of-Assignment : ∀ i e P


                  -- ------------------------ -
                     → « (sub e i P) » ( i := e ; ) « P »
```

Follwed by the two rules of consequence ('D1-Rule-of-Consequence-pre' is omitted as it has the obvious corresponding form to the one below):

```
D1-Rule-of-Consequence-post : ∀ {P} {Q} {R} {S}

                  → « P » Q « R » → R ⇒ S
            -- ----------------------- -
                  → « P » Q « S »
```

Then the rule of composition for the chaining of Hoare triples together.[17]

```
D2-Rule-of-Composition : ∀ {P} {R₁} {R} {Q₁} {Q₂}

              → « P » Q₁ « R₁ » → « R₁ » Q₂ « R »
         -- ------------------------------- -
              → « P » Q₁ THEN Q₂ « R »
```

And finally, most interestingly, the iterative and alternative rules:

```
D3-While-Rule : ∀ {P} {B} {C}

                  → « P ∧ B » C « P »
         -- ------------------------------- -
              → « P » WHILE B DO C ; « ( not B ) ∧ P »
```

```
D4-Conditional-Rule : ∀ {A} {B} {C} {P} {Q}

          → « C ∧ P » A « Q » → « ( not C ) ∧ P » B « Q »
     -- ---------------------------------------- -
          → « P » IF C THEN A ELSE B ; « Q »
```

And with that, the Hoare logic inference rules that are to be formalised within this work have been specified.

[17]NB That «$P$»$Q$«$R$» is the notation within the codebase for $\{\!\{\, \mathcal{P} \,\}\!\}\ \mathcal{Q}\ \{\!\{\, \mathcal{R} \,\}\!\}$ as '{' and '}' are reserved for Agda's syntax.

## 3 Implementation Details and Challenges

With the syntactic aspects out the way, this section covers the semantics of those syntactic aspects as well as some of the more nuanced or tricky aspects of the formalisation.

### 3.1 Evaluation of Programs & Termination

As mentioned in section 2.1, the deep embedding of the Mini-Imp language needs some form of observation or evaluation function to give it semantics. This presented a challenge as Agda demands that all functions be total and features a rather strict termination checker that will only accept functions that it can mechanically prove terminating. This termination checker only checks for *structural recursion* and so some argument of the evaluation function must get structurally smaller on each call. This left the only feasible way forward being to give the Mini-Imp language semantics via a small-step (operational) semantics which then allowed for the evaluation function to take a 'fuel' argument ($\in \mathbb{N}$) that could be decremented with each call, giving the form: ssEvalwithFuel :ℕ → Program → S → Maybe S. The implementation of this function is relatively straightforward, if a little verbose, and the two most interesting cases are reproduced here:

```
---------------------------------
-- SINGLE WHILE
ssEvalwithFuel (suc n) ( WHILE exp DO c ;) s with evalExp exp s
... | nothing = nothing -- Computation failed e.g. div by 0
... | f @ (just _) with toTruthValue {f} (Any.just tt)
... | true = ssEvalwithFuel n ( c THEN WHILE exp DO c ;) s
... | false = just s
---------------------------------
---------------------------------
-- WHILE ; THEN C₂
ssEvalwithFuel (suc n) ((WHILE exp DO c₁) ; c₂) s
  with evalExp exp s
... | nothing = nothing -- Computation failed e.g. div by 0
... | f @ (just _) with toTruthValue {f} (Any.just tt)
... | true = ssEvalwithFuel n (c₁ THEN ((WHILE exp DO c₁) ; c₂)) s
... | false = ssEvalwithFuel n c₂ s
---------------------------------
```

With evaluation defined, termination of a program can now be formalised like so:

```
Terminates : C → S → Set
Terminates c s = Σ[ f ∈ ℕ ] ( Is-just (ssEvalwithFuel f c s ))

⌊ᵗ _,_ ᵗ⌋ : C → S → Set
⌊ᵗ _,_ ᵗ⌋ = Terminates

TerminatesWith : ℕ → C → S → Set
TerminatesWith f c s = Is-just (ssEvalwithFuel f c s)

⌊ᵗ _,_,_ ᵗ⌋ : ℕ → C → S → Set
⌊ᵗ f , c , s ᵗ⌋ = TerminatesWith f c s
```

Pellentesque interdum sapien sed nulla. Proin tincidunt. Aliquam volutpat est vel massa. Sed dolor lacus, imperdiet non, ornare non, commodo eu, neque. Integer pretium semper justo. Proin risus. Nullam id quam. Nam neque. Duis vitae wisi ullamcorper diam congue ultricies. Quisque ligula. Mauris vehicula.

```
EvalDet : ∀ {s f f'} C → (a : ⌊ᵗ f , C , s ᵗ⌋) → (b : ⌊ᵗ f' , C , s ᵗ⌋) → † a ≡ † b
```

Nullam eleifend justo in nisl. In hac habitasse platea dictumst. Morbi nonummy. Aliquam ut felis. In velit leo, dictum vitae, posuere id, vulputate nec, ante. Maecenas vitae pede nec dui dignissim suscipit. Morbi magna. Vestibulum id purus eget velit laoreet laoreet. Praesent sed leo vel nibh convallis blandit. Ut rutrum. Donec nibh. Donec interdum. Fusce sed pede sit amet elit rhoncus ultrices. Nullam at enim vitae pede vehicula iaculis.

```
record Split-⌊ᵗ⌋ s f Q₁ Q₂ (Φ : ⌊ᵗ f , Q₁ THEN Q₂ , s ᵗ⌋) : Set where
  field
    --- Termination Left
    Lᵗ : ⌊ᵗ f , Q₁ , s ᵗ⌋
    --- There's an f' s.t.
    f' : ℕ
    --- Termination Right
    Rᵗ : ⌊ᵗ f' , Q₂ , († Lᵗ) ᵗ⌋
    --- and 2nd proof fuel is less than starting fuel:
    lt : f' ≤'' f
    --- And the output unchanged:
    Δ : † Rᵗ ≡ † Φ
```

Donec a nibh ut elit vestibulum tristique. Integer at pede. Cras volutpat varius magna. Phasellus eu wisi. Praesent risus justo, lobortis eget, scelerisque ac, aliquet in, dolor. Proin id leo. Nunc iaculis, mi vitae accumsan commodo, neque sem lacinia nulla, quis vestibulum justo sem in eros. Quisque sed massa. Morbi lectus ipsum, vulputate a, mollis ut, accumsan placerat, tellus. Nullam in wisi. Vivamus eu ligula a nunc accumsan congue. Suspendisse ac libero. Aliquam erat volutpat. Donec augue. Nunc venenatis fringilla nibh. Fusce accumsan pulvinar justo. Nullam semper, dui ut dignissim auctor, orci libero fringilla massa, blandit pulvinar pede tortor id magna. Nunc adipiscing justo sed velit tincidunt fermentum.

```
«_»_«_» : Assertion → C → Assertion → Set
« P » C « Q » = ( s : S ) → s ⊨ P → (Φ : ⌊ᵗ C , s ᵗ⌋) → († Φ) ⊨ Q
```

Nunc sed pede. Praesent vitae lectus. Praesent neque justo, vehicula eget, interdum id, facilisis et, nibh. Phasellus at purus et libero lacinia dictum. Fusce aliquet. Nulla eu ante placerat leo semper dictum. Mauris metus. Curabitur lobortis. Curabitur sollicitudin hendrerit nunc. Donec ultrices lacus id ipsum.

```
⟦_⟧_⟦_⟧ : Assertion → C → Assertion → Set
⟦ P ⟧ C ⟦ Q ⟧ = ( s : S ) → s ⊨ P → Σ ⌊ᵗ C , s ᵗ⌋ (λ Φ → († Φ) ⊨ Q)
```

Nunc sed pede. Praesent vitae lectus. Praesent neque justo, vehicula eget, interdum id, facilisis et, nibh. Phasellus at purus et libero lacinia dictum. Fusce aliquet. Nulla eu ante placerat leo semper dictum. Mauris metus. Curabitur lobortis. Curabitur sollicitudin hendrerit nunc. Donec ultrices lacus id ipsum.

```
        LawOfExcludedMiracle-wp (:=,-) : ∀ {i e} → sub e i F ≡ F
```

```
    LawOfExcludedMiracle-wp (:=,-) = refl
```

Nunc sed pede. Praesent vitae lectus. Praesent neque justo, vehicula eget, interdum id, facilisis et, nibh. Phasellus at purus et libero lacinia dictum. Fusce aliquet. Nulla eu ante placerat leo semper dictum. Mauris metus. Curabitur lobortis. Curabitur sollicitudin hendrerit nunc. Donec ultrices lacus id ipsum.

Pellentesque interdum sapien sed nulla. Proin tincidunt. Aliquam volutpat est vel massa. Sed dolor lacus, imperdiet non, ornare non, commodo eu, neque. Integer pretium semper justo. Proin risus. Nullam id quam. Nam neque. Duis vitae wisi ullamcorper diam congue ultricies. Quisque ligula. Mauris vehicula.

Nunc sed pede. Praesent vitae lectus. Praesent neque justo, vehicula eget, interdum id, facilisis et, nibh. Phasellus at purus et libero lacinia dictum. Fusce aliquet. Nulla eu ante placerat leo semper dictum. Mauris metus. Curabitur lobortis. Curabitur sollicitudin hendrerit nunc. Donec ultrices lacus id ipsum.

Nunc sed pede. Praesent vitae lectus. Praesent neque justo, vehicula eget, interdum id, facilisis et, nibh. Phasellus at purus et libero lacinia dictum. Fusce aliquet. Nulla eu ante placerat leo semper dictum. Mauris metus. Curabitur lobortis. Curabitur sollicitudin hendrerit nunc. Donec ultrices lacus id ipsum.

Nunc sed pede. Praesent vitae lectus. Praesent neque justo, vehicula eget, interdum id, facilisis et, nibh. Phasellus at purus et libero lacinia dictum. Fusce aliquet. Nulla eu ante placerat leo semper dictum. Mauris metus. Curabitur lobortis. Curabitur sollicitudin hendrerit nunc. Donec ultrices lacus id ipsum.

Nunc sed pede. Praesent vitae lectus. Praesent neque justo, vehicula eget, interdum id, facilisis et, nibh. Phasellus at purus et libero lacinia dictum. Fusce aliquet. Nulla eu ante placerat leo semper dictum. Mauris metus. Curabitur lobortis. Curabitur sollicitudin hendrerit nunc. Donec ultrices lacus id ipsum.

Donec a nibh ut elit vestibulum tristique. Integer at pede. Cras volutpat varius magna. Phasellus eu wisi. Praesent risus justo, lobortis eget, scelerisque ac, aliquet in, dolor. Proin id leo. Nunc iaculis, mi vitae accumsan commodo, neque sem lacinia nulla, quis vestibulum justo sem in eros. Quisque sed massa. Morbi lectus ipsum, vulputate a, mollis ut, accumsan

**Figure 3:** D3-While: Full proof of the while rule; the crucial rule for reasoning with Hoare Logic:

```
D3-While-Rule {P} {B} {C} PBCP s ⊨P (suc f , ⌊ᵗ Cᵗ ⌋) = go (suc f) ⊨P ⌊ᵗ Cᵗ ⌋
  where
  ------------------------------
  - Using mutually recursive functions go and go-true
  go : ∀ {s} f → s ⊨ P → (⌊ᵗ Cᵗ ⌋ : ⌊ᵗ f , (WHILE B DO C ;) , s ᵗ⌋)
      → († ⌊ᵗ Cᵗ ⌋) ⊨ (op₂ (op₁ ¬ₒ B) &&ₒ P )
  - f needs to be an argument by itself outside the Sigma type
  - so we can recurse on it as Agda can't see it always decrements
  - with each call if it is inside the product.
  --------------------------------
  - case where B is true
  go-true : ∀ {s} {f} {v} → s ⊨ P → (evalExp B s ≡ just v)
             → (toTruthValue {just v} (just tt) ≡ true)
             → (⌊ᵗ Cᵗ ⌋ : ⌊ᵗ f , (C THEN WHILE B DO C ;) , s ᵗ⌋)
             → (to-witness ⌊ᵗ Cᵗ ⌋) ⊨ (op₂ (op₁ ¬ₒ B) &&ₒ P)
  go-true {s} {f} ⊨P p₁ p₂ ⌊ᵗ Cᵗ ⌋
    with ⌊ᵗ ⌋-split f s C (WHILE B DO C ;) ⌊ᵗ Cᵗ ⌋
  ... | record { Lᵗ = Lᵗ ; f' = f' ; Rᵗ = Rᵗ ; lt = lt ; Δ = Δ } = Λ
    where
    ⊨B : s ⊨ B
    ⊨B rewrite p₁ = (just tt , subst T (sym p₂) tt)
    ⊨P&B : s ⊨ (op₂ P &&ₒ B)
    ⊨P&B = ConjunctionIntro _ _ ⊨P ⊨B
    ⊨P' : († Lᵗ) ⊨ P
    ⊨P' = PBCP s ⊨P&B (f , Lᵗ)
    - Proof of termination of rhs of split with f'
    Rᵗ+ : ⌊ᵗ f' + (k lt) , (WHILE B DO C ;) , († Lᵗ) ᵗ⌋
    Rᵗ+ = addFuel {WHILE B DO C ;} f' (k lt) Rᵗ
    - f' with (f' ≤ f) implies termination with f fuel
    Rᵗf : ⌊ᵗ f , (WHILE B DO C ;) , († Lᵗ) ᵗ⌋
    Rᵗf = let C₁ = (WHILE B DO C ;) in subst
            (λ f → ⌊ᵗ f , C₁ , († Lᵗ) ᵗ⌋) (proof lt) Rᵗ+

                                  ⋮
```

**Figure 3:** D3-While: Full proof of the while rule; the crucial rule for reasoning with Hoare Logic:

cont.

⋮

```
- This new proof of termination R^t f has same output
isDet : † R^t f ≡ † R^t
isDet = EvalDet {_} {f} {f'} (WHILE B DO C ;) R^t f R^t
- and said output is identical to the original output
Δ' : † R^t f ≡ † ⌊^t C^t⌋
Δ' rewrite isDet = Δ
- which we can now use in a recursive call:  (suc f) ⇒ f
GO : († R^t f) ⊨ (op_2 (op_1 ¬_o B) &&_o P)
GO = go {† L^t} f ⊨P' R^t f

- and finally get the type we need via substitution with Δ'
Λ : († ⌊^t C^t⌋) ⊨ (op_2 (op_1 ¬_o B) &&_o P)
Λ = subst (λ s → s ⊨ (op_2 (op_1 ¬_o B) &&_o P)) Δ' GO
- case where B is false
go-false : ∀ {s} {v} → s ⊨ P → (evalExp B s ≡ just v)
           → (toTruthValue {just v} (just tt) ≡ false)
           → s ⊨ (op_2 (op_1 ¬_o B) &&_o P)
go-false {s} {v} ⊨P p_1 p_2 = ConjunctionIntro _ _ ⊨¬B ⊨P
  where
  ⊭B : ⊭ (just v)
  ⊭B rewrite p_1 = (just tt) , subst (T ∘ not) (sym p_2) tt
  ⊨¬B : s ⊨ (op_1 ¬_o B)
  ⊨¬B rewrite p_1 = (NegationIntro (just v) (⊭B))
--------------------------------
go {s} (suc f) ⊨P ⌊^t C^t⌋ with
      evalExp B s | inspect (evalExp B) s
... | f@(just v) | [ p_1 ] with
      toTruthValue {f} (any tt) | inspect (toTruthValue {f}) (any tt)
... | true | [ p_2 ] = go-true {s} {f} ⊨P p_1 p_2 ⌊^t C^t⌋
... | false | [ p_2 ] rewrite Is-just-just ⌊^t C^t⌋ = go-false ⊨P p_1 p_2
--------------------------------
```

placerat, tellus. Nullam in wisi. Vivamus eu ligula a nunc accumsan congue. Suspendisse ac libero. Aliquam erat volutpat. Donec augue. Nunc venenatis fringilla nibh. Fusce accumsan pulvinar justo. Nullam semper, dui ut dignissim auctor, orci libero fringilla massa, blandit pulvinar pede tortor id magna. Nunc adipiscing justo sed velit tincidunt fermentum.

Integer placerat. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Sed in massa. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Phasellus tempus aliquam risus. Aliquam rutrum purus at metus. Donec posuere odio at erat. Nam non nibh. Phasellus ligula. Quisque venenatis lectus in augue. Sed vestibulum dapibus neque.

Mauris tempus eros at nulla. Sed quis dui dignissim mauris pretium tincidunt. Mauris ac purus. Phasellus ac libero. Etiam dapibus iaculis nunc. In lectus wisi, elementum eu, sollicitudin nec, imperdiet quis, dui. Nulla viverra neque ac libero. Mauris urna leo, adipiscing eu, ultrices non, blandit eu, dui. Maecenas dui neque, suscipit sit amet, rutrum a, laoreet in, eros. Ut eu nibh. Fusce nec erat tempus urna fringilla tempus. Curabitur id enim. Sed ante. Cras sodales enim sit amet wisi. Nunc fermentum consequat quam.

## 3.2 Small Step Evaluation with Fuel

Nunc sed pede. Praesent vitae lectus. Praesent neque justo, vehicula eget, interdum id, facilisis et, nibh. Phasellus at purus et libero lacinia dictum. Fusce aliquet. Nulla eu ante placerat leo semper dictum. Mauris metus. Curabitur lobortis. Curabitur sollicitudin hendrerit nunc. Donec ultrices lacus id ipsum.

Donec a nibh ut elit vestibulum tristique. Integer at pede. Cras volutpat varius magna. Phasellus eu wisi. Praesent risus justo, lobortis eget, scelerisque ac, aliquet in, dolor. Proin id leo. Nunc iaculis, mi vitae accumsan commodo, neque sem lacinia nulla, quis vestibulum justo sem in eros. Quisque sed massa. Morbi lectus ipsum, vulputate a, mollis ut, accumsan placerat, tellus. Nullam in wisi. Vivamus eu ligula a nunc accumsan congue. Suspendisse ac libero. Aliquam erat volutpat. Donec augue. Nunc venenatis fringilla nibh. Fusce accumsan pulvinar justo. Nullam semper, dui ut dignissim auctor, orci libero fringilla massa, blandit pulvinar pede tortor id magna. Nunc adipiscing justo sed velit tincidunt fermentum.

Integer placerat. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Sed in massa. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Phasellus tempus aliquam risus. Aliquam rutrum purus at metus. Donec posuere odio at erat. Nam non nibh. Phasellus ligula. Quisque venenatis lectus in augue. Sed vestibulum dapibus neque.

Mauris tempus eros at nulla. Sed quis dui dignissim mauris pretium tincidunt. Mauris ac purus. Phasellus ac libero. Etiam dapibus iaculis nunc. In lectus wisi, elementum eu, sollicitudin nec, imperdiet quis, dui. Nulla viverra neque ac libero. Mauris urna leo, adipiscing eu, ultrices non, blandit eu, dui. Maecenas dui neque, suscipit sit amet, rutrum a, laoreet in, eros. Ut eu nibh. Fusce nec erat tempus urna fringilla tempus. Curabitur id enim. Sed ante. Cras sodales enim sit amet wisi. Nunc fermentum consequat quam.

## 3.3 Termination Splitting

Nunc sed pede. Praesent vitae lectus. Praesent neque justo, vehicula eget, interdum id, facilisis et, nibh. Phasellus at purus et libero lacinia dictum. Fusce aliquet. Nulla eu ante placerat leo semper dictum. Mauris metus. Curabitur lobortis. Curabitur sollicitudin hendrerit nunc. Donec ultrices lacus id ipsum.

Donec a nibh ut elit vestibulum tristique. Integer at pede. Cras volutpat varius magna. Phasellus eu wisi. Praesent risus justo, lobortis eget, scelerisque ac, aliquet in, dolor. Proin id leo. Nunc iaculis, mi vitae accumsan commodo, neque sem lacinia nulla, quis vestibulum justo sem in eros. Quisque sed massa. Morbi lectus ipsum, vulputate a, mollis ut, accumsan placerat, tellus. Nullam in wisi. Vivamus eu ligula a nunc accumsan congue. Suspendisse ac libero. Aliquam erat volutpat. Donec augue. Nunc venenatis fringilla nibh. Fusce accumsan pulvinar justo. Nullam semper, dui ut dignissim auctor, orci libero fringilla massa, blandit pulvinar pede tortor id magna. Nunc adipiscing justo sed velit tincidunt fermentum.

Integer placerat. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Sed in massa. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Phasellus tempus aliquam risus. Aliquam rutrum purus at metus. Donec posuere odio at erat. Nam non nibh. Phasellus ligula. Quisque venenatis lectus in augue. Sed vestibulum dapibus neque.

Mauris tempus eros at nulla. Sed quis dui dignissim mauris pretium tincidunt. Mauris ac purus. Phasellus ac libero. Etiam dapibus iaculis nunc. In lectus wisi, elementum eu, sollicitudin nec, imperdiet quis, dui. Nulla viverra neque ac libero. Mauris urna leo, adipiscing eu, ultrices non, blandit eu, dui. Maecenas dui neque, suscipit sit amet, rutrum a, laoreet in, eros. Ut eu nibh. Fusce nec erat tempus urna fringilla tempus. Curabitur id enim. Sed ante. Cras sodales enim sit amet wisi. Nunc fermentum consequat quam.

Ut auctor, augue porta dignissim vestibulum, arcu diam lobortis velit, vel scelerisque risus augue sagittis risus. Maecenas eu justo. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris congue ligula eget tortor. Nullam laoreet urna sed enim. Donec eget eros ut eros volutpat convallis. Praesent turpis. Integer mauris diam, elementum quis, egestas ac, rutrum vel, orci. Nulla facilisi. Quisque adipiscing, nulla vitae elementum porta, sem urna volutpat leo, sed porta enim risus sed massa. Integer ac enim quis diam sodales luctus. Ut eget eros a ligula commodo ultricies. Donec eu urna viverra dolor hendrerit feugiat. Aliquam ac orci vel eros congue pharetra. Quisque rhoncus, justo eu volutpat faucibus, augue leo posuere lacus, a rhoncus purus pede vel est. Proin ultrices enim.

Aenean tincidunt laoreet dui. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Integer ipsum lectus, fermentum ac, malesuada in, eleifend ut, lorem. Vivamus ipsum turpis, elementum vel, hendrerit ut, semper at, metus. Vivamus sapien tortor, eleifend id, dapibus in, egestas et, pede. Pellentesque faucibus. Praesent lorem neque, dignissim in, facilisis nec, hendrerit vel, odio. Nam at diam ac neque aliquet viverra. Morbi dapibus ligula sagittis magna. In lobortis. Donec aliquet ultricies libero. Nunc dictum vulputate purus. Morbi varius. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. In tempor. Phasellus commodo porttitor magna. Curabitur vehicula odio vel dolor.

Praesent facilisis, augue a adipiscing venenatis, libero risus molestie odio, pulvinar consectetuer felis erat ac mauris. Nam vestibulum rhoncus quam. Sed velit urna, pharetra eu, eleifend eu, viverra at, wisi. Maecenas ultrices nibh at turpis. Aenean quam. Nulla ipsum. Aliquam posuere luctus erat. Curabitur magna felis, lacinia et, tristique id, ultrices ut, mauris. Suspendisse feugiat. Cras eleifend wisi vitae tortor. Phasellus leo purus, mattis sit amet, auctor in, rutrum in, magna. In hac habitasse platea dictumst. Phasellus imperdiet metus in sem. Vestibulum ac enim non sem ultricies sagittis. Sed vel diam.

Integer vel enim sed turpis adipiscing bibendum. Vestibulum pede dolor, laoreet nec, posuere in, nonummy in, sem. Donec imperdiet sapien placerat erat. Donec viverra. Aliquam eros. Nunc consequat massa id leo. Sed ullamcorper, lorem in sodales dapibus, risus metus sagittis lorem, non porttitor purus odio nec odio. Sed tincidunt posuere elit. Quisque eu enim. Donec libero risus, feugiat ac, dapibus eget, posuere a, felis. Quisque vel lectus ut metus tincidunt eleifend. Duis ut pede. Duis velit erat, venenatis vitae, vulputate a, pharetra sit amet, est. Etiam fringilla faucibus augue.

Aenean velit sem, viverra eu, tempus id, rutrum id, mi. Nullam nec nibh. Proin ullamcorper, dolor in cursus tristique, eros augue tempor nibh, at gravida diam wisi at purus. Donec mattis ullamcorper tellus. Phasellus vel nulla. Praesent interdum, eros in sodales sollicitudin, nunc nulla pulvinar justo, a euismod eros sem nec nibh. Nullam sagittis dapibus lectus. Nullam eget ipsum eu tortor lobortis sodales. Etiam purus leo, pretium nec, feugiat non, ullamcorper vel, nibh. Sed vel elit et quam accumsan facilisis. Nunc leo. Suspendisse faucibus lacus.

Pellentesque interdum sapien sed nulla. Proin tincidunt. Aliquam volutpat est vel massa. Sed dolor lacus, imperdiet non, ornare non, commodo eu, neque. Integer pretium semper justo. Proin risus. Nullam id quam. Nam neque. Duis vitae wisi ullamcorper diam congue ultricies. Quisque ligula. Mauris vehicula.

## 3.4 Axiom & Rules in Agda

# 4 Evaluation

## 4.1 Using the System to Reason about Programs

Reasoning about swap was done for constants but a similar proof could be implemented with any expression as long as the variables to be swapped were also swapped across the whole expression. For example:

```
- Swap values of 𝜒 and 𝑦
swap : Program
swap = 𝑧 :=val 𝜒 ;
       𝜒 :=val 𝑦 ;
       𝑦 :=val 𝑧 ;
```

```
< x == y + 1 > $\leftarrow$ sub x for z

z := x (z == y + 1) $\leftarrow$ sub y for x
```

**Figure 4:** ssEvalwithFuel: The small-step evaluation function

```
---------------------------------
ssEvalwithFuel : ℕ → C → S → Maybe S
---------------------------------
-- Skip always terminates successfully even with zero fuel
ssEvalwithFuel zero (skip ;) s = just s
ssEvalwithFuel (suc n) ( skip ;) s = just s
---------------------------------
-- Out of fuel
-- Need to explicitly give all cases here so Agda can see
-- ‘eval zero C = nothing’ is definitionally true when C≠skip
ssEvalwithFuel zero ( WHILE _ DO _ ;) _ = nothing
ssEvalwithFuel zero ( IF _ THEN _ ELSE _ ;) _ = nothing
ssEvalwithFuel zero ( _ := _ ; ) _ = nothing
ssEvalwithFuel zero ((WHILE _ DO _) ; _) _ = nothing
ssEvalwithFuel zero ((IF _ THEN _ ELSE _) ; _) _ = nothing
ssEvalwithFuel zero ((_ := _) ; _) _ = nothing
ssEvalwithFuel zero ( skip ; b ) s = ssEvalwithFuel zero b s
---------------------------------
-- SINGLE WHILE
ssEvalwithFuel (suc n) ( WHILE exp DO c ;) s with evalExp exp s
... | nothing = nothing -- Computation failed i.e.  div by 0
... | f @ (just _) with toTruthValue {f} (Any.just tt)
... | true = ssEvalwithFuel n ( c THEN WHILE exp DO c ;) s
... | false = just s
---------------------------------
-- SINGLE IF THEN ELSE
ssEvalwithFuel (suc n) ( IF exp THEN c₁ ELSE c₂ ;) s
  with evalExp exp s
... | nothing = nothing -- Computation failed i.e.  div by 0
... | f @ (just _) with toTruthValue {f} (Any.just tt)
... | true = ssEvalwithFuel n c₁ s
... | false = ssEvalwithFuel n c₂ s
---------------------------------
```

⋮

**Figure 4:** ssEvalwithFuel cont.

⋮

```
---------------------------------
-- SINGLE ASSI
ssEvalwithFuel (suc n) ( id := exp ;) s =
  map (λ v → updateState id v s) (evalExp exp s)
---------------------------------
-- SKIP ; THEN C
ssEvalwithFuel (suc n) (skip ; c) s = ssEvalwithFuel (suc n) c s
---------------------------------
-- WHILE ; THEN C₂
ssEvalwithFuel (suc n) ((WHILE exp DO c₁) ; c₂) s
  with evalExp exp s
... | nothing = nothing -- Computation failed i.e.  div by 0
... | f @ (just _) with toTruthValue {f} (Any.just tt)
... | true = ssEvalwithFuel n (c₁ THEN ((WHILE exp DO c₁) ; c₂)) s
... | false = ssEvalwithFuel n c₂ s
---------------------------------
-- IF THEN ELSE ; THEN C₂
ssEvalwithFuel (suc n) ((IF exp THEN c₁ ELSE c₂) ; c₃) s
  with evalExp exp s
... | nothing = nothing -- Computation failed i.e.  div by 0
... | f @ (just _) with toTruthValue {f} (Any.just tt)
... | true = ssEvalwithFuel n (c₁ THEN c₃) s
... | false = ssEvalwithFuel n (c₂ THEN c₃) s
---------------------------------
-- ASSI ; THEN C
ssEvalwithFuel (suc n) ((id := exp) ; c) s with evalExp exp s
... | nothing = nothing -- Computation failed i.e.  div by 0
... | (just v) = ssEvalwithFuel n c (updateState id v s)
---------------------------------
```

**Figure 5:** The full proof that evaluation is deterministic via proof that for any two proofs of termination, the resultant states serving as evidence for each of those proofs - in accordance with them being *constructive* proofs - will be identical.
n.b. that the † function is the function that extracts the witness from the proof of termination - i.e. the resultant state after the computation has terminated successfully.

```
--------------------------------------------------------------------------------
EvalDet : ∀ {s f f'} C
                    → (a : ⌊ᵗ f , C , s ᵗ⌋) → (b : ⌊ᵗ f' , C , s ᵗ⌋) → † a ≡ † b
--------------------------------------------------------------------------------
pattern ⇑ x = suc x
EvaluationIsDeterministic = EvalDet
--------------------------------------------------------------------------------
EvalDet {s} {0} {0} (_ ;) ij₁ ij₂ rewrite ∃!IJ ij₁ ij₂ = refl
EvalDet {s} {0} {⇑ _} (skip ;) ij₁ ij₂ rewrite ∃!IJ ij₁ ij₂ = refl
EvalDet {s} {⇑ _} {0} (skip ;) ij₁ ij₂ rewrite ∃!IJ ij₁ ij₂ = refl
EvalDet {s} {⇑ _} {⇑ _} (skip ;) ij₁ ij₂ rewrite ∃!IJ ij₁ ij₂ = refl
EvalDet {s} {⇑ f} {⇑ f'} ((WHILE exp DO c) ;) ij₁ ij₂
  with evalExp exp s
... | cond@(just _) with toTruthValue {cond} (Any.just tt)
... | false rewrite ∃!IJ ij₁ ij₂ = refl
... | true = EvalDet {s} {f} {f'} _ ij₁ ij₂
EvalDet {s} {⇑ f} {⇑ f'} ((IF exp THEN c₁ ELSE c₂) ;) ij₁ ij₂
  with evalExp exp s
... | cond@(just _) with toTruthValue {cond} (Any.just tt)
... | false = EvalDet {s} {f} {f'} _ ij₁ ij₂
... | true = EvalDet {s} {f} {f'} _ ij₁ ij₂
EvalDet {s} {⇑ f} {⇑ f'} ((id := exp) ;) ij₁ ij₂
  with evalExp exp s
... | (just _) rewrite ∃!IJ ij₁ ij₂ = refl
EvalDet {s} {⇑ f} {⇑ f'} ((WHILE exp DO c₁) ; c₂) ij₁ ij₂
  with evalExp exp s
... | cond@(just _) with toTruthValue {cond} (Any.just tt)
... | false = EvalDet {s} {f} {f'} _ ij₁ ij₂
... | true = EvalDet {s} {f} {f'} _ ij₁ ij₂
EvalDet {s} {⇑ f} {⇑ f'} ((IF exp THEN c₁ ELSE c₂) ; c₃) ij₁ ij₂
  with evalExp exp s
... | cond@(just _) with toTruthValue {cond} (Any.just tt)
... | false = EvalDet {s} {f} {f'} _ ij₁ ij₂
... | true = EvalDet {s} {f} {f'} _ ij₁ ij₂
EvalDet {s} {⇑ f} {⇑ f'} ((id := exp) ; c) ij₁ ij₂
  with evalExp exp s
... | (just v) = EvalDet {updateState id v s} {f} {f'} _ ij₁ ij₂
EvalDet {s} {⇑ f} {⇑ f'} (skip ; c) = EvalDet {s} {⇑ f} {⇑ f'} c
EvalDet {s} {0} {0} (skip ; c) ij₁ ij₂ rewrite ∃!IJ ij₁ ij₂ = refl
-- In the clause below, with f = 0 and f' = ⇑ _, the only possibility if we
-- are still to have two proofs of termination in ij₁ and ij₂ is that the
-- rest of the mechanisms in c are all also 'skip'.  So we take each of the two
-- cases of either c = (skip ;) or c = (skip ; ...  ; (skip ;)) in turn.
-- Annoyingly we have to do this for both permutations of f/f' = 0/⇑ _
EvalDet {s} {0} {⇑ f'} (skip ; (skip ;)) ij₁ ij₂ rewrite ∃!IJ ij₁ ij₂ = refl
EvalDet {s} {0} {⇑ f'} (skip ; (skip ; c)) = EvalDet {s} {0} {⇑ f'} c
EvalDet {s} {⇑ f} {0} (skip ; (skip ;)) ij₁ ij₂ rewrite ∃!IJ ij₁ ij₂ = refl
EvalDet {s} {⇑ f} {0} (skip ; (skip ; c)) = EvalDet {s} {⇑ f} {0} c
--------------------------------------------------------------------------------
```

**Figure 6:** [t]-split: The function for splitting two proofs of termination.

```
-----------------------------------------------------------
⌊ᵗ⌋-split' : ∀ f s Q₁ Q₂ → (t₁₂ : ⌊ᵗ f , Q₁ THEN Q₂ , s ᵗ⌋)
              → Σ ⌊ᵗ f , Q₁ , s ᵗ⌋ (λ t₁
              → Σ ℕ (λ f'
              → f' ≤″ f × Σ ⌊ᵗ f' , Q₂ , † t₁ ᵗ⌋ (λ t₂
              → † t₂ ≡ † t₁₂ )))
-----------------------------------------------------------


-----------------------------------
- Base case:  Q₁ = skip ;
⌊ᵗ⌋-split' f@0 s (skip ;) Q₂ t₁₂ =
  (Any.just tt) , f , ≤with refl            , t₁₂ , refl
⌊ᵗ⌋-split' f@(suc _) s (skip ;) Q₂ t₁₂ =
  (Any.just tt) , f , ≤with (+-comm f 0) , t₁₂ , refl
- Q₁ = skip : Q₁ '
⌊ᵗ⌋-split' f@0 s (skip ; Q₁ ') = ⌊ᵗ⌋-split' f s Q₁ '
⌊ᵗ⌋-split' f@(suc _) s (skip ; Q₁ ') = ⌊ᵗ⌋-split' f s Q₁ '



-----------------------------------
- Most interesting inductive case:  WHILE followed by Q₁' THEN Q₂.
- All other cases follow a similar recursive mechanism
⌊ᵗ⌋-split' (suc f) s Q₁@((WHILE exp DO c) ; Q₁ ') Q₂ t₁₂ = go
  where
  go : Σ ⌊ᵗ suc f , Q₁ , s ᵗ⌋ (λ t₁ → Σ ℕ (λ f' → f' ≤″ suc f ×
       Σ ⌊ᵗ f' , Q₂ , † t₁ ᵗ⌋ (λ t₂ → † t₂ ≡ † t₁₂ )))
  go with evalExp exp s
  go | f@(just _) with toTruthValue {f} (Any.just tt)
  - if false --------------------
  go | f@(just _) | false with ⌊ᵗ⌋-split' f s Q₁ ' Q₂ t₁₂
  go | just _ | false |   t₁ , f ' , lt , t₂ , Δ
                        = t₁ , f ' , suc≤″ lt , t₂ , Δ
  - if true --------------------
  go | f@(just _) | true rewrite
       THEN-comm ((WHILE exp DO c) ;) Q₁ ' Q₂
     | THEN-comm c ((WHILE exp DO c) ; Q₁ ') Q₂ with
       ⌊ᵗ⌋-split' f s (c THEN WHILE exp DO c ; Q₁ ') Q₂ t₁₂
  go | f@(just _) | true |   t₁ , f ' , lt , t₂ , Δ
                           = t₁ , f ' , suc≤″ lt , t₂ , Δ

                                    ⋮
```

**Figure 6:** [t]-split cont.
n.b. some cases have been omitted but none that vary from the general pattern here.

⋮

```
------------------------------------
- Q₁ = if then else
⌊ᵗ⌋-split' (suc f) s Q₁@((IF exp THEN c₁ ELSE c₂) ;) Q₂ t₁₂
  = go
  where
  go : Σ ⌊ᵗ suc f , Q₁ , s ᵗ⌋ (λ t₁ → Σ ℕ (λ f' → f' ≤" suc f ×
         Σ ⌊ᵗ f' , Q₂ , † t₁ ᵗ⌋ (λ t₂ → † t₂ ≡ † t₁₂ )))
  go with evalExp exp s
  go | f@(just _) with toTruthValue {f} (Any.just tt)
  - if false --------------------
  go | f@(just _) | false with ⌊ᵗ⌋-split' f s c₂ Q₂ t₁₂
  go | f@(just _) | false |   t₁ , f' , lt , t₂ , Δ
                          = t₁ , f' , suc≤" lt , t₂ , Δ
  - if true --------------------
  go | f@(just _) | true with ⌊ᵗ⌋-split' f s c₁ Q₂ t₁₂
  go | f@(just _) | true |   t₁ , f' , lt , t₂ , Δ
                          = t₁ , f' , suc≤" lt , t₂ , Δ
------------------------------------
- Q₁ = x := exp ; Q₁'
⌊ᵗ⌋-split' (suc f) s Q₁@( id := exp ; Q₁') Q₂ t₁₂ = go
  where
  go : Σ ⌊ᵗ suc f , Q₁ , s ᵗ⌋ (λ t₁ → Σ ℕ (λ f' → f' ≤" suc f ×
         Σ ⌊ᵗ f' , Q₂ , † t₁ ᵗ⌋ (λ t₂ → † t₂ ≡ † t₁₂ )))
  go with evalExp exp s
  go | f@(just v)
    with ⌊ᵗ⌋-split' f (updateState id v s) Q₁' Q₂ t₁₂
  go | f@(just v) |   t₁ , f' , lt , t₂ , Δ
                  = t₁ , f' , suc≤" lt , t₂ , Δ
------------------------------------
- Q₁ = id := exp ;
⌊ᵗ⌋-split' (suc f) s Q₁@( id := exp ;) Q₂ t₁₂ = go
  where
  go : Σ ⌊ᵗ suc f , Q₁ , s ᵗ⌋ (λ t₁ → Σ ℕ (λ f' → f' ≤" suc f ×
         Σ ⌊ᵗ f' , Q₂ , † t₁ ᵗ⌋ (λ t₂ → † t₂ ≡ † t₁₂ )))
  go with evalExp exp s
  ... | f@(just _)
    = (Any.just tt) , f , ≤with (+-comm f 1) , t₁₂ , refl
------------------------------------
```

**Figure 7:** SWAP: Using the library to formalise the correctness of the SWAP program:

```
SWAP : ∀ X Y →
                    « x == (const X) ∧ y == (const Y) » - Precondition

                              z := val x ;
                              x := val y ;
                              y := val z ;

                    « x == (const Y) ∧ y == (const X) » - Postcondition
SWAP X Y = ■
  where

  - Reasoning backwards from Postcondition Q to Precondition P

  PRE : Assertion
  PRE = x == (const X) ∧ y == (const Y)

  POST : Assertion
  POST = x == (const Y) ∧ y == (const X)

  A₁ : Assertion
  A₁ = ((sub (val z) y (val x)) == (const Y)) ∧ ( z == (const X))

  s₁ : « A₁ » y := val z ; « POST »
  s₁ = let Ψ = D0-Axiom-of-Assignment y (val z) POST in go Ψ
       where
       go : « ((sub (val z) y (val x)) == (const Y))
            ∧ ((sub (val z) y (val y)) == (const X)) »
            y := val z ; « POST » →
            « A₁ » y := val z ; « POST »
       go t with y ?id= x
       go t | yes p rewrite p with x ?id= x
       go t | yes p | yes q = t
       go t | yes p | no ¬q = ⊥-elim (¬q refl)
       go t | no ¬p with y ?id= y
       go t | no ¬p | yes q = t
       go t | no ¬p | no ¬q = ⊥-elim (¬q refl)

                                  ⋮
```

**Figure 7:** SWAP: Using the library to formalise the correctness of the SWAP program: cont.

⋮

```
A₂ : Assertion
A₂ = ((sub (val y) x (sub (val z) y (val x))) == (const Y)) ∧ ( z == (const X))

s₂ : « A₂ » x := val y ; « A₁ »
s₂ = let Ψ = D0-Axiom-of-Assignment x (val y) A₁ in go Ψ
     where
     go : « ((sub (val y) x (sub (val z) y (val x))) == (const Y))
          ∧ ((sub (val y) x (val z)) == (const X)) »
          x := val y ; « A₁ » →
          « A₂ » x := val y ; « A₁ »
     go t with x ?id= z
     go t | yes p = ⊥-elim (x≢z p)
     go t | no _ = t

A₃ : Assertion
A₃ = ((sub (val x) z (sub (val y) x (sub (val z) y (val x)))) == (const Y))
     ∧ ( x == (const X) )

s₃ : « A₃ » z := val x ; « A₂ »
s₃ = let Ψ = D0-Axiom-of-Assignment z (val x) A₂ in go Ψ
     where
     go : « ((sub (val x) z (sub (val y) x (sub (val z) y (val x)))) == (const Y))
          ∧ ((sub (val x) z (val z)) == (const X)) »
          z := val x ; « A₂ » →
          « A₃ » z := val x ; « A₂ »
     go t with z ?id= z
     go t | yes _ = t
     go t | no ¬p = ⊥-elim (¬p refl)
```

⋮

**Figure 7:** SWAP: Using the library to formalise the correctness of the SWAP program: cont.

⋮

```
s₄ : A₃ ≡ ( y == (const Y) ∧ x == (const X) )
s₄ with y ?id= x
s₄ | yes _ with x ?id= z
s₄ | yes _ | yes q = ⊥-elim (x≢z q)
s₄ | yes _ | no _ with z ?id= z
s₄ | yes p | no _ | yes _ rewrite p = refl
s₄ | yes _ | no _ | no w = ⊥-elim (w refl)
s₄ | no ¬p with x ?id= x
s₄ | no _ | no ¬q = ⊥-elim (¬q refl)
s₄ | no _ | yes _ with z ?id= y
s₄ | no _ | yes _ | yes w = ⊥-elim (y≢z (sym w))
s₄ | no _ | yes _ | no _ = refl
s₅ : « A₂ » x := val y ;  y := val z ; « POST »
s₅ = D2-Rule-of-Composition {A₂} {A₁} {POST} s₂ s₁


s₆ : « A₃ » z := val x ;  x := val y ; y := val z ; « POST »
s₆ = D2-Rule-of-Composition {A₃} {A₂} {POST} s₃ s₅

■ : « PRE » z := val x ; x := val y ; y := val z ; « POST »
■ = D1-Rule-of-Consequence-pre {A₃} {swap} {POST} {PRE} s₆ go
      where
      go : PRE ⇒ A₃
      go s x rewrite ConjunctionComm
                          (evalExp (x == const X) s )
                          (evalExp (y == const Y) s )
           = subst (λ p → s ⊨ p ) (sym s₄) x
```

```
x := y (z == x + 1) $\leftarrow$ sub z for y

y := z <y == x + 1 >
```

Some maths: $\ll \mathtt{P} \gg \mathtt{C} \ll \mathtt{Q} \gg$ test test hello hello

## 4.2 Using Agda

Getting better at working with Agda — thanks to unicode suport, psychological bias of aesthetic but incorrect signature.

'I believe the hard part of building software to be the specification, design, and testing of this conceptual construct, not the labor of representing it and testing the fidelity of the representation.' - Fred Brooks who evidently, never used Agda.

## 4.3 Missteps

Downsides: Having to define all logical manipulations in the interface. Some mechanism for making this less painful would be nice. Equally however, said logical manipulations are not really of the main concern. Just because nothing can be postulated for the proof to have a computational meaning, doesn't mean we need be bound by this restriction. Indeed, there is little reason for us to care about whether or not our proofs have this computational context so long as we trust the parts we omit, and as these ommisions are oft simple logical manipulations, leaving them only in the record but not actually proved would be perfectly sensible and allow more focus to be on the manipulation of Hoare triples to reason about programs.

Actually, if I was doing it again, it would have been very sensible to not bother with implementing the interfaces at all. It was probably not a worthwhile use of my time to prove De Morgans law, or the commutativity of boolean and, in Agda when I could have instead focused on the more salient parts of the code base.

With more time a more expansive interface would be given allowing for as many identifiers as were necessary to reason about the desired program along with a mechanism for obtaining free identifiers from an expression, with the identifiers being represented as natural numbers, a new *free* identifier could always be generated by summing the numerical value of the identifiers present in the supplied expression, or in a given list.

In hindsight, it would have been prudent to abstract away whole expression language, not just the data/values. (page 42 surface properties (Ligler))

## 4.4 Future Work

Gries page 164 'a fine balance between the two' . . . but! automation, Infer,

parse a C program and create formal proof in background. Complain if fail

If the Agda code is to work as a library, there had ought to be some functionality for allowing potential users to add to Data-Implementation.agda, to define their own logical rules.

Ref paper: tactics for separation logic and how some reworking of data-interface could allow full use of HOL in Agda when manipulating assertions.

## 4.5 Conclusion

Hoare's surprise at test case success (see retrospective)

Not all that useful in practice, other tools are far more sophisticated and far better suited to practical applications, whether that be the verified software toolchain for reasoning about embedded software that needs to be correct, or Infer for catching a litany of bugs before they make their way into production. There's not a lot of room to claim that this Agda library has any real practical purpose. But that doesn't mean no purpose.

Constructive mathematics and interactive theorem provers are not front and center in mainstream mathematics, and perhaps never will be, but one oft talked about benefits of constructive mathematics is quite simply that it is fun to do [link MHE blog post] and it is on that note that one finds the most compelling use for this Agda library; it's actually a lot of fun — if you're of a particular sort — to reason about even the most simple of programs. I certainly found it enjoyable to reason about the swap program and have Agda check my workings for me.

Never been so intimately aquainted with the three lines of code comprising the swap function.

## 5 Appendix

Wrting scrap pile: (TO BE DELETED BEFORE SUBMISSION)

The wording here can be a little confusing. In accordance with the literature, the term 'weakest precondition' could reasonably be referring to one of three things:

1. The weakest precondition for an unspecified state $\mathcal{S}$, *and* an unspecified postcondition $\mathcal{R}$. Denoted: $wp(\mathcal{S}, \mathcal{R})$.
2. The *function* of type $\texttt{Postcondition} \to \texttt{Precondition}$ for a fixed command/mechanism $\mathcal{S}$, say the $\texttt{skip}$ command, that takes an unspecified postcondition $\mathcal{R}$. Denoted: $wp(\ \texttt{skip}\ , \mathcal{R})$.
3. Or, the actual *output* of the function, that is, the predicate itself, for a given command $\mathcal{S}$ and a given postcondition $\mathcal{R}$. Also denoted: $wp(\mathcal{S}, \mathcal{R})$.

Note that in the text [dijkstra], $wp(S, R)$, is used interchangeably as a predicate and as the state space that said predicate captures. With our constructive formalisation however, this lack of precision is not possible nor desired, so we end up with the, perhaps superfluous, distinction between predicates and the state space that they describe. Meaning that under our formalisation, $wp(S, F)$ is empty when considered as a state space, but inhabited when considered as a predicate (inhabited uniquely by $F$ itself). This exposition also explains why $«F» S «Q»$ is an inhabited type, as $F$ *is* a valid precondition of any computation for any postcondition (think absurd function, or bluff function). << Actually explained by the fact that what we have formalised is the weakest *liberal* precodnition!

Weakest Liberal Precondition is what has actually been formalised! Total correctness is denoted by $[\![P]\!]\, S\, [\![Q]\!]$

7 regions of the statesapce. As such, we can — if we wish — give a semantics to the notion of a derterministic mechanism as one in which the last four regions of the state space are empty.

As a *deductive system* Hoare logic is only interested in the preservation of truth: treating not WFF as false.

Carving up state space. Every predicate denotes a subset of the statespace (which in our case is infinite).

(day = 23) Dijkstra's example

T/F, x == 2

Relationship between logical operators and set theoretic operators i.e. $\wedge \Leftrightarrow \cap$

$$\mathit{even}\langle\_\rangle : \mathsf{Exp} \to \mathsf{Exp}$$
$$\mathit{even}\langle\ P\ \rangle = \mathsf{op}_2\ (\mathsf{op}_2\ P\ \%_o\ (\mathsf{const}\ ②))\ ==_o\ (\mathsf{const}\ ⓪)$$

Ought to have differentiated between non stuck-ness and termination. I.e. D(E) as domain of expression E, to eliminate divide by zero and non-defined variables in an expression, as that is a problem that can be handled distinctly from termination (i.e. (I think anway) that given a state S, and an expression E, one can deterministically/decidably determine whether or not it is a WFF).

Donec ullamcorper, felis non sodales.. Lorem ipsum dolor sit amet, consectetuer adipiscing elit.

## References

[1] Andrew W. Appel. Verified software toolchain. In Gilles Barthe, editor, *Programming Languages and Systems*, pages 1–17, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[2] Edsger W. Dijkstra. Some meditations on advanced programming. In *Information Processing, Proceedings of the 2nd IFIP Congress 1962, Munich, Germany, August 27 - September 1, 1962*, pages 535–538. North-Holland, 1962.

[3] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, August 1975.

[4] Edsger W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[5] Robert W. Floyd. Assigning meanings to programs. *Proceedings of Symposium on Applied Mathematics*, 19:19–32, 1967.

[6] David Gries. *The Science of Programming.* Texts and Monographs in Computer Science. Springer, 1981.

[7] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[8] Charles Antony Richard Hoare. Viewpoint retrospective: An axiomatic basis for computer programming. *Communications of the ACM*, 52(10):30–32, 2009.

[9] Donald E. Knuth and Peter B. Bendix. Simple word problems in universal algebras. In John Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon, 1970.

[10] George T. Ligler. A mathematical approach to language design. In *Proceedings of the 2nd ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '75, page 41–53, New York, NY, USA, 1975. Association for Computing Machinery.

[11] George T. Ligler. The assignment axiom and programming language design. In *Proceedings of the 1976 Annual Conference*, ACM '76, page 2–6, New York, NY, USA, 1976. Association for Computing Machinery.

[12] Ulf Norell. *Towards a practical programming language based on dependent type theory*, volume 32. Citeseer, 2007.

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