University of Birmingham
School of Computer Science
MSc Advanced Computer Science

Second Semester Mini Project

Formalising Operational and Denotational Semantics in Agda

Natalie Ravenhill

Supervisor : Neelakantan Krishnaswami

Date: April 22, 2016

Abstract

Both operational and denotational semantics are used by Computer Scientists in order to understand the meaning of programs and prove their correctness. Here we define the syntax of a small imperative language and formalize its operational and denotational semantics in the proof assistant Agda. Soundness and Completeness of the operational semantics are proved with relation to the denotational semantics. The formalization of this proof ensures that the semantics are correct, so we have a computer checked proof that all programs that can be executed are correct.

Keywords

Operational Semantics, Denotational Semantics, Formal Verification, Agda, Soundness, Completeness

Contents

Chapter 1

Introduction

1.1 Semantics

Semantics is the study of the meaning of programs. It is used as a method to express programs, so we can understand how they work and reason about them mathematically.

Early programming languages such as ALGOL 60 did not have formal semantics. As has been explained in the textbooks [Gunter, 1992] and [Hüttel, 2010], ALGOL 60 was originally specified in a report [Backus et al., 1960], which contained English language descriptions of the semantics of the language. These descriptions were ambiguous, and a revised report [Backus et al., 1963] was created that still contained ambiguities, as described in [Knuth, 1967]. These ambiguities would have been much easier to spot and remove if there had been a way to formally specify and verify ALGOL 60's semantics.

There are still languages that use informal semantics, Javascript being a good example. It has a specification that is maintained by an organisation, [ECMA International, 2011] , but did not have formal semantics until recently. [Maffeis et al., 2008, Guha et al., 2010].

Early work in semantics included formalising programming languages in terms of λ calculus, starting with ALGOL 60 in [Landin, 1964, 1965]. This enables the λ calculus to be used as a model in which properties of the language can be expressed and proved . These models (or calculi) have also been created for modern imperative languages, such as Featherweight Java, [Igarashi et al., 1999] for Java, which formalises a subset of Java to be used in proofs.

Models can also be created for the λ calculus itself. PCF is a programming language that is an augmented version of the typed lambda calculus. It was created by Dana Scott [Scott, 1993] and described by [Milner, 1973]. Plotkin

[Plotkin, 1977] related its operational semantics to its denotational semantics.

This work lead to **Denotational Semantics**, which was created by Dana Scott and Christopher Strachey in the late 1960s/ early 1970s. [Scott, 1970, Scott and Strachey, 1971]. In this style of semantics, a program is specified by constructing a mathematical model of it in which we study the effects of the program. The models are usually created using *domains*, which are partially ordered sets with extra propertes.

Operational Semantics is a contrasting approach where we specify a program by describing its execution on states in the form of a derivation tree. There are two main types of operational semantics.

- *Structural Operational Semantics* [Plotkin, 1981, 2004]. In this approach, every step of the execution is described.
- *Natural Semantics* [Clément et al., 1986], in which we just consider the input an overall output of the execution.

Having both approaches lets us relate one to the other, to prove they are useful. There are other approaches to semantics such as Axiomatic Semantics, in which we use logical assertions inbetween program statements that must hold before/after the statements, and game semantics, which applies Game Theory concepts to semantics. In our report we just consider operational and denotational semantics, as they are the most traditional approaches to semantics.

1.2 Related Work

As well as writng proofs in our formal semantics, we can be even more confident about their correctness by creating formal computer proofs of our semantics. There has been much work in this area.

One of the applications of this formalised semantics is correct compilers for programming languages, created from executable formal semantics of a language. The most well known example is CompCert, [Leroy, 2006, 2009], a verified compiler for C programs. Tthere is also an executable formal semantics for PHP [Filaretti and Maffeis, 2014].

The formal semantics of Javascript mentioned previously has been used to create JSCert, [Bodin et al., 2014], a mechanised specification of Javascript, written in the Coq proof assistant [Bertot et al., 2004]. It can be used to write formal proofs with the Javascript semantics. It is used in JSRef, an interpreter for Javascript that extracts OCaml programs from the Coq specification, so the programs will be correct with respect to the JSCert specification.

Proof assistants other than Coq have also been used to formalise programming language semantics. such as HOL/Isabelle [Martini, 2013, Norrish, 1998]. We are interested in using the proof assistant Agda [Norell, 2009], as it has been

used to implement semantics in some contexts [Danielsson, 2006, 2012, Jeffrey, 2013] but is less widely used than Coq.

1.3 Agda

Agda is a proof assistant and dependently typed programming language, based on Martin Löf type theory. Its syntax is similar to Haskell, but unlike Haskell, dependent types can be defined. Another feature of Agda is that source files can be used to produce LaTeX documents using the extension ".lagda". This report is a pdf file that was generated from a ".lagda" file, so if that file were to be loaded into Agda, we can use Agda's typechecker to show that the proofs are correct.

1.4 Contributions

In this project we study operational semantics and denotational semantics, by formalising a small imperative language similar to the **While** language described in [Nielson and Nielson, 2007]. We define the syntax of the language in section 2.2, formalise the operational semantics of the language in Agda in chapter 3 and denotational semantics in chapter 4.

Finally we prove two properties, in chapter 5, that the operational semantics are sound with respect to the denotaional semantics, and that the denotational semantics are complete (that is, any valid denotation can be executed by the operational semantics). This means that we have proved that we cannot write any incorrect program in our language and all programs that are valid semantically can be implemented in our language.

Chapter 2

Language

2.1 Syntax of Language

Our language is a small imperative language with variable assignment, arithmetic expressions, boolean expressions, conditional branching and repetition.

Grammar We give the following grammar for our language, in a similar way to the language in [Nielson and Nielson, 2007]:

$$
\begin{aligned}
&a ::= \text{n: } n \mid \text{v: } x \mid a +' a \mid a * a \\
&b ::= tt \mid ff \mid a =' a \mid a \leq' a \mid \neg' b \mid b \wedge' b \\
&S ::= x := a \mid \text{skip} \mid S ;' S \mid \text{if } b \text{ then } S \text{ else } S \mid \\
&\text{repeat } n \text{ do } S
\end{aligned}
$$

There are 3 rules in the grammar:

- a represents an arithmetic expression, which is a number n, from the set $\mathbb{N}$, of the natural numbers, or a variable x from a set of variables defined in the program, or it is formed by the addition / multiplication of two other arithmetic expressions
- b represents the boolean expressions, which can be the terminals tt or ff, which represent true and false, a boolean expression representing the application of equality or $\leq$ on two arithmetic expressions, or an expression representing the negation of another boolean expression, or conjunction of two other boolean expressions. These constructors can be used to form other logical formulae, such as disjunction and implication. (see section 2.2)
- S represents any statement that can be executed in our language. This includes assignment, in which we can assign the value stored in a variable

x to be an arithmetic expression a, a skip statement that does nothing, statement composition, conditional branching with an if statement, and repetition with a repeat statement.

Differences from While We made some changes from the **While** language in [Nielson and Nielson, 2007], to make its Agda implementation possible/easier:

- The data type Agda uses for natural numbers does not implement subtraction, so we removed subtraction from our arithmetic expressions.
- All functions in Agda are total, so non termination requires the use of partial functions, which are difficult to implement without using compiler options. Therefore, we removed the while loop from the language, to ensure that all statements will terminate.

2.2 Syntax in Agda

Now we want to recreate our syntax as an executable Agda file. All Agda files must begin with the required imports, so first we must import the Data.Nat file from the Agda library. This allows us to use the set of natural numbers defined by Agda and not have to reimplement it ourselves:

```
open import Data.Nat
```

The datatype of natural numbers is defined inductively and has two constructors, zero which takes no parameters and just returns a natural number, and suc, which takes aniother natural number n as parameter and returns suc.n, which is a number one more than n.

We create a datatype, Var, which is a set of variables in which each variable, x, is individually specified.

```
data Var : Set where
  x' : Var
  y' : Var
  z' : Var
```

Above we have a set of three variables x', y' and z' defined that can be used in our programs. Now we can assign an arithmetic expression to any of these three variables in a statement, or read their value in an arithmetic expression.

Arithmetic Expressions We define a data type Aexp, in which each constructor is a possible application of the rule a in our grammar for arithmetic expressions:

```
data Aexp : Set where
  n:_ : ℕ → Aexp
  v:_ : Var → Aexp
  _+'_ : Aexp → Aexp → Aexp
  _*'_ : Aexp → Aexp → Aexp
```

The n: and v: constructors take an element of ℕ or our variable set Var and return a value of type Aexp. The addition and multiplication constructors take two other Aexp values and return a new value of type Aexp.

The underscores will be replaced with the values given to the constructors. For example, if we have an expression that is just the number 5 , we have:

```
aex : Aexp
aex = n: 5
```

where _ is replaced with the element of ℕ we passed as a parameter, which here is 5. In our grammar we used the n: and v: terminals for numbers and variables because if we had just used the numbers and letters themselves, then Agda's type inference (in which it guesses the type of a value) would not be able to tell if 5 should be a natural number or an arithmetic expression.

Boolean Expressions We define a data type Bexp, in which each constructor is a possible application of of the rule b in our grammar for boolean expressions:

```
data Bexp : Set where
  tt : Bexp
  ff : Bexp
  _≡'_ : Aexp → Aexp → Bexp
  _≤'_ : Aexp → Aexp → Bexp
  ¬'_ : Bexp → Bexp
  _∧'_ : Bexp → Bexp → Bexp
```

tt and ff are terminals in our grammar, so their constructors take no parameters. $\equiv'$ and $\leq'$ take two Aexp parameters, which are the arithmetic expressions being compared. $\neg'$ and $\wedge'$ take an appropriate number of Bexp parameters for the arity of the logical formulae.

For example, if we wanted the boolean expression for $3 \leq 5$, we would give n: 3 and n: 5 as parameters to the $\leq'$ constructor:

```
bex : Bexp
bex = (n: 3) ≤' (n: 5)
```

Now we can see that the underscores in _≤'_ have been replaced with the two Aexps n: 3 and n: 5, giving us a Bexp value.

Extending the expressions The data types for expressions can easily be extended to add more expressions. Here we specify a Bexp' type with more logical operators added.

```
data Bexp' : Set where
  tt : Bexp'
  ff : Bexp'
  _≡'_ : Aexp → Aexp → Bexp'
  _≠'_ : Aexp → Aexp → Bexp'
  _≤'_ : Aexp → Aexp → Bexp'
  _≥'_ : Aexp → Aexp → Bexp'
  _<'_ : Aexp → Aexp → Bexp'
  _>'_ : Aexp → Aexp → Bexp'
  ¬'_  : Bexp' → Bexp'
  _∧'_ : Bexp' → Bexp' → Bexp'
  _∨'_ : Bexp' → Bexp' → Bexp'
  _⇒_ : Bexp' → Bexp' → Bexp'
  _⇔_ : Bexp' → Bexp' → Bexp'
```

However, these operators can be formed from the ones we already have. We get ∨' from De Morgan's Law and Double Negation Elimination:

```
or : Bexp → Bexp → Bexp
or p q = ¬' ((¬' p) ∧' (¬' q))
```

Now we can use this to get implication, as it is equivalent to $(\neg p) \vee q$:

```
imp : Bexp → Bexp → Bexp
imp  p q = or (¬' p) q
```

Then we can use implication to get bidirectional implication:

```
bimp : Bexp → Bexp → Bexp
bimp p q = imp p q ∧' imp q p
```

For the arithmetic expressions, we get ≠' with ≡' and ¬'.

```
ne : Aexp → Aexp → Bexp
ne x y = ¬' (x ≡' y)
```

We can use this with $\leq'$ to form $<'$:

```
lt' : Aexp → Aexp → Bexp
lt' x y = (x ≤' y) ∧' ne x y
```

Then $>$ is the same as "not less than":

```
gt' : Aexp → Aexp → Bexp
gt' x y = ¬' lt' x y
```

And finally $\geq'$ is just greater than or $\equiv'$:

```
geq' : Aexp → Aexp → Bexp
geq' x y = or (gt' x y) (x ≡' y)
```

Now we can see that the type of boolean expressions in our original language is just as expressive as the type Bexp' that contains additional expressions.

State In a program at a certain point of execution, the state represents the values of all of the variables at that current point of execution. Therefore in Agda, we model the state as a function that maps elements of type Var to the natural number they contain. We use a natural number because variables can only be assigned arithmetic expressions, which reduce to either a natural number, or another variable. Therefore each variable must eventually reduce to a natural number.

```
State : Set
State = Var → ℕ
```

Statements We define a type Stm for statements, in which each contructor is a possible application of the rule S in our grammar for statements:

```
data Stm : Set where
  _:=_ : Var → Aexp → Stm
  skip : Stm
  _;'_ : Stm → Stm → Stm
  If_then_else_ : Bexp → Stm → Stm → Stm
  repeat_do_ : ℕ → Stm → Stm
```

skip is a terminal in our grammar, so its constructor takes no parameter. Variable assignment takes a variable from Var and an arithmetic expression as its

parameters. Composition takes two statements to form a third combined statement, if_then_else takes a boolean expression for its condition and two statements, one of which will be executed on the result of the Bexp, and this creates another statement value. Finally repeat_do takes a number n in $\mathbb{N}$ and a statement and executes that statment n times, so the Stm value returned reprsents this repeated execution.

Now we have given all of the syntax of the language, so we can write a program as a series of statements and obtain a parse tree from that. However to work out what the actual behaviour of the program is and the effect it has on the state, we need to define the semantics of our languge.

Chapter 3

Operational Semantics

Operational Semantics describe how a program affects the state by actually executing each statement of the program on a representation of the state. The executions are completed using inference rules.

Of the two types of operational semantics, we implement Natural Semantics, as just requiring the initial and final state makes the proofs much shorter and easier to explain.

We define the Operational Semantics in their own Agda file, to ensure they are completely separate from the denotational semantics. First we import the necessary files, including the syntax we defined and our llibrary of helper functions (see section 7.1).

```
open import dSyntax
open import Lib
```

And also the natural numbers and booleans, so we don't need to redefine them ourselves.

```
open import Data.Nat
open import Data.Bool
```

The rules we give in Natural Semantics are inference rules, so we can specify these rules for each of our Arithmetic and Boolean expressions and statements:

3.1 Arithmetic Expressions

Variables and numbers do not require any parameters, so their rules need no assumptions. Therefore they have the simplest inference rules:

VARIABLES

$$s \vdash \mathsf{v}{:}x \Downarrow s(x)$$

NUMBERS

$$s \vdash \mathsf{n}{:}n \Downarrow n$$

In these rules we start with a state s, and the turnstile symbol $\vdash$ says that from this s we can get what is on the right hand side of the turnstile. The $\Downarrow$ means "evaluates".

In the variable rule we have $\mathsf{v}{:}\ x$, so the x is applied as an argument to the state function, which returns the natural number stored in that variable. The numbers rule just takes the number n supplied and returns that.

Next, we have the rules for addition and multiplication, which will be identical to each other except for the mathematical operators they use. As they both complete calculations based on other arithmetic expressions, a_1 and a_2, we must first assume that we have evaluated these expressions to obtain their results, which are the numbers we are adding/muiltplying.

ADDITION

$$\frac{s \vdash a_1 \Downarrow n_1 \qquad s \vdash a_2 \Downarrow n_2}{s \vdash a_1 +' a_2 \Downarrow n_1 + n_2}$$

MULTIPLICATION

$$\frac{s \vdash a_1 \Downarrow n_1 \qquad s \vdash a_2 \Downarrow n_2}{s \vdash a_1 *' a_2 \Downarrow n_1 * n_2}$$

As an example of this, if we have $(\mathsf{n}{:}\ 1)\ +'\ (\mathsf{v}{:}\ x)$ where x contains the value 3, this will evaluate to

$$\frac{s \vdash \mathsf{n}{:}1 \Downarrow 1 \qquad s \vdash \mathsf{v}{:}x \Downarrow s(x)}{s \vdash \mathsf{n}{:}1 +' \mathsf{v}{:}\mathsf{x} \Downarrow 4}$$

Because $\mathsf{s}\ x$ evaluates to 3 in our state we defined in dSyntax..

3.1.1 Arithmetic Expressions in Agda

What follows is the Agda data type for these inference rules.

```
data aeval : State → Aexp → ℕ → Set where
  aeval-var : (s : State) → (x : Var) → aeval s (v: x) (s x)
  aeval-num : (s : State) → (n : ℕ)  → aeval s (n: n) n
  aeval-+ : (s : State) → (a₁ a₂ : Aexp) → (n₁ n₂ : ℕ)
    → aeval s a₁ n₁
    → aeval s a₂ n₂
    → aeval s (a₁ +' a₂) (n₁ + n₂)
  aeval-* : (s : State) → (a₁ a₂ : Aexp) → (n₁ n₂ : ℕ)
    → aeval s a₁ n₁
    → aeval s a₂ n₂
    → aeval s (a₁ *' a₂) (n₁ * n₂)
```

We chose to implement the operational semantics as a data type as opposed to a function, because the derivations we obtain for a program from the inference rules for expressions form trees. The data type we have above is a very specific form of tree, where the branches are represented by the evaluations of different parameters to the expression at the root of the tree, which is specified by the constructor of aeval.

The inference rules for each different arithmetic expression are represented by different constructors in the data type. Each constructor takes a different amount of parameters of different type depending on the inference rule we are implemehting. All values of aeval we output include a state, arithmetic expression (which is the entire expression we are evaluating) and the overall numerical value of that expression. The addition and multiplication functions on natural numbers that give the actual values are defined in the Data.Nat library file.

The syntax in Agda is different to that of the inference rules, as we have to give the data type a name to show Agda the values are of this type. Therefore our type is called aeval and contains 3 arguments; the state we are applying the expression to, the Aexp we are evaluating and the natural number value that wil be the result of the evaluation. Therfore an element of aeval s a n is the same as writing $s \vdash a \downarrow n$.

The return type is Set, because all possible elements of aeval form a set, just like all elements of natural numbers do, or the set $\{true, false\}$ of boolean values does. Higher types can be formed, which are types of types, that use Set_1 in Agda, but we do not use these in this project.

For aeval-+ and aeval-* we supply extra aeval arguments to the data type, which represent the parameters to the inference rule that are arithmetic expressions, the result of wich must be obtained before evaluating the whole expression.

Then the element of aeval we return is the same as the bottom half of the inference rule.

Rephrasing our example from before as an element of our Agda datatype, we get:

```
aexp-ex' : (s : State) → aeval s ((n: 1) +' (v: x')) (1 + (s x'))
aexp-ex' s = aeval-+ s (n: 1) (v: x') 1 (s x')
  (aeval-num s (suc zero))
  (aeval-var s x')
```

The constructor we use is aeval-+ , which takes a state s, the arithmetic expressions (n: 1) and (v: x'), the results of evaluating these, which are 1 and s x', then finally an acceptable evaluation of the expressions, which are obtained using the aeval-num and aeval-var constructors.

3.2 Boolean Expressions

True and false do not take any parameters, so have the simplest inference rules:

TRUE
$$s \vdash \mathsf{tt} \Downarrow true$$

FALSE
$$s \vdash \mathsf{ff} \Downarrow false$$

The constructor tt evaluates to the boolean value *true* and the ff constructor value evaluates to $false$.

Now we have the rules for equality and $\leq$. These rules take arithmetic expressons, so their assumptions will be the same as the rules for addition and muliplication we already have:

EQUALS
$$\frac{s \vdash a_1 \Downarrow n_1 \qquad s \vdash a_2 \Downarrow n_2}{s \vdash a_1 \equiv' a_2 \Downarrow \mathsf{equals}\ n_1\ n_2}$$

LESS
$$\frac{s \vdash a_1 \Downarrow n_1 \qquad s \vdash a_2 \Downarrow n_2}{s \vdash a_1 \leq' a_2 \Downarrow \mathsf{leq}\ n_1\ n_2}$$

As an example of this, if we have (n: 1) $\leq'$ (v: x) where x contains the value 3, this will evaluate to

$$\frac{s \vdash \mathsf{n{:}1} \Downarrow 1 \qquad s \vdash \mathsf{v{:}}x \Downarrow s(x)}{s \vdash \mathsf{n{:}1} \leq' \mathsf{v{:}}x \Downarrow true}$$

Because $s(x)$ evaluates to 3. This is very similar to our addition example; all we changed was the operator used and the numerical result to a boolean result, *true*, in the conclusion.

We also have boolean expressions that use other boolean expressions as parameters. Their inference rules are similar, except now we have boolean expressions in the assumptions of the rule. Also, $\neg'$ only takes one parameter, as it is a unary function, so we only have one parameter in the top half of its rule:

$$\text{NOT}\ \frac{s \vdash b \Downarrow t}{s \vdash \neg'\ b \Downarrow \mathsf{not}\ t}$$

$$\text{AND}\ \frac{s \vdash b_1 \Downarrow t_1 \qquad s \vdash b_2 \Downarrow t_2}{s \vdash b_1\ \wedge'\ b_2 \Downarrow t_1\ \wedge'\ t_2}$$

As AgdaGreen∧' completes its calculation based on other boolean expressions, b_1 and b_2, we must first evaluate these expressions to obtain their results as the boolean values t_1 and t_2, which are then given to the logical formulae.

3.2.1 Boolean Expressions in Agda

What follows is the Agda data type for these inference rules.

```
data beval : State → Bexp →  Bool → Set where
  beval-true : (s : State) → beval s tt true
  beval-false : (s : State) → beval s ff false
  beval-equals : (s : State) → (a₁ a₂ : Aexp) → (n₁ n₂ : ℕ)
    → aeval s a₁ n₁
    → aeval s a₂ n₂
    → beval s (a₁ ≡' a₂) (equals n₁ n₂)
  beval-leq : (s : State) → (a₁ a₂ : Aexp) → (n₁ n₂ : ℕ)
    → aeval s a₁ n₁
    → aeval s a₂ n₂
    → beval s (a₁ ≤' a₂) (leq n₁  n₂)
  beval-not : (s : State) → (b : Bexp) → (t : Bool)
    →  beval s b t
    → beval s (¬' b) (not t)
  beval-and : (s : State)       → (b₁ b₂ : Bexp) → (t₁ t₂ : Bool)
    → beval s b₁ t₁
```

```
  → beval s b₂ t₂
  → beval s (b₁ ∧' b₂) (t₁ ∧ t₂)
```

The inference rules for each different boolean expression are represented by different constructors in the data type. Each constructor takes a different amount of parameters of different type depending on the inference rule we are implemehting. All values of beval we output include a state, boolean expression (which is the entire expression we are evaluating) and the overall boolean value of that expression. equals and leq are reimplemented in Lib, because their Agda library versions work on Relation types, not natural numbers. not and ∧ are from the Data.Bool library file.

beval-leq and beval-not we supply extra aeval arguments to the data type, which represent the parameters to the inference rule that are arithmetic expressions, the result of wich must be obtained before evaluating the whole expression. Then the state we output is the same as the bottom half of the inference rule.

beval-not and beval-and are similar, but with we extra beval arguments to the data type.

Rephrasing our example from before as an element of our Agda datatype, we get:

```
bexp-ex' : (s : State) → beval s  ((n: 1) ≤' (v: x')) (leq 1 (s x'))
bexp-ex' s = beval-leq s (n: 1) (v: x') (suc zero) (s x')
  (aeval-num s (suc zero))
  (aeval-var s x')
```

The constructor we use is beval-leq , which takes a state s, the boolean expressions (n: 1) and (v: x'), the results of evaluating these, which are 1 and (s x'), then finally an acceptable evaluation of the expressions, which are obtained using the aeval-num and aeval-var constructors. Apart from the beval-leq constructor, this is the same as the addition example above, as they both had the same assumptions in their derivation from the inference rules.

3.3 Statements

The skip statement does not take any parameters, as it is the empty statement:

$$\frac{}{s \vdash \text{skip} \Downarrow s}\ \textsc{skip}$$

The rule for skip takes a state s and just returns the same state.

Assignment The rule for assignment is the following:

$$\frac{}{s \vdash \mathsf{x} := \mathsf{a} \Downarrow \mathsf{update}\ \ s\ x\ n}\ \textsc{assign}$$

Where update is the following Agda function:

```
update : State → Var → ℕ → State
update s x n = λ v → if eqVar v x then n else s v
```

update takes a state, a variable and a numerical value and if that variable exists in the program, it creates a state in which the variable x is now mapped to the value n. This is the same as substituting the variable x with the value n, but in Agda we must return a function, so we recreate the function.

Then state returned by our evaluation rule is just the state function returned from update.

Next we have the rule for composition:

$$\frac{s \vdash S_1 \Downarrow s' \qquad s' \vdash S_2 \Downarrow s''}{s \vdash\ S_1 :' S_2\ \Downarrow s''}\ \textsc{comp}$$

If we start in a state s and get a state s' by applying S_1 and we also have a state s" obtained from applying S_2 to the state s', then we can get the evaluation for the composition of these statements, starting in s and returning the state s".

There are two possible executions of the if statement, depending on the result of the boolean expression, so we have two rules for it:

$$\frac{s \vdash b \Downarrow true \qquad s \vdash S_1 \Downarrow s'}{s \vdash\ if\ b\ then\ S_1\ else\ S_2}\ \Downarrow \mathsf{equals}\ s'\ \textsc{if\_true}$$

Starting in a state s, we have an evaluation of b returning true and an evaluation of S_1 that gives us a state s'. Therefore the evaluation of if b {then S_1 else S_2 when b is true returns the state s'.

$$\frac{s \vdash b \Downarrow false \qquad s \vdash S_2 \Downarrow s'}{s \vdash\ if\ bthen\ S_1\ else\ S_2}\ \Downarrow \mathsf{equals}\ s'\ \textsc{if\_false}$$

For the false rules, we also start in a state s but now have an evaluation of b to false and an evaluation of S_2 that gives us a state s'. Therefore the evaluation of if b then S_1 else S_2 when b is false returns the state s'.

For our repetition operator, we define two rules by induction on the number of repetitions. The base case is when n is 0 and is defined by:

$$\frac{}{s \vdash \text{repeat 0 do } S \Downarrow s}\ \textsc{repeat-0}$$

This is the same as the skip rule; given a state s, we just return this state.

The inductive case, where we have suc n repetitions is the following:

$$\frac{s \vdash \text{S} \Downarrow s' \qquad s' \vdash \text{repeat n do } S \Downarrow s''}{s \vdash \text{suc n do } S \Downarrow s''}$$

The inductive hypothesis here is the given evaluation repeat n do S which given a state s' does the rest of the iterations and returns a state s". The iteration currently being completed happens in state s and returns the state s' that is passed to the rest of the evaluation. Overall this gives us an evaluation starting in s that does all of the iterations and returns a state s".

3.3.1 Statements in Agda

What follows is the Agda data type for these inference rules.

```
data seval : State → Stm → State → Set where
  seval:= : (s : State) → (x : Var) → (a : Aexp) → (n : ℕ)
    → aeval s a n
    → seval s (x := a) (update s x n)
  seval-skip : (s : State) → seval s skip s
  seval-; : (s s' s" : State) → (S₁ S₂ : Stm)
    →          seval s S₁ s'
    → seval s' S₂ s"
    → seval s (S₁ ;' S₂) s"
  seval-if-true : (s s' : State) → (S₁ S₂ : Stm) → (b : Bexp)
    → beval s b true
    → seval s S₁ s'
    → seval s (If b then S₁ else S₂) s'
  seval-if-false : (s s' : State) → (S₁ S₂ : Stm) → (b : Bexp)
    → beval s b false
    → seval s S₂ s'
    → seval s (If b then S₁ else S₂) s'
  seval-repeat : (s s' s" : State) → (S : Stm) → (n : ℕ)
    → seval s S s'
    → seval s' (repeat n do S) s"
    → seval s (repeat (suc n) do S) s"
  seval-repeat-0 : (s : State) → (S : Stm) → seval s (repeat 0 do S) s
```

Each different statement is represented by different constructors in the data type. Each constructor takes a different amount of parameters of different type depending on the inference rule we are implemehting. All values of seval we output include a state, statement (which is the entire statement we are evaluating) and the state returned from the overall execution of that expression.

Chapter 4

Denotational Semantics

In Denotational Semantics we model the programs as mathematical objects and create functions mapping the programs to these objects. In our denotational semantics, we map states to $\mathbb{N}$, the set of natural numbers. Usually denotational semantics uses more complicated mathematical objects called *domains*, which are a type of partially ordered set (see [Gunter, 1992]), but as our language is simple and has no scope for non-termination, we can just use sets.

As with our operational semantics, we specify our denotational semantics in their own file, so first we give the imports:

```
open import dSyntax
open import Data.Nat
open import Relation.Binary.PropositionalEquality
open import Lib
open import Data.List
open import Data.Bool
open import Data.Product
```

4.0.1 Aexp Function

We define a semantic function, A, for the arithmetic expressions in the language. This function takes an expression and a state and maps them to a natural number:

```
A : Aexp → (State → ℕ)
A (n: m) s = m
A (v: a) s = s a
```

```
A (a₁ +' a₂) s = A a₁ s + A a₂ s
A (a₁ *' a₂) s = A a₁ s * A a₂ s
```

For numbers, we just return the number given and for variables we apply the state function to that variable, as in the operational semantics. For the addition and multiplication we first apply A to a_1 and a_2, then add/multiply the results.

As we have kept our semantics separate, we define a new state to use in our denotational examples, in which x' contains 3 and everthing else is just 0:.

```
s' : State
s' x' =  3
s' _ = 0
```

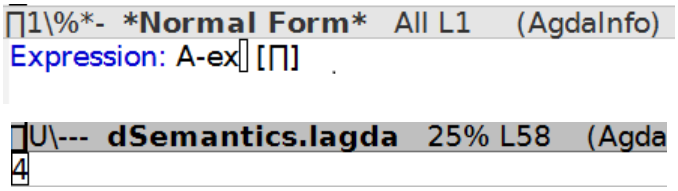
Now for the (1 + x') example we had before, we would give the following in our denotational semantics:

```
A-ex : ℕ
A-ex = A ((n: 1) +' (v: x')) s'
```

When we apply A to this argument, we get

$$\mathsf{A(n{:}1)s'} + \mathsf{A(v{:}x)}\{s' = 1 + \mathsf{sx'} = 1 + 3 = 4$$

In Agda we can evaluate this function using the shortcut "C-c C-n". This returns 4.



All of the semantic functions must be *total*, which means that for each possible arithmetic expression a and each possible state, s, there is exactly one numerical value n such that A a s = n

We can also write this as "for all a : Aexp and s : State, for m and m' : ℕ, where A a s = m = m' , then m = m' ", which is easier to express in Agda:

```
total-A : {a : Aexp} → {s : State} → {m m' : ℕ}
   → A a s ≡ m → A a s ≡ m'
```

```
    → m ≡ m'
total-A refl refl = refl
```

All functions in Agda must be total, so this proof is quite trivial, as it just reduces to reflexivity of equality!

4.0.2 Bexp Function

Next we define a semantic function B for the boolean expressions, which takes a boolean expression and a state and maps them to a boolean value (true or false).

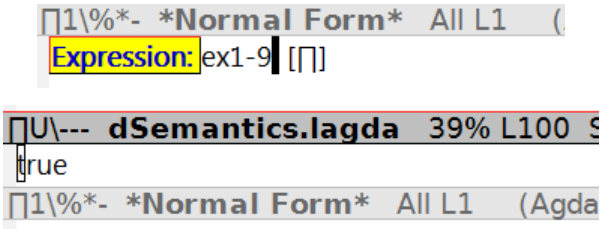
```
B : Bexp → (State → Bool)
B tt s = true
B ff s = false
B (a₁ ≡' a₂) s =  equals (A a₁ s) (A a₂ s)
B (a₁ ≤' a₂) s = leq (A a₁ s) (A a₂ s)
B (¬' b) s = not (B b s)
B (b₁ ∧' b₂) s = (B b₁ s) ∧ (B b₂ s)
```

tt and ff are our syntactic representations of true and false, so we map these to true and false, which are defined in the Agda library file, Data.Bool. Then we use our versions of equals, leq, not, which we reimplmented in Lib. (see 3.2.1), to implement the binary operators on boolean values. We first evaluate the parameters with A, as they are given as arithmetic expressions, then give the results as arguments to the library functions. ∧ is from Data.Bool.

As an example we have the following expression representing $\neg(x \equiv 1)$:

```
ex1-9 : Bool
ex1-9 = B (¬' (v: x' ≡' n: 1)) s'
```

this will evaluate to true:



B must also be total, so for each b : Bexp and each s in State there is exactly one true or false value:

```
total-B : (b : Bexp) → (s : State) → {m m' : Bool}
   → B b s ≡ m → B b s ≡ m'
   → m ≡ m'
total-B b s refl refl = refl
```

For any boolean value, we have an equivalent boolean expression:

```
Bt : Bool → Bexp
Bt true = tt
Bt false = ff
```

And if we convert from a boolean to a Bexp and back again, we should get the same value:

```
pf : (t : Bool) → (s : State) →  B (Bt t) s ≡ t
pf true s = refl
pf false s = refl
```

Extending the expressions In section 2.2, we discussed extending the language with other operators. If we did this, we could easily create a function B' with extra cases for the extra operators:

```
B' : Bexp' → (State → Bool)
B' tt s = true
B' ff s = false
B' (a₁ ≡' a₂) s = equals (A a₁ s) (A a₂ s)
B' (a₁ ≠' a₂) s = not (equals (A a₁ s) (A a₂ s))
B' (a₁ ≤' a₂) s = leq (A a₁ s) (A a₂ s)
B' (a₁ ≥' a₂) s = geq (A a₁ s) (A a₂ s)
B' (a₁ <' a₂) s = (leq (A a₁ s) (A a₂ s))
   ∧ (not (equals (A a₁ s) (A a₂ s)))
B' (a₁ >' a₂) s = (geq (A a₁ s) (A a₂ s))
      ∧ (not (equals (A a₁ s) (A a₂ s)))
B' (¬' b) s = not (B' b s)
B' (b₁ ∧' b₂) s =  (B' b₁ s) ∧ (B' b₂ s)
B' (b₁ ∨' b₂) s = (B' b₁ s) ∨ (B' b₂ s)
B' (b₁ ⇒ b₂) s = or' (not (B' b₁ s)) (B' b₂ s)
```

```
B' (b₁ ⇔ b₂) s = (or' (not (B' b₁ s)) (B' b₂ s))
    ∧ (or' (not (B' b₂ s)) (B' b₁ s))
```

Then we can now prove that for each boolean expression b' in Bexp, there exists a b in Bexp such that b' and b are equivalent:

```
equivBB' : (b' : Bexp') → (s : State) → ∃' Bexp λ b → B b s ≡ B' b' s
```

The cases where we already have the constructor in Bexp are easily solved with reflexivity:

```
equivBB' tt s = tt , refl
equivBB' ff s = ff , refl
equivBB' (a₁ ≡' a₂) s = (a₁ ≡' a₂) , refl
equivBB' (a₁ ≠' a₂) s = (¬' (a₁ ≡' a₂)) , refl
equivBB' (a₁ ≤' a₂) s = (a₁ ≤' a₂) , refl
```

The other cases are proved using properties of equality and converting to/from boolean values:

```
equivBB' (a₁ ≥' a₂) s = (¬' ((a₁ ≤' a₂) ∧' (¬' (a₁ ≡' a₂)))) ,
    sym (≥-equiv (A a₁ s) (A a₂ s))
equivBB' (a₁ <' a₂) s = ((a₁ ≤' a₂) ∧' (¬' (a₁ ≡' a₂))) , refl
equivBB' (a₁ >' a₂) s = ((¬' ((a₁ ≤' a₂) ∧' (¬' (a₁ ≡' a₂))))
  ∧' (¬' (a₁ ≡' a₂))) ,
    cong (λ x →  x ∧(not (equals (A a₁ s) (A a₂ s))))
    (sym (≥-equiv (A a₁ s) (A a₂ s)))
equivBB' (¬' b') s = Bt (B' (¬' b') s)
  , pf (not (B' b' s)) s
equivBB' (b₁ ∧' b₂) s = (Bt (B' (b₁ ∧' b₂) s))
    , pf ((B' b₁ s) ∧ (B' b₂ s)) s
equivBB' (b₁ ∨' b₂) s = (Bt (B' (b₁ ∨' b₂) s))
  , pf ( (B' b₁ s) ∨ (B' b₂ s)) s
equivBB' (b₁ ⇒ b₂) s = (Bt (B' (b₁ ⇒ b₂) s))
    , pf (or' (not (B' b₁ s)) (B' b₂ s)) s
equivBB' (b₁ ⇔ b₂) s = (Bt (B' (b₁ ⇔ b₂) s)) , pf
  ((or' (not (B' b₁ s)) (B' b₂ s))
  ∧ (or' (not (B' b₂ s)) (B' b₁ s))) s
```

4.0.3 Statement Function

Before we define our function for statements, there are some helper functions we need.

cond is used to evaluate If_then_else statements. It takes the result of evaluating a boolean expression, b, the statement executed when b is true and the statement executed when b is false. These parameters are functions because we

want to know how they are evaluated in any state. The final parameter is a state in which these functions are evaluated.

```
cond : (State → Bool) → (State → State) → (State → State)
   → (State → State)
cond p g₁ g₂ s = if p s then g₁ s else g₂ s
```

cond uses p to select which of g_1 or g_2 to execute, then executes the chosen function on the statment.

iterate is used to evaluate repeat n do S statements. It takes a number of iterations n, a function, f, that takes a state and returns a new state, and a state s to execute a function in. It applies the function f to the result of the last application of f, until there are no more iterations left.

```
iterate : (n : ℕ) → (State → State) → State → State
iterate zero f s = s
iterate (suc n) f s = iterate n f (f s)
```

Now we have everything we need to write the semantic function for statements:

```
S : Stm → State → State
S (x := a) s = update' s x (A a s)
S skip s = s
S (S₁ :' S₂) s = S S₂ (S S₁ s)
S (If b then S₁ else S₂) s = cond (B b) (S S₁) (S S₂) s
S (repeat n do S₁) s = iterate n (S S₁) s
```

We use our update function (see section 3.3), to replace x in the state s with the result of a, which is evaluated using A. For skip, S just returns the same state it is given.

For composing statements, we apply S S_2 to the result of S S_1. Then for the if and repeat statements, we apply the functions we described above.

We can write for loops as a special case of a repeat statement. The loop is comprised the compositions of two statements: the first statement sets a counter variable i to the value of 0. The second statement is the repetition, which contains the composition of the statement S' being executed and a statement that adds 1 to the counter variable i and assigns this new value to i. This is usually called *syntactic sugar*, as it provides no new functionality, but makes writing a commonly used expression easier.

```
for : Var → ℕ → Stm → Stm
for i n S' = (i := (n: 0)) :' (repeat n do  S' :' (i := ((v: i) +' (n: 1))))
```

Chapter 5

Soundness and Completeness

In order for our semantics to be useful, we must prove that they are sound and complete. However we can only prove this in one direction, so it is usually the Operational Semantics we are proving the correctness of with relation to the Denotational Semantics. Therefore we take the Denotational Semantics as ground truth.

This means that we have

- Soundness - everything that can be specified in the operational semantics is allowed as the result of one of the semantic functions *(so we don't have anything in the operational semantics that cannot be mathematically evaluated)*
- Completeness - everything that is allowable in a semantic function can be specified in the operational semantics *(so we haven't missed out any executions that could be mathematically evaluated)*

To prove these two statements, we must prove them for the arithmetic and boolean expressions and statements individually. We write these proofs in a separate file, so we first import the relevant library files.

```
open import Data.Nat
open import Data.Bool
open import Relation.Binary.PropositionalEquality
```

Then our library of helper functions

```
open import Lib
```

And finally our syntax and semantics we have already defined:

```
open import dSyntax
open import dSemantics
open import opSemantics
```

5.1 Soundness

To prove soundness, we must know that the arithmetic and boolean expressions and statements are all sound individually:

5.1.1 Soundness of Arithmetic expressions

First we prove the soundness of arithmetic expressions. This means that for every arithmetic expression, a, we can define using our syntax, if we have an evaluation of a in the operational semantics in a state s that returns n, then applying the semantic function A to a and s will give us the value n:

```
aexp-soundness : (s : State) → (a : Aexp) → (n : ℕ)
  → aeval s a n → A a s ≡ n
```

We prove this by induction on the defintition of arithmetic expressions. The inductive hypothesis will be for any a in Aexp, n in $\mathbb{N}$ and s in State, aeval s a $n \rightarrow$ A a $s \equiv n$. The base cases will be for variables and numbers, because these elements are not defined using any other arithmetic expressions:

Numbers and Variables `aexp-soundness s .(v: x) .(s x) (aeval-var .s x) = refl`

For variables, given any variable x, its numerical value will be s x. By definition of the semantics, aeval-var gives us s x. By definition of the semantic function A, A (v: x) $s = s$ x.

Now our proof can be rewritten as $s\ x \equiv s\ x$, which we know is true by reflexivity of equality. In Agda, we construct this equality using the constuctor refl from Relation.Binary.PropositionalEquality.

Soundness of numerical values can be proved similarly. Both the operational semantics and the semantic function return the numerical value n, stored in n: n, so again we get $n \equiv n$, which can be proved by reflexivity of equality:

```
aexp-soundness s .(n: n) n (aeval-num .s .n) = refl
```

Addition Now we prove the inductive cases, starting with addition. For this goal, we want to prove that we can obtain A a_1 s + A a_2 $s \equiv n_1 + n_2$ from the result of evaluating aeval a_1 + a_2 s in the operational semantics:

```
aexp-soundness s .(a₁ +' a₂) .(n₁ + n₂) (aeval-+ .s a₁ a₂ n₁ n₂ x x₁) =
  A a₁ s + A a₂ s
  ≡[ p1 ]
  n₁ + A a₂ s
    ≡[ p2 ] n₁ + n₂
  done
```

To prove the inductive cases, we use equational reasoning, which we define using transitivity of equality, in our library file. (see 7.1). The proof inside the ≡[ ] is the proof that what comes before it is equal to what is after it.

Therefore p1 is a proof that A a_1 s + A a_2 s is equal to n_1 + A a_2 s

```
  where
    p1 :  A a₁ s + A a₂ s ≡ n₁ + A a₂ s
    p1 = cong (λ y → y + A a₂ s) (aexp-soundness s a₁ n₁ x)
```

We prove this using congruence, which means for some function f, that if x ≡ y, then $f(x) \equiv f(y)$. Here we can use it to prove the equality by just proving that A a_1 s = n_1 and replacing the y in y + A a_2 s with n_1.

We get the proof we need for congruence by using the inductive hypothesis for aexp-soundness, as we have already proved it for the base case of numerical values. In our operational semantics evaluation we give as a parameter to aexp-soundness, we have x, which is the evaluation of a_1, so this can be used to define an element of aexp-soundness that represents the part of the proof for a_1

Then we can use congruence again to prove this result is equal to $n_1 + n_2$. We prove A a_2 s = n_2, by applying aexp-soundness with our operational semantics evaluation of a_2, which is given in the parameter x_1.

```
  p2 : n₁ + A a₂ s ≡ n₁ + n₂
  p2 = cong (λ y → n₁ + y) (aexp-soundness s a₂ n₂ x₁)
```

Now we have A a_1 s + A a_2 s ≡ n_1 + n_2, which is exactly the same as the definition in the denotational semantics.

Multiplication We prove the soundness of multiplication in a similar way, as we want to prove that we can obtain A a_1 s * A a_2 $s \equiv n_1$ * n_2 from the result of evaluating aeval a_1 * a_2 s in the operational semantics:

```
aexp-soundness s .(a₁ *' a₂) .(n₁ * n₂) (aeval-* .s a₁ a₂ n₁ n₂ x x₁) =
  A a₁ s * A a₂ s
    ≡[ p1 ]
```

```
    n₁ * A a₂ s
  ≡[ p2 ]
    n₁ * n₂
  done
```

First we use congruence with the proof of soundness for the first parameter, to get A a_1 s * A a_2 s ≡ n_1 * A a_2 s

```
  where
    p1 : A a₁ s * A a₂ s ≡ n₁ * A a₂ s
    p1 = cong (λ y → y * A a₂ s) (aexp-soundness s a₁ n₁ x)
```

Then we can use congruence again to prove this result is equal to $n_1 * n_2$. We prove A a_2 s = n_2, by applying aexp-soundness, with our operational semantics evaluation of a_2, which is given in the parameter x_1, as we did for addition:

```
  p2 : n₁ * A a₂ s ≡ n₁ * n₂
  p2 = cong (λ y → n₁ * y) (aexp-soundness s a₂ n₂ x₁)
```

Now we have proofs for every constructor for arithmetic expressions, so we know they are sound with relation to the denotational semantics.

5.1.2 Soundness of Boolean Expressions

Now we prove the soundness of boolean expressions. This means that for every boolean expression we can define using our syntax, if we have an evaluation of b in the operational semantics in a state s that returns a boolean value t, then applying the semantic function B to b and s will give us the same boolean value t:

```
bexp-soundness : (s : State) → (b : Bexp) → (t : Bool)
  → beval s b t → B b s ≡ t
```

We prove this by induction on the defintition of boolean expressions. The inductive hypothesis will be for any b in Bexp, t in Bool and s in State, beval s b t → B b s ≡ t. The base cases will be for the syntactic constants tt and ff, because these elements are not defined using any other boolean expressions.

True and False For tt, given any state s, its only possible value is true. By definition of the semantic function, B tt s will always be true, so we are now just proving true ≡ true, which we get from refl, the reflexivity of equality:

```
bexp-soundness s .tt .true (beval-true .s) = refl
```

We get a similar result for ff and false, where the proof just reduces to false ≡ false:

```
bexp-soundness s .ff .false (beval-false .s) = refl
```

Equality Now we prove the inductive cases, starting with equality. For this goal, we want to prove that we can obtain equals (A a_1 s) (A a_2 s) ≡ equals n_1 n_2 from the result of evaluating beval $a_1 \equiv' a_2$ s in the operational semantics:

```
bexp-soundness s .(a₁ ≡' a₂) .(equals n₁ n₂) (beval-equals .s a₁ a₂ n₁ n₂ x x₁) =
    equals (A a₁ s) (A a₂ s)
  ≡[ p1 ]
    equals n₁ (A a₂ s)
    ≡[ p2 ]
      equals n₁ n₂
    done
```

Again we use equational reasoning to prove the inductive cases. p1 is a proof that equals (A a_1 s) (A a_2 s) is equal to equals n_1 n_2.

```
  where
    p1 : equals (A a₁ s) (A a₂ s) ≡ equals n₁ (A a₂ s)
    p1 = cong (λ y → equals y (A a₂ s)) (aexp-soundness s a₁ n₁ x)
```

We prove this using congruence and the soundness of arithmetic expressions (which we have already proved holds) to replace (A a_1 s) in equals (A a_1 s) (A a_2 s) with n_1 to get equals n_1 (A a_2 s). x is the operational semantics evaluation of a_1 that we need to define the instance of aexp-soundness that for the inductive hypothesis.

Then we can use congruence again to prove this result is equal to equals n_1 n_2. We prove A a_2 s = n_2 by applying aexp-soundness, with our operational semantics evaluation of a_2, which is given in the parameter x_1.

```
    p2 : equals n₁ (A a₂ s) ≡ equals n₁ n₂
    p2 = cong (λ y → equals n₁ y) (aexp-soundness s a₂ n₂ x₁)
```

Now we have equals (A a_1 s) (A a_2 s) ≡ equals n_1 n_2, which is exactly the same as the definition in the denotational semantics.

Less than or equal to We prove the soundness of $\leq'$ in a similar way, as we want to prove that we can obtain leq (A a_1 s) (A a_2 s) ≡ leq n_1 n_2. This proof is almost identical, apart from the use of leq instead of equals in the denotational semantics, and beval-leq instead of beval-equals in the operational semantics.

```
bexp-soundness s .(a₁ ≤' a₂) .(leq n₁ n₂) (beval-leq .s a₁ a₂ n₁ n₂ x x₁) =
  leq (A a₁ s) (A a₂ s)
```

```
    ≡[ p1 ]
    leq n₁ (A a₂ s)
    ≡[ p2 ]
      leq n₁ n₂
    done
      where
        p1 : leq (A a₁ s) (A a₂ s) ≡ leq n₁ (A a₂ s)
        p1 = cong (λ y → leq y (A a₂ s)) (aexp-soundness s a₁ n₁ x)
        p2 : leq n₁ (A a₂ s) ≡ leq n₁ n₂
        p2 = cong (λ y → leq n₁ y) (aexp-soundness s a₂ n₂ x₁)
```

Negation Negation is a unary operation, so its boolean expression has only one parameter. Evaluating B ¬′ b in the denotational semantics gives us not (B b s) ≡ not t , so we must prove this equality. From the inductive hypothesis we know B b s ≡ t, so we can just use congruence with the function not:

```
bexp-soundness s .(¬' b) .(not t) (beval-not .s b t x) =
  cong (λ y → not y) (bexp-soundness s b t x)
```

Conjunction Finally we have conjunction. ∧ is defined as an infix operator, so the proof is identical to the proofs for ≡′ and ≤′ apart from the positions in the arguments to B b_1 s ∧ B b_2 s and t_1 ∧ t_2.

```
bexp-soundness s .(b₁ ∧' b₂) .(t₁ ∧ t₂) (beval-and .s b₁ b₂ t₁ t₂ x x₁) =
  B b₁ s ∧ B b₂ s
  ≡[ p1 ]
  t₁ ∧ B b₂ s
  ≡[ p2 ]
  t₁ ∧ t₂
  done
```

In ∧′, the parameters to the boolean expression are other boolean expresions so we use the inductive hypothesis beval s b t → B b s ≡ t, instead of soundness of Aexps:

```
  where
    p1 : B b₁ s ∧ B b₂ s ≡ t₁ ∧ B b₂ s
    p1 = cong (λ y → y ∧ B b₂ s) (bexp-soundness s b₁ t₁ x)
    p2 : t₁ ∧ B b₂ s ≡ t₁ ∧ t₂
    p2 = cong (λ y → t₁ ∧ y) (bexp-soundness s b₂ t₂ x₁)
```

Now we have completed all the cases so we know soundness holds for boolean expressions.

5.1.3 Soundness of Statements

Finally we must prove soundness for statements: For every program statement we can define using our syntax, if we have an evaluation of S' in the operational semantics in a state s that returns a new state s', then applying the semantic function S to S' and s will give us the state s':

```
stm-soundness : (s : State) → (S' : Stm) → (s' : State)
  → seval s S' s' → S S' s ≡ s'
```

We prove this by induction on the defintition of statements. The inductive hypothesis will be for any S' in Stm, and s and s' in State, seval s S' s' → S S' $s \equiv s'$.

Skip The base case will be for the skip statement, as this statement just returns the same state it received:

```
stm-soundness s skip .s (seval-skip .s) = refl
```

Given any state s, S skip s just returns s. By definition of the semantics, seval-skip also gives us s. Therefore our proof will reduce to $s \equiv s$ which can be solved using reflexivity of equality.

Assignment Now we prove the inductive cases, starting with assigment. S (x :≡ a) s is the same as update s x (A a s), which is defined by λ v → if eqVar v x then n else s v.

Therefore for this goal, we want to prove that (λ v → if eqVar v x then (A a s) else s v) ≡ (λ v → if eqVar v x then n else s v):

```
stm-soundness s (x := a) ._ (seval:= .s .x .a n x₁) =
  (λ z → if eqVar z x then A a s else s z)
    ≡[ cong (λ v → λ z → if eqVar z x then v else s z) p ]
  (λ z → if eqVar z x then n else s z) done
  where
    p : A a s ≡ n
    p = aexp-soundness s a n x₁
```

We can prove this using congruence and the soundness of the artihmetic expression being assigned to x in order to replace the A a s with its value n.

Composition Next we prove soundness for composition of statements. For this goal, we want to prove that we can obtain S S_2 (S S_1 s) ≡ s'' , from the result of evaluating seval S_1 :' S_2 s in the operational semantics:

```
stm-soundness s (S₁ :' S₂) s" (seval-: .s s' .s" .S₁ .S₂ x x₁) =
    S S₂ (S S₁ s)
    ≡[ cong (λ v → S S₂ v) p1 ]
    S S₂ s'
  ≡[ p2 ]
    s"
  done
    where
      p1 : S S₁ s ≡ s'
      p1 = stm-soundness s S₁ s' x
      p2 : S S₂ s' ≡ s"
      p2 = stm-soundness s' S₂ s" x₁
```

First we use the inductive hypothesis to get a new state from evaluating S_1. We use congruence with the hypothesis to replace (S S_1 s) with its result, s'. Then we do the same with S_2 in the state s' to get the state s''.

If statement Next we prove soundness for the if statement. Because we have two different rules in the operational semantics, there will be two cases, the first being for when the condition in the if statement is true:

```
stm-soundness s (If x then S' else S") s'
  (seval-if-true .s .s' .S' .S" .x x₁ x₂) =
```

S (If x then S' else S'') s is the same as if B x s then S S' s else {S S'' s). Therefore we want to prove that this is equal to the state s' that is obtained by executing S'. The first thing we do is replace the B x s with the boolean value it evaluates to. We do this using the soundness of boolean expressions and congruence to get S S' s. Now we just need to be able to execute this statement, to get its new state s'. We use the inductive hypothesis for statement soundness, as we have x_2 in the operational semantics parameters, which is the element of seval s S_1 s' that we need to assume soundness holds for S_1.

```
  (if B x s then S S' s else S S" s)
    ≡[ cong (λ v → if v then S S' s else S S" s) p ]
S S' s
≡[ stm-soundness s S' s' x₂ ]
s' done
  where
    p : B x s ≡ true
    p = bexp-soundness s x true x₁
```

We prove soundness of if statements where the condition is false in an almost identical way. The main difference is the element of the operational semantics data type passed to the proof uses the constructor seval-if-false instead of seval-

if-true and the use of boolean expression soundness requires us to give a proof that B x $s \equiv$ false instead of true:

```
stm-soundness s (If x then S' else S'') s' (seval-if-false .s .s' .S' .S'' .x x₁ x₂) =
  (if B x s then S S' s else S S'' s)
  ≡[ cong (λ v → if v then S S' s else S S'' s) p ]
  S S'' s
  ≡[ stm-soundness s S'' s' x₂ ]
  s' done
  where
    p : B x s ≡ false
    p = bexp-soundness s x false x₁
```

Repetition Finally we must prove soundness of repetition. In the syntax we have defined two cases for repetition statements, so we must prove each individually. For zero iterations we have the base case. S repeat zero do S' evaluates to iterate 0 (S S') s which is just s by definition of the iterate function. Therefore our proof is now just s $\equiv$ s, which we prove with reflexivity.

```
stm-soundness s (repeat zero do S') .s (seval-repeat-0 .s .S') = refl
```

For the inductive case, S (repeat suc n do S') s evaluates to iterate n (S S') s, which gives us iterate n (S S') (S S' s). Now we must show that evaluating this eventually returns the state s':

```
stm-soundness s (repeat suc n do S') s' (seval-repeat .s s'' .s' .S' .n x x₁) =
  iterate n (S S') (S S' s)
  ≡[ cong (λ v → iterate n (S S') v) (stm-soundness s S' s'' x) ]
```

We use the inductive hypothesis for the given statement S' to show that evaluating S' gets us to the intermediate state s'' given by the operational semantics, so we now have iterate n (S S') s'':

```
  iterate n (S S') s''
  ≡[ stm-soundness s'' (repeat n do S') s' x₁ ]
  s' done
```

In the given operational semantics we have x_1 which is an element of seval s'' (repeat n {do S') s'. This is what we need to apply the inductive hypothesis of statement soundness. In the operational semantics we complete the repetition case by case until there are no more iterations left, thereforewe know by induction that the other iterations are also sound.

Now we have proved soundness for every possible statement, arithmetic expression and boolean expression in our language! As this is all the syntax we have defined our semantics with, it means the operational semantics of our whole language are sound with respect to the denotational semantics.

5.2 Completeness

Now we prove completeness, so we must know that the arithmetic and boolean expressions and statements are all complete individually:

5.2.1 Completeness of Arithmetic Expressions

We get a valid denotation in a given state s, by applying the semantic function (from states to the natural numbers) on the state s. From every valid denotation we can create from our arithmetic expressions and the given state, we obtain a numerical value. Completeness says there is a way of executing the expression in the operational semantics, with the state s, to get exactly the same value as the one we obtained from the semantic function.

```
aexp-completeness : (s : State) → (a : Aexp) → (n : ℕ)
  → A a s ≡ n → aeval s a n
```

We also prove this by induction on the definition of arithmetic expressions. The cases in the expressions are reduced by Agda, using its case splitting functionality, so this is why some of the arguments are preceded by ".". Agda also automatically fills in the A a $s \equiv n$ parameter with refl, as the element of n reduced by Agda is calculated by evaluating A on the arithmetic expression. Therefore this proof will always just be given by reflexivity of equality.

To prove this for each arithmetic expression, we must construct an element of the aeval type. We use the same base cases as before:

Numbers and variables `aexp-completeness s (n: x) .x refl = aeval-num s x`

The operational semantics for numbers is created by the aeval-num constructor. We give it the state s and the number x as parameters, which are given in the arithmetic expression.

```
aexp-completeness s (v: x) .(s x) refl = aeval-var s x
```

The operational semantics for variables is created by the aeval-var constructor. We give it the state s and the variable x as parameters, which are also given in the arithmetic expression.

Addition Now we prove the inductive cases, starting with addition. The inductive hypothesis will be for any a in Aexp, n in $\mathbb{N}$ and s in State, A a $s \equiv n \rightarrow$ aeval s a n. To define the operational semantics for addition, we use the aeval-+ constructor. Its parameters include the state s, and the two arithmetic expressions a_1 and a_2 which are the parameters of the arithmetic expression $a_1 + a_2$.

We also need the values of a_1 and a_2, which we obtain by executing the semantic function A on each value and the state s. Finally we need an elements of aeval s a_1 n_1 and aeval s a_2 n_2, which are constructed using the inductive hypotheses IH_1 and IH_2:

```
aexp-completeness s (a₁ +' a₂) .(A a₁ s + A a₂ s) refl =
  aeval-+ s a₁ a₂ (A a₁ s) (A a₂ s) IH₁ IH₂
```

For IH_1, we use the inductive hypothesis with a_1, so we call the aexp-completeness again with the first expression, a_1. and the result of A a_1 in the state s.

```
  where
    IH₁ : aeval s a₁ (A a₁ s)
    IH₁ = aexp-completeness s a₁ (A a₁ s) refl
```

We then use the inductive hypothesis again with a_2:

```
    IH₂ : aeval s a₂ (A a₂ s)
    IH₂ = aexp-completeness s a₂ (A a₂ s) refl
```

Multiplication We prove completeness for multiplication in almost the same way, but by defining an element of aeval-*, with the state s, a_1 and a_2, their evaluations A a_1 s and A a_2 s, and finally the applications of the inductive hypothesis for a_1 and a_2.

```
aexp-completeness s (a₁ *' a₂) .(A a₁ s * A a₂ s) refl =
  aeval-* s a₁ a₂ (A a₁ s) (A a₂ s) IH₁ IH₂
    where
      IH₁ : aeval s a₁ (A a₁ s)
      IH₁ = aexp-completeness s a₁ (A a₁ s) refl
      IH₂ : aeval s a₂ (A a₂ s)
      IH₂ = aexp-completeness s a₂ (A a₂ s) refl
```

Now we have proofs for every constructor for arithmetic expressions, so completeness of arithmetic expressions holds.

5.2.2 Completeness of Boolean Expressions

For boolean expressions, we get a valid denotation in a given state s, by applying the semantic function (from states to boolean values) on the state s. From every valid denotation we can create from our boolean expressions and the state, we obtain a boolean value, t. Completeness says there is a way of executing the expression in the operational semantics, with the state s, to get exactly the same value, t:

```
bexp-completeness : (s : State) → (b : Bexp) → (t : Bool)
  → B b s ≡ t → beval s b t
```

We prove this by induction on the boolean expressions. For each expression, we must construct an element of the beval type, from the denotational semantic evaluation given.

True and False First we have our base cases for tt and ff.

```
bexp-completeness s tt .true refl = beval-true s
bexp-completeness s ff .false refl = beval-false s
```

The operational semantics for tt is given by the beval-true constructor. We give it the state s and the boolean value true as parameters. The ff case is the same but with the constructor beval-false.

Equality Now we prove the inductive cases, starting with equality. The inductive hypothesis will be for any b in Bexp, t in Bool and s in State, B b s $\equiv$ t $\rightarrow$ beval s b t. To define the operational semantics for equality, we use the beval-equals constructor. Its parameters include the state s, and the two arithmetic expressions a_1 and a_2.

We also need the values of a_1 and a_2, which we obtain by executing the semantic function A on each value and the state s. Finally we need an elements of aeval s a_1 n_1 and aeval s a_2 n_2, which are obtained by the completeness of arithmetic exprssions, which we have already proved:

```
bexp-completeness s (a₁ ≡' a₂) .(equals (A a₁ s) (A a₂ s)) refl =
  beval-equals s a₁ a₂ (A a₁ s) (A a₂ s)
    (aexp-completeness s a₁ (A a₁ s) refl) (aexp-completeness s a₂ (A a₂ s) refl)
```

The proof for $\leq'$ is almost identical, except we use the beval-leq constructor to get our operational semantics:

```
bexp-completeness s (a₁ ≤' a₂) .(leq (A a₁ s) (A a₂ s)) refl =
  beval-leq s a₁ a₂ (A a₁ s) (A a₂ s)
  (aexp-completeness s a₁ (A a₁ s) refl) (aexp-completeness s a₂ (A a₂ s) refl)
```

Negation Negation is a unary operator so is slightly different, as we now have one boolean parameter. We use the constructor beval-not. Its parameters include the state s, and a boolean expression b. We also need the boolean value of b which is obtained from B b s. Then we use the inductive hypothesis of completeness for boolean expressions to get the element of beval s b (B b s) needed to complete the operational semantics.

```
bexp-completeness s (¬' b) .(not (B b s)) refl =
  beval-not s b (B b s) (bexp-completeness s b (B b s) refl)
```

Conjunction Finally we have conjunction. $\wedge'$ is defined as an infix operator so the proof is identical to the proofs for $\equiv'$ and $\leq'$ apart from the positions in the arguments to B b_1 s $\wedge$ B b_2 s and t_1 $\wedge$ t_2, and the fact that we use the inductive hypothesis for boolean expression soundness instead of arithmetic expression soundness:

```
bexp-completeness s (b₁ ∧' b₂) .(B b₁ s ∧ B b₂ s) refl =
  beval-and s b₁ b₂ (B b₁ s) (B b₂ s)
    (bexp-completeness s b₁ (B b₁ s) refl) (bexp-completeness s b₂ (B b₂ s) refl)
```

Now we have completed all the cases so we know soundness holds for boolean expressions.

5.2.3 Completeness of Statements

Finally we must prove soundness for statements. We get a valid denotation in a given state s, by applying the semantic function (from states to new states) on the state s. From every valid denotation we can create from our statement and the given state, we obtain a new state, s'. Completeness says there is a way of executing the expression in the operational semantics, with the state s, to get exactly the same state s' as the one we obtained from the semantic function:

```
stm-completeness : (s : State) → (S' : Stm) → (s' : State)
  → S S' s ≡ s' → seval s S' s'
```

We prove this by induction on the statements. For each statement, we must construct an element of the seval type, from the denotational semantic evaluation given.

Skip The base case will be for the skip statement, as this statement just returns the same state it received:

```
stm-completeness s skip .s refl = seval-skip s
```

Here we just construct an element of the operational semantics using the seval-skip constructor, which only requires the state we have given (and will just return) as a parameter.

Assignment Now we prove the inductive cases, starting with assignment. The inductive hypothesis will be or any S' in Stm, and s and s' in State, S S' $s \equiv s' \rightarrow$ seval s S' s'.

Our given denotational semantics is update s x (A a s) $\equiv s'$ and we need to create an element of seval s $(x := a)$ s'. To start our proof we use Agda's rewrite syntax, in which an equality is used to update the goal of the proof. We

want to change the state s' in the goal, so we switch the order of the equality using symmetry of equality (the Agda function sym), in addition to the rewrite function.

Now we want to find an element of seval s (x := a) (λ v $\rightarrow$ if eqVar v x then A a s else s v), because we know that update s x (A a s) $\equiv$ s', and the anonymous function is how update is defined:

```
stm-completeness s (x := a) s' p rewrite (sym p) =
  seval:= s x a (A a s) (aexp-completeness s a (A a s) refl)
```

Now we define our operational semantics using the seval:= constructor. Its parameters include the state s, a variable x, the arithmetic expression a being assigned to it and the value n of the expression a, which we obtain by executing the semantic function A on a and s. Finally we need an element of aeval s a which is obtained by the completeness of arithmetic expressions (which we have already proved).

Composition Next we prove completeness for composition of statements. To define the operational semantics, we use the seval-: constructor. Its parameters include the intial state s , the state obtained from executing the statement S_1 in s and the final state s'. It also has the statements S_1 and S_2. Finally we need an elements of seval s S_1 (S S_1 s) and seval (S S_1 s) S_2 s', which are constructed using the inductive hypothesis.

```
stm-completeness s (S₁ :' S₂) s' x = seval-: s (S S₁ s) s' S₁ S₂
  (stm-completeness s S₁ (S S₁ s) refl) (stm-completeness (S S₁ s) S₂ s' x)
```

If statement Next we prove completeness for the if statement. Because there is only one case for if statements in the semantic function S„ we now only need one case in the proof.

Our given denotational semantics is cond (B b) (S S_1) (S S_2) s ,and we need to create an element of seval s (If x then S_1 else S_2) s'. To start our proof we use Agda's rewrite syntax, in which an equality is used to update the goal of the proof. We want to change the state s' in the goal, so we switch the order of the equality using symmetry in addition to the rewrite function.

Now we want to find an element of seval s (If x then S_1 else S_2) (if B x s then S S_1 s else S S_2 s), because our new final state matches the definition of cond.

```
stm-completeness s (If x then S₁ else S₂) s' p rewrite (sym p)  =
  if-completeness s S₁ S₂ x (B x s) refl
    (bexp-completeness s x (B x s) refl)
    (stm-completeness s S₁ (S S₁ s) refl)
    (stm-completeness s S₂ (S S₂ s) refl)
```

Now we want two different results, as there are two possible operational semantics that can be constructed, depending on the value of x. Therefore we use the lemma if-completeness to express this.

Given our state s, the possible statements we can execute, S_1 and S_2, the boolean expression b and its value t, if-completeness says that given a proof that when the denotation of b is equal to t, we can construct the operational semantics for this expression, we can prove the completeness of the if statement.

In our use of if-completeness above, we give the value of t as the result of evaluating B x s, so this proof is just reduced to reflexivity. Then we define the element beval s b t using bexp-completeness as we already know this holds.

Now to complete the proof we need the elements of the operational semantics for each of the statements individually, which we get from the inductive hypothesis for statement completeness.

```
where
  if-completeness : (s : State) → (S₁ S₂ : Stm) → (b : Bexp) → (t : Bool)
    → B b s ≡ t
    → beval s b t
    → seval s S₁ (S S₁ s)
    → seval s S₂ (S S₂ s)
```

Now we have all of these we can get an element of:

```
→ seval s (If b then S₁ else S₂) (S (If b then S₁ else S₂) s)
```

In order to know we have this element we now need to actually define the lemma. We prove it by induction on the boolean value t, so there are two cases

p is the proof that B b $s \equiv t$, so by rewriting the goal with this, we can get the correct final state in our goal. When t is true, this will be an element of seval s (If b then S_1 else S_2) S S_1 s). Its parameters include the state s, the final state obtained by executing the statement S_1 in s, and the possible statements S_1 and S_2. we also need the boolean expression b, then the operational semantics for it and for the statement we are executing. These have been given as parameters to the lemma, so it is now simple to define:

```
  if-completeness s S₁ S₂ b true p be se1 se2 rewrite p =
    seval-if-true s (S S₁ s) S₁ S₂ b be se1
```

The case for false is defined in almost exactly the same way, except we use the seval-if-false constructor, the final state is obtained by executing the statement S_2 in s and the element of seval we want is now defined by $se2$:

```
  if-completeness s S₁ S₂ b false p be se1 se2 rewrite p =
    seval-if-false s (S S₂ s) S₁ S₂ b be se2
```

Now we have finished the lemma, so the completeness of if statements defintiely holds.

Repetition Finally we must prove completeness of repetition. In the syntax we defined two cases for repetition statements, so we must prove each individually. For zero iterations we have the base case, for which we must construct an element of seval-repeat-0. This just takes the state s and the statement S' as parameters, so is easy to define:

```
stm-completeness s (repeat zero do S') .s refl = seval-repeat-0 s S'
```

For the inductive case, we must define an element of seval-repeat. Its parameters include the the state s, the intermediate state obatained by evaluating S on S' in s and the final state of the repetition, s'. We also have the statement S' and , n, which is the one less than the number of iterations , because of our definition of natural numbers.

Then we need the seval elements that define the assumptions of the rule, which are for executing just S' and for the other n iterations. They are defined using the inductive hypothesis for statement completeness.

```
stm-completeness s (repeat suc n do S') s' p = seval-repeat s (S S' s) s' S' n
  (stm-completeness s S' (S S' s) refl) (stm-completeness (S S' s) (repeat n do S') s' p)
```

Now we have covered every case for statement completeness. This means that we have proved completeness for every possible statement, arithmetic expression and boolean expression in our language! As this is all the syntax we have defined our semantics with, everything that can is valid in the semantic function S can be executed in the operational semantics.

Chapter 6

Conclusion and Future Work

In this project I have constructed a small language and its operational and denotational semantics and proved that the semantics are both sound and complete.

Athough the language was small, the results I proved for the language are significant as it means any program that can be created in the language has a correct mathematical derivation.

Formalising the language in Agda was fun to do and increased my confidence in writing proofs, as I know they are correct because they pass Agda's correctness checks.

The hardest part of formalising the language was proving the soundness and completeness, but because the operational and denotational semantics were defined separately for each type of expression and statement, it made structuring the proofs much easier.

I started out with the book 'by [Nielson and Nielson, 2007] and wanted to formalise as much as possible, but formalising an entire book in the semester I had for the project was a little ambitious!

However, the next thing I could try with my language is to implement the structural operational semantics and prove that they are *equivalent* to the natural semantics I defined.

I could also extend the language to have more constructs such as methods or even more advanced executions like parallelism and non-determinism, as described in Chapter 3 of [Nielson and Nielson, 2007].

In the area of denotational semantics, I could explore using domains instead of sets to model my language. However Domain Theory is a huge subject in itself, and there are not many implementations of it in proof assistants, apart from [Benton et al., 2009] in Coq , so this would be a substantial project.

Overall I am happy with what I have achieved. I managed to achieve everything that I wanted to in my declaration. I did not implement extensions to the language, but that point was quite vague, as there are many possible extensions, as discussed above. However I have barely scratched the surface in the area of programming language semantics, so I now hope to study more complicated languages and their semantics, so I can implement more complex proofs. The skills I have learnt in this project will be very helpful in achieving this.

Chapter 7

Appendices

7.1 Agda helper functions

The following are simple functions we defined to help us with our proofs. First we give the imports, as this is a separate Agda file:

```
open import Relation.Binary.PropositionalEquality
open import Data.Nat
open import Data.Bool
open import dSyntax
```

Existential Quantifier The following data type represents the existential quantifer, ∃. It takes an element a of type A and a proposition which is true if the function P can be defined on a, as parameters.

```
data ∃'(A : Set) (P : A → Set) : Set where
  _,'_ : (a : A) → P a → ∃' A P
```

Relating two numbers equals checks two numbers are equal and gives the result as a boolean value:

```
equals : (x y : ℕ) → Bool
equals zero zero = true
equals zero (suc y) = false
equals (suc x) zero = false
equals (suc x) (suc y) = equals x y
```

leq checks if the first number is less than or equal to the second and returns a boolean value:

```
leq : (x y : ℕ) → Bool
leq zero y = true
leq (suc x) zero = false
leq (suc x) (suc y) = leq x y
```

geq checks if the first number is more than or equal to the second and returns a boolean value:

```
geq : (x y : ℕ) → Bool
geq zero zero = true
geq zero (suc y) = false
geq (suc x) zero = true
geq (suc x) (suc y) = geq x y
```

ge" is another version of greater than or equal to, implemented using the leq and not functions.

```
ge" : (x y : ℕ) → Bool
ge" x y =  not ( (leq x y) ∧ (not (equals x y)))
```

Both of these implementations of $\geq$ are equivalent to each other, which we can prove:

```
≥-equiv : (x y : ℕ) → geq x y ≡ ge" x y
≥-equiv zero zero = refl
≥-equiv zero (suc y) = refl
≥-equiv (suc x) zero = refl
≥-equiv (suc x) (suc y) = ≥-equiv x y
```

Logical Formulae or' maps two boolean values to their disjunction:

```
or' : Bool →  Bool → Bool
or' false false = false
or' _ _ = true
```

Equality of Variables eqVar checks if the first given variable is the same as the second, and is used to check if a variable is present in the state in our proofs.

```
eqVar : (x y : Var) → Bool
eqVar x' x' = true
eqVar y' y' = true
eqVar z' z' = true
eqVar _ _ = false
```

Equational Reasoning Here we define a syntax for equational reasoning, based on transitivity of equality. Given x, y and z which are elements of a type

A, if we have a proof that $x \equiv y$ and a proof $y \equiv z$, then we can prove that $x \equiv z$.

In Agda elements of the equality type can only be constructed with refl, which represents reflexivity of equality. Therefore this is all we need to define the syntax for equational reasoning.

```
_≡[_]_ : ∀ {A : Set} (x : A) {y z : A} → x ≡ y → y ≡ z → x ≡ z
x ≡[ refl ] refl = refl
infixr 2 _≡[_]_
```

_done is just syntactic sugar for reflexivity and can be used to end a proof:

```
_done : ∀ {A : Set} (x : A) → x ≡ x
x done = refl
infixr 2 _done
```

Substitution This is just the substitution function for changing the value of variables in a state, as we described in section 3.3:

```
update' : State → Var → ℕ → State
update' s x n = λ v → if eqVar v x then n else s v
```

7.2 Mini Project Declaration

The University of Birmingham

School of Computer Science

Second Semester Mini-Project: Declaration

This form is to be used to declare your choice of mini-project. Please complete all three sections and upload an electronic copy of the form to Canvas: https://canvas.bham.ac.uk/courses/16021

Deadline: 12 noon, 29 January 2016

1. Project Details

Name: Natalie Ravenhill

Student number: 1249790

Mini-project title: Formalising Operational and Denotational Semantics

Mini-project supervisor:

---

2. Project Description

The following questions should be answered in conjunction with a reading of your programme handbook.

Aim of mini-project	Understand how to form and use the operational and denotational semantics of programming languages by formalising the semantics of a small imperative language

Objectives to be achieved	• Construct a simple imperative language and its operational and denotational semantics. • Implement the language and its semantics in a proof assistant (Agda) to formally verify them both • Implement (and verify) extensions to the language if time permits.

Project management skills **Briefly explain how you will devise a management plan to allow your supervisor to evaluate your progress**	• A plan for each week that I have devised will be checked by my supervisor to ensure that it can reasonably be achieved in the time allocated to the project. • I hope that we will meet weekly, if it fits with both our timetables. If this is not possible, I will keep my supervisor informed by weekly email reports

Systematic literature skills **Briefly explain how you will find previous relevant work**	The categories of literature will be conference papers, journal articles, textbooks and theses. So far I have been using Google Scholar and the citations of textbooks and well known papers to find my resources. If Google Scholar does not prove adequate for good peer reviewed work, I will use other indexing sites such as Scopus and Web of Science. The search will have a wide range in the context of semantics, with operational and denotational semantics originating in the 1960s. Formalising the semantics in a proof assistant will have a much more recent search range, being restricted to the 21st century. The current version of Agda was created in 2007, so articles directly relating to Agda will be restricted to this year and after.

Communication skills **What communication skills will you practise during this mini-project?**	Written skills to be demonstrated by report. Oral skills demonstrated by verbal reports to supervisor and potentially giving a talk to other students/members of the Theory group at the "Lab Lunch" seminar.

3. Project Ethics Self-Assessment Form

Does the research involve contact with NHS staff or patients?
No

Does the research involve animals?
No

Will any of the research be conducted overseas?
No

Will any of the data cross international boarders?
No

Will the research project involve humans as subjects of the research, including as participants in a requirements gathering or evaluation exercise?
No

Are the results of the research project likely to expose any person to physical or psychological harm?
No

Will you have access to personal information that allows you to identify individuals, or to corporate or company confidential information (that is not covered by confidentiality terms within an agreement or by a separate confidentiality agreement)?
No

Does the research project present a significant risk to the environment or society?
No

Are there any ethical issues raised by this research project that in the opinion of the PI require further ethical review?
No

IF the only box ticked was about human subjects:

*** does the project have the potential to cause stress or anxiety in the people you are involving - e.g. does it address potentially sensitive issues of health, death, religion, self-worth, financial security or other such issues?**
*** does the project involve people under 18?**
*** does the project involve a lack of consent or uninformed consent?**
*** does the project involve misleading the subjects in any way?**

If the project's involvement of people relates to straightforward information gathering,

requirements specification, or simple usability testing, and the answer to all the above questions is NO, then further ethical review is not needed.

If any of the above questions is answered YES, or you are unsure if further review is needed (the first point is usually a good indicator - may cause stress or anxiety) then you should refer it for review.

Further review will involve the School Ethics Officer meeting with the supervisor and ideally the student, reviewing the project, and suggesting any procedures necessary to ensure ethical compliance.

DECLARATION

By submitting this form, I declare that the questions above have been answered truthfully and to the best of my knowledge and belief, and that I take full responsibility for these responses. I undertake to observe ethical principles throughout the research project and to report any changes that affect the ethics of the project to the University Ethical Review Committee for review.

7.3 Information Searching

7.3.1 Parameters of literature search

Forms of literature to be retrieved

The important forms are firstly conference papers and journal articles, as many of the important developments in the field have been communicated in this way, particularly at conferences in programming languages research such as POPL.

I will also consider PhD theses, as there are many researchers in programming language semantics whose students also work in this area and produce notable original work in their theses. They are often supported with extensive background material that helps give a overview of the field.

Finally I will look at books, as there are several well known textbooks such as [Gunter, 1992] and [Nielson and Nielson, 2007], that provide a useful reference for understanding both of the areas of semantics I am studying.

Geographical/language coverage

Important work will be from many areas of the world. For example there are research groups working in the UK, France and the US, which is where most of my background reading has originated, as the foundations of Denotational and Operational Semantics were created mostly by researchers in the UK.

The work in formalising programming language semantics is often done using the Coq proof assistant, which originated from research at INRIA in France. INRIA is also the origin of the CompCert compiler, and work on the JSCert project has involved researchers there too, as well as from Imperial College in the UK.

However, I do not need to restrict my search to just these countries, as progrmaming language semantics is a wide area and is being studied all over the world.

The only language covered is English, as unfortunately I am unable to understand papers in any other language, but fortunately I have yet to come across any relevant papers that were not written in English, so this has not been as issue.

Retrospective coverage and currency

It is sufficient to search retrospectively for 50 years for denotational semantics, as it originated in the 1960s. For operational semantics, we can use a more recent search, of 30 years, as it originated in the 1980s.

For formalsing semantics proofs in proof assistants, we can look for even more recent papers. Coq was created in 1989, so we only have to look back around 25 years. Agda was created in 2007, so we can look back less than 10 years for Agda formalisations.

I use machine based services to do my literature search as they are suitable for my needs. Every paper I have needed to access is available from online databases and university library online resources.

7.3.2 Appropriate search tools

ACM Digital Library

ACM Digital Library is a database of all publications by the Association for Computing Machinery (ACM) including articles, magazines and conference proceedings. It does not include PhD theses. I used ACM Digital Library `http://dl.acm.org` to search for conference and journal papers. It has the retrospective coverage and currency required.

Google Scholar

Google Scholar is part of the search engine that is exclusively used for searching academic documents. I used to search for conference and journal papers.

7.3.3 Search statements

The search statements used will be based on: ‘

"operational semantics"
"denotational semantics"
"semantics" AND "coq"
"semantics" AND "agda"

This may need to be refined in the number of recalled items is too large. Keywords from items that have been found will be reviewed in order to refine the set of keywords used.

7.3.4 Brief evaluation of the search

ACM Digital Library

Searching for the phrase "operational semantics" in papers published since 1980 gave 749 results, of which 578 were conference proceedings, 207 were newsletter articles, 140 were journal articles and 1 was a book. The conference articles are

generally republished in the newsletters, so this is why we have more results when we split them into their different forms.

Searching for the phrase "denotational semantics" in papers published since 1960 gave 346 results, of which 226 were conference proceedings, 93 were newsletter articles, 74 were journal articles and 1 was a book. The conference articles are generally republished in the newsletters, so this is why we have more results when we split them into their different forms.

Searching for the phrases "semantics" and "coq" on ACM DL, for papers published since 1989, gaves 159 relevant results, of which of which 141 were conference proceedings, 79 were newsletter articles and 9 were journal articles.

Searching for "semantics" and "agda" since 2006 gave 28 relevant results, of which. 28 were preceedings and 15 were newsletter articles.

The search terms with actual proof assitants are much more specific, so return less results and more focused results. However, they still return too many results, so we can reorder our results by relevance and/or citation count to get the 10 most popular and relevant papers, for a good overview.

Google Scholar

Searching for the phrase "denotational semantics" since 1960 in Google Scholar gave us 16,400 results and "operational semantics" since 1980 gave us 40,100 results. Therefore these search terms are much too broad to get any useful information

Searching for "semantics" and "agda" since 2006 gave us 1220 results and searching for "semantics" and "coq" since 1989 gave us 8550 results. These result sets are also very large, so we can take the most cited results. As Google Scholar does not give much information about the origin of articles, it is harder to tell their forms. Therefore, we assume the ones with the most citations are from peer reviewed conferences and journals, unless stated otherwise.

Bibliography

J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. Report on the Algorithmic Language ALGOL 60. *Commun. ACM*, 3(5):299–314, May 1960. ISSN 0001-0782. doi: 10.1145/367236.367262. URL `http://doi.acm.org/10.1145/367236.367262`.

J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. Revised report on the algorithm language ALGOL 60. *Commun. ACM*, 6(1):1–17, January 1963. ISSN 0001-0782. doi: 10.1145/366193.366201. URL `http://doi.acm.org/10.1145/366193.366201`.

Nick Benton, Andrew Kennedy, and Carsten Varming. Some domain theory and denotational semantics in coq. In *Theorem Proving in Higher Order Logics*, pages 115–130. Springer, 2009.

Yves Bertot, Pierre Casteran, Gerard (informaticien) Huet, and Christine Paulin-Mohring. *Interactive theorem proving and program development : Coq'Art : the calculus of inductive constructions*. Texts in theoretical computer science. Springer, Berlin, New York, 2004. ISBN 978-3-540-20854-9. URL `http://opac.inria.fr/record=b1101046`.

Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. A trusted mechanised JavaScript specification. *ACM SIGPLAN Notices*, 49(1):87–100, 2014.

Dominique Clément, Thierry Despeyroux, Gilles Kahn, and Joëlle Despeyroux. A Simple Applicative Language: Mini-ML. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, LFP '86, pages 13–27, New York, NY, USA, 1986. ACM. ISBN 0-89791-200-4. doi: 10.1145/319838.319847. URL `http://doi.acm.org/10.1145/319838.319847`.

Nils Anders Danielsson. A formalisation of a dependently typed language as an inductive-recursive family. In *Types for Proofs and Programs*, pages 93–109. Springer, 2006.

Nils Anders Danielsson. Operational semantics using the partiality monad. In *ACM SIGPLAN Notices*, volume 47, pages 127–138. ACM, 2012.

ECMA International. *Standard ECMA-262 - ECMAScript Language Specification*. 5.1 edition, 2011. URL `http://www.ecma-international.org/publications/standards/Ecma-262.htm`.

Daniele Filaretti and Sergio Maffeis. An executable formal semantics of PHP. In *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings*, pages 567–592, 2014. URL `http://dx.doi.org/10.1007/978-3-662-44202-9_23`.

Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The essence of JavaScript. In *ECOOP 2010–Object-Oriented Programming*, pages 126–150. Springer, 2010.

Carl A Gunter. *Semantics of programming languages: structures and techniques*. MIT press, 1992.

H. Hüttel. *Transitions and Trees: An Introduction to Structural Operational Semantics*. Cambridge University Press, 2010. ISBN 9781139788595. URL `https://books.google.co.uk/books?id=f9zmrShQj3YC`.

Atshushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *ACM SIGPLAN Notices*, volume 34, pages 132–146. ACM, 1999.

Alan Jeffrey. Dependently Typed Web Client Applications. In *Practical Aspects of Declarative Languages*, pages 228–243. Springer, 2013.

Donald E. Knuth. The remaining trouble spots in ALGOL 60. *Commun. ACM*, 10(10):611–618, oct 1967. ISSN 0001-0782. doi: 10.1145/363717.363743. URL `http://doi.acm.org/10.1145/363717.363743`.

Peter J Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.

Peter J Landin. Correspondence between ALGOL 60 and Church's Lambda-notation: part i. *Communications of the ACM*, 8(2):89–101, 1965.

Xavier Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *33rd ACM symposium on Principles of Programming Languages*, pages 42–54. ACM Press, 2006. URL `http://gallium.inria.fr/~xleroy/publi/compiler-certif.pdf`.

Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

Sergio Maffeis, John C Mitchell, and Ankur Taly. An operational semantics for JavaScript. In *Programming languages and systems*, pages 307–325. Springer, 2008.

Alfio Martini. Programming language semantics with Isabelle/HOL. In *Proceedings of the 2013 2Nd Workshop-School on Theoretical Computer Science*, WEIT '13, pages 14–21, Washington, DC, USA, 2013. IEEE Computer Society. ISBN 978-1-4799-3057-9. doi: 10.1109/WEIT.2013.29. URL `http://dx.doi.org/10.1109/WEIT.2013.29`.

John McCarthy. Recursive functions of symbolic expressions and their computation by machinePart I. *Communications of the ACM*, 3(4):184–195, 1960.

Robin Milner. Models of LCF. Technical report, Stanford, CA, USA, 1973.

Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: An Appetizer (Undergraduate Topics in Computer Science)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007. ISBN 1846286913.

Ulf Norell. Dependently typed programming in Agda. In *Advanced Functional Programming*, pages 230–266. Springer, 2009.

Michael Norrish. *C formalised in HOL*. Number 453. 1998.

Gordon D. Plotkin. LCF considered as a programming language. *Theoretical computer science*, 5(3):223–255, 1977.

Gordon D Plotkin. A structural approach to operational semantics. 1981.

Gordon D Plotkin. The origins of structural operational semantics. *The Journal of Logic and Algebraic Programming*, 60-61:3 – 15, 2004. ISSN 1567-8326. doi: http://dx.doi.org/10.1016/j.jlap.2004.03.009. URL `http://www.sciencedirect.com/science/article/pii/S1567832604000268`. Structural Operational Semantics.

Dana Scott. *Outline of a mathematical theory of computation*. Oxford University Computing Laboratory, Programming Research Group Oxford, UK, 1970.

Dana S. Scott. A type-theoretical alternative to ISWIM, CUCH, OWHY. *Theor. Comput. Sci.*, 121(1-2):411–440, December 1993. ISSN 0304-3975. doi: 10.1016/0304-3975(93)90095-B. URL `http://dx.doi.org/10.1016/0304-3975(93)90095-B`.

Dana S Scott and Christopher Strachey. *Toward a mathematical semantics for computer languages*, volume 1. Oxford University Computing Laboratory, Programming Research Group, 1971.