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

Project Report
Word Count: 10,000
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

Introduction

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 separate 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 a constructive formalisation of Hoare logic and an Agda library that is fit for the purpose of 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 **3**
- 1.1 Programming Language Semantics . . . 3
  - 1.1.1 Axiomatic Semantics via Hoare Logic . . . 4
  - 1.1.2 Predicate Transformer Semantics via Dijkstra's *Weakest Precondition* . . . 8
- 1.2 Agda as an Interactive Theorem Prover . . . 11
  - 1.2.1 Formal Proof . . . 11
  - 1.2.2 Constructive Mathematics . . . 11
  - 1.2.3 Interactive Theorem Provers . . . 13
- 1.3 Modern Literature Review . . . 15

2 Specification of the Formalisation **16**
- 2.1 Shallow vs. Deep Embedding . . . 16
- 2.2 Proof Obligation Interfaces . . . 20
- 2.3 The Exppresion and/or Assertion Language . . . 22
- 2.4 The 'Mini-Imp' Programming Language . . . 25
- 2.5 The Rules to be Implemented . . . 26

3 Implementation Details **28**
- 3.1 Small-step Evaluation & Termination . . . 28
- 3.2 Termination Proof Splitting . . . 30
- 3.3 Hoare Triples in Agda . . . 30
- 3.4 Axiom of Assignment . . . 31
- 3.5 The Rule of Iteration / While Rule . . . 31
- 3.6 Relation to Predicate Transformer Semantics . . . 32

4 Project Evaluation **36**
- 4.1 Using the System to Reason about Programs . . . 36
- 4.2 Deliverables . . . 36
- 4.3 Reflection on the use of Agda . . . 37
- 4.4 Missteps & Drawbacks . . . 37
- 4.5 Future Work . . . 38
- 4.6 Conclusion & Personal Reflection . . . 38

References **39**

Appendix **40**

1 Preliminaries & Literature Review

1.1 Programming Language Semantics

"Programming began as an art"

- David Gries, *The Science of Programming*

Around the late 60s – early 70s, 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 elegant languages 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 programs 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 elegance 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 attachment 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 c of one of these types, a *verification condition* $\mathcal{V}_c(\mathcal{P}; \mathcal{Q})$ on the antecedents and consequents of 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 elegance 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:

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

...but both of these are erroneous. The first gives the false consequent $\vdash \{\!\{ \ x = 1 \ \}\!\} \ x := 0 \ \{\!\{ \ x = 1 \ \}\!\}$ as a direct consequence of the fact that $(x = 1)[0/(x)] \ = \ (x = 1)$; as 0 doesn't occur in '$x = 1$'. The second gives the false consequent of $\vdash \{\!\{ \ x = y \ \}\!\} \ x := z \ \{\!\{ \ z = y \ \}\!\}$ via substituting x 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 precondition/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 figure 3

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

D1 - Rules of Consequence: (2)

$$\text{If} \quad \vdash \{\!\{ \mathcal{P} \}\!\} \, \mathcal{Q} \, \{\!\{ \mathcal{R} \}\!\} \quad \text{and} \quad \vdash \mathcal{R} \Rightarrow \mathcal{S} \quad \text{then} \quad \vdash \{\!\{ \mathcal{P} \}\!\} \, \mathcal{Q} \, \{\!\{ \mathcal{S} \}\!\}$$
$$\text{If} \quad \vdash \{\!\{ \mathcal{P} \}\!\} \, \mathcal{Q} \, \{\!\{ \mathcal{R} \}\!\} \quad \text{and} \quad \vdash \mathcal{S} \Rightarrow \mathcal{P} \quad \text{then} \quad \vdash \{\!\{ \mathcal{S} \}\!\} \, \mathcal{Q} \, \{\!\{ \mathcal{R} \}\!\}$$

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.

D2 - Rule of Composition: (3)

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

Finally the most interesting rule, the rule of iteration:

D3 - Rule of Iteration: (4)

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

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 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 industry and academics — or rather, the theoretically inclined academics — as the 60s 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 $\mathcal{P} \Rightarrow wp(\mathcal{S}, \mathcal{R})$.

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

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 groundwork 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:

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

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

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

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 towards 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 unambiguous 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 *axiom* 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 algorithmically 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 proceed 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. An early demonstration of the use of a computer to aid mathematical endeavour is given by Knuth and Bendix in 1970[8] 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 development of mathematics with computational aid has come a long way since then and there are now a plethora of so called 'Interactive Theorem Provers' 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 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 χ" where χ is some other natural number.

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

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

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. In addition to the above, 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[9] is the interactive theorem prover used within this formalisation, chosen as it is 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 reduction rule but at the cost of our proof having algorithmic meaning, because, as was mentioned in the previous section, PEM has no computational meaning. So this restriction is what allows Agda to also claim itself a *programming language* and operate under the *propositions as types* paradigm, also known as the Curry-Howard correspondence, in which the type of a program as described by its signature becomes a proposition and an implementation of that program becomes a proof of that proposition. Disciplined use of Agda in this way showcases the fact that the formalisation is possible without LEM/PEM, an unsurprising, but nice to verify, fact.

[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 despite the methodology 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 — likely as a result of programs continuing to swell in complexity. Hoare logic has been expanded upon, giving rise most notably to *Separation Logic* which has become a success story of the theoretical world after making its way into industry in the form of the *Infer* tool which is now in use in a plethora of tech companies. The Infer tool is a static analyser for Java, C, C++, and other languages, capable of catching a plethora of bugs including null pointer errors and memory leaks before they make their way into production.

Separation logic originated from a couple of papers ([10]/[11]) in which Hoare logic was extended 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 and reasoning 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 on the modern scene 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.

An interesting thing to note is that the two systems above, Infer and VST, work via the 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 diminishes the first constraint by constraining the programmer from constructing all possible programs.

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 regard 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 to apply to needs to be constructed and formalised first. This language itself, will actually comprise *two* languages, the language defining the commands of the language (`IF_ELSE_` etc...) and the expression and or assertion language defining both the expressions that appear *within* those commands and the assertions for the logical calculus.

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 implemented 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 provides 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 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.

This project, then, demands a choice of embedding strategy 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 justification for the imperative language to be a deep embedding comes from the recognition that rather than just formalising Hoare logic in Agda, this project could also have practical utility in forming 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 of the system lose confidence that the program proved correct in Agda accurately captures the one they started with.

Another 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 embed 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 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 embedded language itself.

Despite this drawback, once again the desire for this project to not only be a theoretical achievement but also to have potential practical use 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 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. He also pointed to the fact that the actual arithmetic or overflow strategy could be identified via a set of mutually exclusive axioms — a fact made use of in the *proof obligation interfaces* described in the next section.

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 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 calculus 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 and or 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**: to abstract out the representation of the identifiers, values, and operations and lemmas thereupon.
- **State-Interface**: to abstract out the representation of the state space and lemmas thereupon.

The intention of these interfaces would be twofold. First, to allow the user to forestall, perhaps indefinitely, the obligation of proving simple or obvious lemmas when proving a program correct within the library. 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 prove programs correct.

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:*		
_||_	:	...	DeMorgan_1	:	...
_&&_	:	...	DeMorgan_2	:	...
⋮			$\text{ConjunctionElim}_{left}$	:	...
_+_	:	...	⋮		
_*_	:	...	NegationElim	:	...

STATE-INTERFACE:

definition of state space:

$\mathcal{S}$: Set

empty/initial state:

● : $\mathcal{S}$

state operations:

updateState : Id → Val → $\mathcal{S}$ → $\mathcal{S}$

getIdVal : Id → $\mathcal{S}$ → Maybe Val

dropValue : Id → $\mathcal{S}$ → $\mathcal{S}$

state space lemmas as needed:

⋮

NB that the interfaces have been instantiated in full as part of this project with ID $\stackrel{\text{def}}{=}$ ℕ and VAL $\stackrel{\text{def}}{=}$ (ℤ x BOOL) — where ℕ, ℤ, and 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 defined as lists of pairs of ID's and VAL's — i.e. $\mathcal{S} \stackrel{\text{def}}{=}$ (LIST (ID x VAL)).

2.3 The Expression 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:

```
<Exp>      ::= <Exp> <Op₂> <Exp> | <Op₁> <Exp> | <terminal>
<Op₂>      ::= &&ₒ | ||ₒ | ==ₒ | ≤ₒ | ... | +ₒ | -ₒ | ... | %ₒ
<Op₁>      ::= ++ₒ | -ₒ | ¬ₒ | -ₒ
<terminal> ::= const <num> | var <id> | true | false
<num>      ::= 1 | 2 | 3 | ...
<id>       ::= x | y | z | ...
```

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 immediately 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:

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

...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 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 '$(\mathtt{x} == (\mathtt{y}\ /\ 0)) \stackrel{\text{def}}{=} \mathit{false}$' 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
private a₁ : Assertion
a₁ = ((val χ) == (const ②))
     ∧
     ((val y) == (const ①))

a₂ ≝ χ == 2
private a₂ : Assertion
a₂ = (val χ) == (const ②)

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 language 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):

$$\begin{array}{l}\text{D1-Rule-of-Consequence-post} : \forall\ \{P\}\ \{Q\}\ \{R\}\ \{S\} \\ \qquad \rightarrow\ \ll\ P\ \gg\ Q\ \ll\ R\ \gg \rightarrow R \Rightarrow S \\ \text{--}\ \text{----------------------------}\ \text{--} \\ \qquad \rightarrow\ \ll\ P\ \gg\ Q\ \ll\ S\ \gg\end{array}$$

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

$$\begin{array}{l}\text{D2-Rule-of-Composition} : \forall\ \{P\}\ \{R_1\}\ \{R\}\ \{Q_1\}\ \{Q_2\} \\ \qquad \rightarrow\ \ll\ P\ \gg\ Q_1\ \ll\ R_1\ \gg \rightarrow\ \ll\ R_1\ \gg\ Q_2\ \ll\ R\ \gg \\ \text{--}\ \text{----------------------------------}\ \text{--} \\ \qquad \rightarrow\ \ll\ P\ \gg\ Q_1\ \texttt{THEN}\ Q_2\ \ll\ R\ \gg\end{array}$$

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

$$\begin{array}{l}\text{D3-While-Rule} : \forall\ \{P\}\ \{B\}\ \{C\} \\ \qquad \rightarrow\ \ll\ P \wedge B\ \gg\ C\ \ll\ P\ \gg \\ \text{--}\ \text{----------------------------}\ \text{--} \\ \qquad \rightarrow\ \ll\ P\ \gg\ \texttt{WHILE}\ B\ \texttt{DO}\ C\ \texttt{;}\ \ll\ (\mathit{not}\ B) \wedge P\ \gg\end{array}$$

$$\begin{array}{l}\text{D4-Conditional-Rule} : \forall\ \{A\}\ \{B\}\ \{C\}\ \{P\}\ \{Q\} \\ \qquad \rightarrow\ \ll\ C \wedge P\ \gg\ A\ \ll\ Q\ \gg \rightarrow\ \ll\ (\mathit{not}\ C) \wedge P\ \gg\ B\ \ll\ Q\ \gg \\ \text{--}\ \text{----------------------------------}\ \text{--} \\ \qquad \rightarrow\ \ll\ P\ \gg\ \texttt{IF}\ C\ \texttt{THEN}\ A\ \texttt{ELSE}\ B\ \texttt{;}\ \ll\ Q\ \gg\end{array}$$

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

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

3 Implementation Details

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 Small-step Evaluation & 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. Said 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 : $\mathbb{N} \to$ Program $\to$ S $\to$ Maybe S. The implementation of this function is straightforward, if a little verbose, with the two most interesting cases reproduced below and the full implementation given in figure 4 in the appendix.

```
-------------------------------------------------------
-- 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 a small-step evaluation function defined, what it means for a program to terminate can now be formalised like so:[18]

```
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
```

The different notations above are just different ways of saying the same thing, viz, that a program terminates if there exists some $n \in \mathbb{N}$ such that the evaluation of the program succeeds with fuel $= n$. With this constructive definition of termination, some useful lemmas for later use in formalising the Hoare logic rules were proved:

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

The above function relates any two proofs of termination of the same program from the same initial state via identifying the resultant states. This can be taken as a proof that evaluation is deterministic — which it obviously is — hence the name EvalDet.[19] Closely related is the addFuel function below that takes any proof of termination and generates a new proof of termination with a given extra amount of fuel. The implementation of both of these lemmas is fairly straightforward albeit tedious, with the central mechanism being induction on the structure of C. Both lemmas are used in the constructive proof of the D3-While-Rule (see figure 6).

```
addFuel : ∀ {C} {s} f f' → ⌊t f , C , s t⌋ → ⌊t (f +N f') , C , s t⌋
```

[18] NB That C is introduced here as a type synonym for Program.

[19] The implementation/proof of this function is given in figure 7 in the Appendix.

3.2 Termination Proof Splitting

On the topic of termination, a key lemma that is utilised within the proof of both the D3-While-Rule and the D2-Rule-of-Composition is the mechanism $\lfloor^t\rfloor$-split[20] that takes a proof of termination of some program of the form 'Q_1 `THEN` Q_2' and outputs the following Split-$\lfloor^t\rfloor$ record:

```
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
```

That is, it 'splits' a proof of termination into two proofs of termination of the two constituent parts. It is only the Δ component that is necessary within the rule of composition as a means of identifying the resultant state of $\ll R_1 \gg Q_2 \ll R \gg$ with the resultant state of $\ll P \gg Q_1$ `THEN` $Q_2 \ll R \gg$. On the other hand, for the while rule, the full record is needed.

3.3 Hoare Triples in Agda

With termination covered, it is now possible to give the definition Hoare triples within the Agda library. The choice of notation is a result of the curly braces being reserved for Agda's syntax and so $\ll P \gg Q \ll R \gg$ and $[\![P]\!]\, Q\, [\![R]\!]$ are used for partial and total correctness respectively.

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

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

[20] The implementation/proof of this function is given in figure 8 in the Appendix.

3.4 Axiom of Assignment

Rather than go into detail of the proofs of each and every rule, only the proofs of D0-Axiom-of-Assignment and D3-While-Rule are detailed in this report, as these are the most salient rules.

The formulation of the axiom of assignment first requires the construction of a function that substitutes an expression into an assertion but given that assertions are synonymous with expressions here, this function has signature: sub : Exp → Id → Exp → Exp. The proof of D0-Axiom-of-Assignment, then, isn't too complicated, with the key mechanisms within the proof being two lemmas that were added to the state interface:

- updateGet :

$$\forall\ i\ v\ s \rightarrow \mathsf{getIdVal}\ i\ (\mathsf{updateState}\ i\ v\ s) \equiv \mathsf{just}\ v$$

- ignoreTop :

$$\forall\ i\ v\ x \rightarrow \neg\ i \equiv x \rightarrow (s : \mathsf{S}) \rightarrow \mathsf{getIdVal}\ x\ (\mathsf{updateState}\ i\ v\ s) \equiv \mathsf{getVarVal}\ x\ s$$

That is, any state space implementation must satisfy the above, fairly intuitive, lemmas for the axiom of assignment to hold. Both have been implemented within the 'state as list' representation as part of this formalisation.

3.5 The Rule of Iteration / While Rule

The proof/implementation of D3-While-Rule is a little more involved, but the basic mechanism is similar to that of the proofs relating to termination/evaluation of programs, with induction being performed on the structure of a Program. The most salient mechanism as mentioned in the previous section is the use of the $\lfloor^t\rfloor$-split function. This function is needed for the case where the condition B of the while loop is true and therefore the body of the loop is evaluated. Under this case, the (assumed) proof of termination reduces to: $\lfloor^t\ f\ ,\ (C\ \texttt{THEN WHILE}\ B\ \texttt{DO}\ C\ \texttt{;})\ ,\ s\ ^t\rfloor$ but for the recursive call a proof of termination of the form: $\lfloor^t\ f\ ,\ (\texttt{WHILE}\ B\ \texttt{DO}\ C\ \texttt{;})\ ,\ s\ ^t\rfloor$ is needed. Obviously it is the case that if the former terminates with f fuel then so will the latter and so the $\lfloor^t\rfloor$-split function, along with addFuel, and EvalDet, is used to transform the proof of the former into a proof of the latter.[21]

[21] NB That the full implementations/proofs of the axiom of assignment and the While Rule are reproduced in the appendix in figures 5, and 6 respectively.

3.6 Relation to Predicate Transformer Semantics

In [4] Dijkstra outlines some properties that the notion of *wp* and *wlp* must satisfy to make sense as a means of giving semantics to a mechanism and or programming construct. Failure to satisfy these properties would mean we were no longer manipulating pre/post-conditions but instead just 'massaging predicates.'

These properties are proved classically in that exposition but given the scope of this project, it seemed natural to consider including these properties in the formalisation as a means of sanity checking the formalisations of the Mini-Imp mechanisms that have been defined.

However, the constructs/mechanisms that have been formalised here ($:=$, `IF_THEN_ELSE_`, `WHILE_DO_` ...) have been formalised in terms of how they are to be executed, which is precisely the approach to defining programming constructs that the notions of *wp* and *wlp* were trying to avoid. So there is an incongruency there.

The only mechanism for which *wp*/*wlp* have come close to being formalised is the assignment ($:=$) mechanism via the sub function. This is because the sub function actually *is* the weakest precondition.

$$\text{i.e. } wp(i := e\ ,\ \mathcal{R}\ ) = \mathsf{sub}\ e\ i\ \mathcal{R}$$

With that said, while the sub function has been formalised, the fact that it *is* the weakest precondition hasn't been. What has been formalised is the fact that $\mathsf{sub}\ e\ i\ \mathcal{R} \Rightarrow wp(\ i := e\ ,\ \mathcal{R}\ )$, via the proof of the axiom of assignment.

For the rest of the mechanisms, however, no attempt has been made to formalise their corresponding *wp*/*wlp*. Nonetheless, it may be within reach to confirm the *wp*-properties for just the *wp* that has been defined, viz the sub function. This proves to be trivial for property 1, the so called 'Law of the Excluded Miracle' ($wp(\mathcal{S},F) = F$) ...

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

... and trivial for the second property of monotonicity, with the proof not reproduced here. In fact, after this second property was confirmed it became clear that this was a detour from the project scope that was adding little value to the formalisation so the exploration of the formalisation's relationship to predicate transformer semantics stopped there.

Figure 3: Using the library to formalise the correctness of the `SWAP` program: `[1/3]`

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

                        z := val χ ;
                        χ := val y ;
                        y := val z ;

                « χ == (const Y) ∧ y == (const X) » - Postcondition
SWAP X Y = ■
  where
  - - Reasoning backwards from Postcondition Q to Precondition P

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

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

  A₁ : Assertion
  A₁ = ((sub (val z) y (val χ)) == (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 χ)) == (const Y))
            ∧ ((sub (val z) y (val y)) == (const X)) »
            y := val z ; « POST » →
            « A₁ » y := val z ; « POST »
       go t with y ?id= χ
       go t | yes p rewrite p with χ ?id= χ
       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 3: Using the library to formalise the correctness of `SWAP` program cont. [2/3]

⋮

```
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 3: Using the library to formalise the correctness of `SWAP` program cont. [3/3]

```
⋮

s₄ : A₃ ≡ ( y == (const X) ∧ 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
```

4 Project Evaluation

4.1 Using the System to Reason about Programs

With the Hoare logic rules implemented, proofs of program correctness can now be constructed using the Agda library that has been developed.

Only a simple example has been constructed, a proof of correctness of the `SWAP` program, of the form shown below. See figure 3 for the actual proof.

$$\ll x == (const\ X) \wedge y == (const\ Y) \gg \texttt{ - Precondition}$$

$$\begin{aligned} z &:= val\ x\ ; \\ x &:= val\ y\ ; \\ y &:= val\ z\ ; \end{aligned}$$

$$\ll x == (const\ Y) \wedge y == (const\ X) \gg \texttt{ - Postcondition}$$

While this is a very simple and meagre example that doesn't even utilise the While Rule, it is sufficient for evaluating the use of the Agda library that has been produced for the task of formalising proofs of correctness of simple programs. The process, ultimately, even for incredibly simple programs, is rather tedious as it forces the user to think very carefully and belabour with great assiduity. In the end, more time is spent manipulating syntax than convincing oneself of correctness, but perhaps that is to be expected when one considers that the task in a formalisation such as this one is ultimately to convince the *machine* of the correctness, not the user.

4.2 Deliverables

The most salient achievement of this project, then, is a novel and constructive formalisation of a selection of Hoare logic inference rules in Agda. This has been achieved with a deep embedding of a simple *while* language 'Mini-Imp.' This language was given an operational semantics via a small-step evaluation function *and* an axiomatic semantics via the Hoare logic rules thus demonstrating, nay, *formalising* the interplay and or consistency between the two approaches to programming language semantics. In terms of practical utility the Agda library as a system for reasoning about programs leaves a lot to be desired, but perhaps with much more work it could prove a useful building block for a much more useful tool.

4.3 Reflection on the use of Agda

Reflecting on the usage of Agda for this project, there are some clear positives and negatives. One positive is Agda's support for unicode in source code that allows for some beautiful proofs. With great power, however, comes great responsibility and one shouldn't use fancy unicode characters just because one *can* — a rule that was fallen afoul of more than a few times — but rather, it was learned, only when they can provide greater clarity or terseness.

On that note, an interesting observation is that frequently the aesthetic nature of a signature rendered in all its unicode splendour led to a reluctance to sanity check it semantically — even after hours spent in vain trying to prove it. This psychological bias, once noticed, was overcome by paying closer attention to the semantic content of a signature rather than its syntactic form and also by spending more time considering the signatures themselves before charging in and attempting to prove something syntactically pretty but semantically absurd.

Finally, an oft encountered negative of Agda was the verbose and unhelpful nature of its reporting of goal types. Perhaps this was just a problem particular to this project, but at times, upon asking for the type of a hole in a proof while constructing a proof interactively, a myriad of symbols was produced as a result of Agda fully unpacking all the declarations. See figure 9 in the appendix for an example.

4.4 Missteps & Drawbacks

Many wrong turns were taken on the way to completing this project. Of the most notable was an initial insistence on enforcing a type system within the assertion language without impacting the expression language. The desire was for `‘x + 5’` and other integer valued expressions to *not* be valid assertions but eventually it was realised that the formalisation became far simpler and terser once implicit casting between integers and booleans was assumed.

A big drawback of the current system, somewhate mitigated by the proof obligation interfaces, is the need to formalise simple and obvious lemmas of the assertion language. The cumbersome necessity of formalising the obvious, however, is part and parcel of most formalisations so is not worth much commentary here.

Another shortcoming is the inability to specify the free or non-free variables of an expression and or assertion. Such a facility would allow the proof

of correctness of the SWAP program to be generalised over expressions. With the current formulation, SWAP cannot be said to swap the values of x and y if their initial values X and Y are described by arbitrary expressions as, being arbitrary, they may contain x or y. Relatedly, the only variables available for program specification currently are x, y, and z with extra variables needed to be added as required. This is clunky and ideally some facility or interface would exist that would allow a user the functionality described as 'give me a fresh variable' or, 'give me a fresh variable that is not used in this expression $\mathcal{E}$.' Such functionality, however, is left for possible future work.

4.5 Future Work

As mentioned in section 4.2, the present work is not all that useful in practice. With considerably more work, this library could be a useful tool for proving correct *very* simple programs, the issues addressed in the previous section would need to be addressed, however. Perhaps some semi-automated tool could be developed to at least ease the burden of having to transcribe program snippets into Agda or the pain of tedious syntax manipulation; the final output of the tool being a proof checkable by Agda.

In terms of the formalisation, the next step beyond fixing the issues mentioned prior would be expanding the system with Separation Logic. This would allow for the manipulation of assertions containing pointers and claims upon the heap space, rather than just the values of local variables. In theory this is imminently achievable, but in practice may prove otherwise.

4.6 Conclusion & Personal Reflection

If the authors whose works made up much of the literature referenced in this report could comment on this work, a 'we told you so' would be more than justified. Each one at some point in the papers or monographs read in preparation for this project mentioned the need for 'a fine balance' between 'formality and common sense' and it's fair to say that the scales have tipped away from common sense and towards formality in this project and as a result the most 'practical' output of this work is a three-page proof of the correctness of the SWAP program.

But as I now believe, the central purpose of a constructive formalisation isn't in practicality but in the education and entertainment of the person doing the constructing and in that sense, the project is a great success.

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] 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.

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

[10] Peter O'Hearn, John Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In Laurent Fribourg, editor, *Computer Science Logic*, pages 1–19, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[11] 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.

5 Appendix

Figure 4: The full definition of the small-step evaluation function [1/2]

```
-------------------------------------------------------------
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: The full definition of the small-step evaluation function cont. [2/2]


```
-------------------------------------------------------------
-- 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/implementation of the Axiom of Assignment.

```
D0-Axiom-of-Assignment i e P s (wff , ⊢sub) (suc n , p)
  with evalExp e s | inspect (evalExp e) s
... | (just v) | [ eq ] = Ψ
  where

  lem₁ : ∀ v s → evalExp (term (Var v)) s ≡ getIdVal v s
  lem₁ v s = refl

  Is-just-witness-rewrite : ∀ {l} → {A : Set l} → {a : A}
    → (p : Is-just (just a)) → to-witness p ≡ a
  Is-just-witness-rewrite (Any.just x) = refl

  ----------------------------------------------------------
  updateState⇄sub : ∀ P e i v s → evalExp e s ≡ just v
    → evalExp P (updateState i v s) ≡ evalExp (sub e i P) s
  updateState⇄sub (op₂ P x P₁) e i v s comp
    rewrite updateState⇄sub P e i v s comp
      | updateState⇄sub P₁ e i v s comp = refl
  updateState⇄sub (op₁ x P) e i v s comp
    rewrite updateState⇄sub P e i v s comp = refl
  updateState⇄sub (term (Const x)) e i v s comp = refl
  updateState⇄sub (term true) e i v s comp = refl
  updateState⇄sub (term false) e i v s comp = refl
  updateState⇄sub (term (Var x)) e i v s comp with i ?id= x
  ... | yes q rewrite lem₁ x (updateState i v s)
                    | q | updateGet x v s = sym comp
  ... | no q rewrite lem₁ x (updateState i v s)
                    | ignoreTop i v x q s = refl
  ----------------------------------------------------------

  Λ : (updateState i v s) ⊨ P
  Λ rewrite updateState⇄sub P e i v s eq = wff , ⊢sub

  Ψ : (to-witness p) ⊨ P
  Ψ rewrite Is-just-witness-rewrite p = Λ
  ----------------------------------------------------------
```

Figure 6: The full proof of the 'While Rule'/'Rule of Iteration' [1/2]

```
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 6: The full proof of the 'While Rule'/'Rule of Iteration' [2/2] 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) ⊨ (op₂ (op₁ ¬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 ⌋) ⊨ (op₂ (op₁ ¬ₒ B) &&ₒ P)
Λ = subst (λ s → s ⊨ (op₂ (op₁ ¬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 ⊨ (op₂ (op₁ ¬ₒ B) &&ₒ P)
go-false {s} {v} ⊨P p₁ p₂ = ConjunctionIntro _ _ ⊨¬B ⊨P
  where
  ⊭B : ⊬ (just v)
  ⊭B rewrite p₁ = (just tt) , subst (T ∘ not) (sym p₂) tt
  ⊨¬B : s ⊨ (op₁ ¬ₒ B)
  ⊨¬B rewrite p₁ = (NegationIntro (just v) (⊭B))
------------------------------------------------------------------
go {s} (suc f) ⊨P ⌊^t C^t ⌋ with
      evalExp B s | inspect (evalExp B) s
... | f@0(just v) | [ p₁ ] with
      toTruthValue {f} (any tt) | inspect (toTruthValue {f}) (any tt)
... | true | [ p₂ ] = go-true {s} {f} ⊨P p₁ p₂ ⌊^t C^t ⌋
... | false | [ p₂ ] rewrite ls-just-witness-rewrite ⌊^t C^t ⌋ = go-false ⊨P p₁ p₂
------------------------------------------------------------------
```

Figure 7: The proof/implementation of EvalDet.
NB 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'} ((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'} ((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 to have two proofs of termination in ij₁ and ij₂ is that the
-- rest of the mechanisms in c are all also 'skip'.  So take 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 8: The proof/implementation of $\lfloor^t\rfloor$-split [1/3]
NB some cases have been omitted but none that vary from the general pattern here.

```
------------------------------------------------------------
⌊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 8: The proof/implementation of $\lfloor^t\rfloor$-split [2/3]
NB some cases have been omitted but none that vary from the general pattern here.


```
------------------------------------------------------------
- Q₁ = WHILE ;
⌊ᵗ⌋-split' (suc f) s Q₁@((WHILE exp DO c) ;) 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
     = (Any.just tt) , f , ≤with (+-comm f 1) , t₁₂ , refl
  - if true ---------------------
  go | f@(just _) | true rewrite
       THEN-comm c ((WHILE exp DO c) ;) Q₂ with
       ⌊ᵗ⌋-split' f s (c THEN WHILE exp DO c ;) Q₂ t₁₂
  go | f@(just _) | true |  t₁ , f' , lt , t₂ , Δ
                         = t₁ , f' , suc≤'' lt , t₂ , Δ
------------------------------------------------------------
- Q₁ = if then else ; Q₁'
⌊ᵗ⌋-split' (suc f) s Q₁@((IF exp THEN c₁ ELSE c₂) ; 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 rewrite THEN-comm c₂ Q₁' Q₂
                    with ⌊ᵗ⌋-split' f s (c₂ THEN Q₁') Q₂ t₁₂
  go | f@(just _) | false |  t₁ , f' , lt , t₂ , Δ
                         = t₁ , f' , suc≤'' lt , t₂ , Δ
  - if true ---------------------
  go | f@(just _) | true rewrite THEN-comm c₁ Q₁' Q₂
                   with ⌊ᵗ⌋-split' f s (c₁ THEN Q₁') Q₂ t₁₂
  go | f@(just _) | true |  t₁ , f' , lt , t₂ , Δ
                         = t₁ , f' , suc≤'' lt , t₂ , Δ
------------------------------------------------------------
```

Figure 8: The proof/implementation of $\lfloor^t\rfloor$-split [3/3]
NB 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 9: An example of the verbosity that was sometimes encountered when Agda reported the type of a hole that was semantically simple, yet syntactically complex.

```
?0
  : (s : S) →
    Σ
    (Data.Maybe.Relation.Unary.Any.Any (λ _ → Agda.Builtin.Unit.⊤)
     ((getVarVal x s ==ᵥ evalExp (sub (term (Var z)) y Y) s) &&ᵥ
      (getVarVal z s ==ᵥ evalExp (sub (term (Var z)) y X) s)))
    (λ x → Data.Bool.Base.T (toTruthValue x)) →
    (φ
     : Σ Agda.Builtin.Nat.Nat
       (λ 𝓕 →
          Data.Maybe.Relation.Unary.Any.Any (λ _ → Agda.Builtin.Unit.⊤)
          (Evaluation.Evaluation.ssEvalwithFuel ♭ ♯ 𝓕 (y := term (Var z) ;)
           s))) →
    Σ
    (Data.Maybe.Relation.Unary.Any.Any (λ _ → Agda.Builtin.Unit.⊤)
     ((getVarVal x (Data.Maybe.to-witness (proj₂ φ)) ==ᵥ
       evalExp Y (Data.Maybe.to-witness (proj₂ φ)))
      &&ᵥ
      (getVarVal y (Data.Maybe.to-witness (proj₂ φ)) ==ᵥ
       evalExp X (Data.Maybe.to-witness (proj₂ φ)))))
    (λ x → Data.Bool.Base.T (toTruthValue x))
?1
  : (s : S) →
    Σ
    (Data.Maybe.Relation.Unary.Any.Any (λ _ → Agda.Builtin.Unit.⊤)
     ((getVarVal x s ==ᵥ evalExp (sub (term (Var z)) y Y) s) &&ᵥ
      (getVarVal z s ==ᵥ evalExp (sub (term (Var z)) y X) s)))
    (λ x → Data.Bool.Base.T (toTruthValue x)) →
    (φ
     : Σ Agda.Builtin.Nat.Nat
       (λ 𝓕 →
          Data.Maybe.Relation.Unary.Any.Any (λ _ → Agda.Builtin.Unit.⊤)
          (Evaluation.Evaluation.ssEvalwithFuel ♭ ♯ 𝓕 (y := term (Var z) ;)
           s))) →
    Σ
    (Data.Maybe.Relation.Unary.Any.Any (λ _ → Agda.Builtin.Unit.⊤)
     ((getVarVal x (Data.Maybe.to-witness (proj₂ φ)) ==ᵥ
       evalExp Y (Data.Maybe.to-witness (proj₂ φ)))
      &&ᵥ
      (getVarVal y (Data.Maybe.to-witness (proj₂ φ)) ==ᵥ
       evalExp X (Data.Maybe.to-witness (proj₂ φ)))))
    (λ x → Data.Bool.Base.T (toTruthValue x))
?2
  : ⟪ (_ Language.Expressions.∧ _) (y =ℕ̌ Y) (z =ℕ̌ X) ⟫
    x := term (Var y) ; ⟪
    (_ Language.Expressions.∧ _) (x =ℕ̌ Y) (z =ℕ̌ X) ⟫
?3
  : ⟪ (_ Language.Expressions.∧ _) (y =ℕ̌ Y) (x =ℕ̌ X) ⟫
    z := term (Var x) ; ⟪
    (_ Language.Expressions.∧ _) (x =ℕ̌ Y) (z =ℕ̌ X) ⟫
?4 : 𝒫 ((_ Language.Expressions.∧ _) (x =ℕ̌ Y) (y =ℕ̌ X)) (′ φ)
?5 : Data.Bool.Base.T (toTruthValue ?4)
```