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

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

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

at the
University of Birmingham
School of Computer Science
July 2021

Abstract

Program correctness is a perennial problem for software engineers and computer scientists alike. Many methods exist for establishing the correctness of a program and broadly speaking these methods fall into one of two paradigms; a program can be tested 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 the propositional calculus used in the reasoning about programs in the guise of Mini-Imp's expression language. Interfaces were also used to seperate out the concerns of proving program correctness and proving trivial results within the expression language such as conjunction elimination or the distributivity of multiplication over addition.

The final result is an Agda library that is fit for the purpose of reasoning about and proving correct simple imperative-style programs using the implemented Hoare logic rules. A limitation of the work is the simplicity of the Mini-Imp language and corresponding lack of more sophisticated logical rules meaning there is no facility for reasoning about more complex language constructs like procedures, arrays or pointers. However, more powerful logics such as 'separation logic' — an extension of Hoare 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 Introduction

This is some text. That will not be in my report. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortisfacilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdietmi nec ante. Donec ullamcorper, felis non sodales.. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam lobortisfacilisis sem. Nullam nec mi et neque pharetra sollicitudin. Praesent imperdietmi nec ante. Donec ullamcorper, felis non sodales...

2 Preliminaries & Literature Review

2.1 Weakest Precondition

How is one to give a semantics to computation? One answer is to provide a model of computation that denotes how a mechansim, a computation, or program[1] is to be computed; such as a Turing machine, or a FSA. Another way however is to describe what the mechanism, computation, or program can *do* for us; that is, specifiying what input states it can accept, and what states it will produce.

OR: given a desired output (of states) specifying, how (read, in which input states), if at all, it can produce the desired output.

This approach gives us a way of specifying a . . . without caring about the eventual form of the mechanism if indeed it ever takes form at all!

Computation as traversing the state space. Descartes - Cartesian product - [Dijkstra,p12]

Weakest precondition (according to Dijskstra) is unique when considered as a state space, but multiple predicates could denote the same space. (i.e. $x == y$ and $y == x$)

Strongest postcondition? [Gries, exercise 4 section 9.1]

'If for a given P, S, and R, $P \Rightarrow wp(S, R)$ holds, this can often be proved without explicit formulation — or, if you prefer, "computation", or "derivation" — of the predicate $wp(S, R)$'

Note that in the text [dijkstra], $wp(S, R)$, is used interchangeably as a predicate and as the state space that said predicate captures. With our constructive formalisation however, this lack of precision is not possible nor

[1] the three words here being used synonymously

desired, so we end up with the, perhaps superfluous, distinction between predicates and the state space that they describe. Meaning that under our formalisation, $wp(S, F)$ is empty when considered as a state space, but inhabited when considered as a predicate (inhabited uniquely by F itself). This exposition also explains why «F»S«Q» is an inhabited type, as F *is* a valid precondition of any computation for any postcondition (think absurd function, or bluff function). << Actually explained by the fact that what we have formalised is the weakest *liberal* precodnition!

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

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

2.2 Hoare Logic

2.3 Agda

2.4 Constructive Mathematics

'Agda is a constructive mathematical system by default, which amounts to saying that it can also be considered as a programming language for manipulating mathematical objects.' - MHE

2.5 Formal Proof

2.6 Applications

3 Specification

3.1 Obfuscating Interfaces

Might have also wanted to abstract away expression language (page 42 surface properties (Ligler))

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

3.2 Exppresion Language

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

(day = 23) Dijkstra's example

T/F, x == 2

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

```
even⟨_⟩ : Exp → Exp
even⟨ P ⟩ = op₂ (op₂ P %ₒ (const ②)) ==ₒ (const ⓪)
```

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

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

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

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

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

Figure 1: Example of reasoning with the deep embedding of propositional logic

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

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

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

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

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

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

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

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 ;) ;) );
```

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

Aliquam tortor. Morbi ipsum massa, imperdiet non, consectetuer vel, feugiat vel, lorem. Quisque eget lorem nec elit malesuada vestibulum. Quisque sollicitudin ipsum vel sem. Nulla enim. Proin nonummy felis vitae felis. Nullam pellentesque. Duis rutrum feugiat felis. Mauris vel pede sed libero tincidunt mollis. Phasellus sed urna rhoncus diam euismod bibendum. Phasellus sed nisl. Integer condimentum justo id orci iaculis varius. Quisque et lacus. Phasellus elementum, justo at dignissim auctor, wisi odio lobortis arcu, sed sollicitudin felis felis eu neque. Praesent at lacus.

```
- Program composition
_THEN_ : Program → Program → Program
```

Figure 3: The deep embedding in Agda of our ‘Mini-Imp’ imperative language.

```
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
```

```
(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
```

Aliquam tortor. Morbi ipsum massa, imperdiet non, consectetuer vel, feugiat vel, lorem. Quisque eget lorem nec elit malesuada vestibulum. Quisque sollicitudin ipsum vel sem. Nulla enim. Proin nonummy felis vitae felis. Nullam pellentesque. Duis rutrum feugiat felis. Mauris vel pede sed libero tincidunt mollis. Phasellus sed urna rhoncus diam euismod bibendum. Phasellus sed nisl. Integer condimentum justo id orci iaculis varius. Quisque et lacus. Phasellus elementum, justo at dignissim auctor, wisi odio lobortis arcu, sed sollicitudin felis felis eu neque. Praesent at lacus.

```
ssEvalwithFuel : ℕ → Program → S → Maybe S
```

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

```
--------------------------------
-- 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
--------------------------------
```

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

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

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

```
⌊ᵗ_,_ᵗ⌋ : C → S → Set
⌊ᵗ_,_ᵗ⌋ = Terminates

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

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

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

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

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

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

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

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

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

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

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

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

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

3.3 Language

3.4 Axioms & Rules

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

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

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

$\mathsf{D0\text{-}Axiom\text{-}of\text{-}Assignment} : \forall\ i\ e\ P$

$$\frac{}{\to \ll (\mathsf{sub}\ e\ i\ P) \gg (\ i := e\ ;\ ) \ll P \gg}$$

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

$\mathsf{D1\text{-}Rule\text{-}of\text{-}Consequence\text{-}post} : \forall\ \{P\}\ \{Q\}\ \{R\}\ \{S\}$

$$\frac{\to \ll P \gg Q \ll R \gg \to R \Rightarrow S}{\to \ll P \gg Q \ll S \gg}$$

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

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

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

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

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

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

```
D3-While-Rule : ∀ {P} {B} {C}

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

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

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

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

4 Implementation

4.1 Constructive Termination

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

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

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

Mauris tempus eros at nulla. Sed quis dui dignissim mauris pretium tincidunt. Mauris ac purus. Phasellus ac libero. Etiam dapibus iaculis nunc. In lectus wisi, elementum eu, sollicitudin nec, imperdiet quis, dui. Nulla viverra neque ac libero. Mauris urna leo, adipiscing eu, ultrices non, blandit

Figure 4: D3-While: Full proof of the while rule; the crucial rule for reasoning with Hoare Logic:

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

                        ⋮
```

Figure 4: D3-While: Full proof of the while rule; the crucial rule for reasoning with Hoare Logic:

cont.

⋮

```
- This new proof of termination Rᵗf has same output
isDet : † Rᵗf ≡ † Rᵗ
isDet = EvalDet {_} {f} {f '} (WHILE B DO C ;) Rᵗf Rᵗ
- and said output is identical to the original output
Δ' : † Rᵗf ≡ † ⌊ᵗ Cᵗ⌋
Δ' rewrite isDet = Δ
- which we can now use in a recursive call:  (suc f) ⇒ f
GO : († Rᵗf) ⊨ (op₂ (op₁ ¬ₒ B) &&ₒ P)
GO = go {† Lᵗ} f ⊨P' Rᵗf

- and finally get the type we need via substitution with Δ'
Λ : († ⌊ᵗ Cᵗ⌋) ⊨ (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 Cᵗ⌋ with
      evalExp B s | inspect (evalExp B) s
... | f@(just v) | [ p₁ ] with
      toTruthValue {f} (any tt) | inspect (toTruthValue {f}) (any tt)
... | true | [ p₂ ] = go-true {s} {f} ⊨P p₁ p₂ ⌊ᵗ Cᵗ⌋
... | false | [ p₂ ] rewrite Is-just-just ⌊ᵗ Cᵗ⌋ = go-false ⊨P p₁ p₂
--------------------------------
```

eu, dui. Maecenas dui neque, suscipit sit amet, rutrum a, laoreet in, eros. Ut eu nibh. Fusce nec erat tempus urna fringilla tempus. Curabitur id enim. Sed ante. Cras sodales enim sit amet wisi. Nunc fermentum consequat quam.

4.2 Small Step Evaluation with Fuel

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

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

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

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

4.3 Termination Splitting

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

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

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

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

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

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

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

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

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

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

4.4 Axiom & Rules in Agda

4.5 Using the System to Reason about Programs

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

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

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

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

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

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

5 Evaluation

5.1 Using Agda

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

5.2 Missteps

Downsides: Having to define all logical manipulations in the interface. Some mechanism for making this less painful would be nice. Equally however, said logical manipulations are not really of the main concern. Just because nothing can be postulated for the proof to have a computational meaning,

Figure 5: ssEvalwithFuel: The small-step evaluation function

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

⋮

Figure 5: ssEvalwithFuel cont.

⋮

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

Figure 6: The full proof that evaluation is deterministic via proof that for any two proofs of termination, the resultant states serving as evidence for each of those proofs - in accordance with them being *constructive* proofs - will be identical.
n.b. that the † function is the function that extracts the witness from the proof of termination - i.e. the resultant state after the computation has terminated successfully.

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

Figure 7: [t]-split: The function for splitting two proofs of termination.

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


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



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

                              ⋮
```

Figure 7: [t]-split cont.
n.b. some cases have been omitted but none that vary from the general pattern here.

⋮

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

Figure 8: SWAP: Using the library to formalise the correctness of the SWAP program:

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

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

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

  - Reasoning backwards from Postcondition Q to Precondition P

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

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

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

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

                                   ⋮
```

Figure 8: SWAP: Using the library to formalise the correctness of the SWAP program: cont.

⋮

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

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

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

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

⋮

Figure 8: SWAP: Using the library to formalise the correctness of the SWAP program: cont.

⋮

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


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

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

doesn't mean we need be bound by this restriction. Indeed, there is little reason for us to care about whether or not our proofs have this computational context so long as we trust the parts we omit, and as these ommisions are oft simple logical manipulations, leaving them only in the record but not actually proved would be perfectly sensible and allow more focus to be on the manipulation of Hoare triples to reason about programs.

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

5.3 Future Work

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

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

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

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

5.4 Conclusion

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

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

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

I certainly found it enjoyable to reason about the swap program and have Agda check my workings for me.

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

6 Appendix

References

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

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

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

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

[5] 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.

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

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

[8] 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.