Improving Trust and Security through First-Class Certificates on Probabilistic Software Behaviour

School of Computer Science

University of St Andrews

Thomas Ekström Hansen
150015673

Supervisor:
Dr. Edwin Brady

12 April 2019

Declaration

I declare that the material submitted for assessment is my own work except where credit is explicitly given to others by citation or acknowledgement. This work was performed during the current academic year except where otherwise stated. The main text of this project report is $14,200$ words long, including project specification and plan. In submitting this project report to the University of St Andrews, I give permission for it to be made available for use in accordance with the regulations of the University Library. I also give permission for the report to be made available on the Web, for this work to be used in research within the University of St Andrews, and for any software to be released on an open source basis. I retain the copyright in this work, and ownership of any resulting intellectual property.

Abstract

This project extends and evaluates work done by the TeamPlay [11] project in terms of constructing a framework for reasoning about extra-functional properties of programs designed for embedded systems. The framework presented works by annotating existing C code, measuring the compiled code on hardware to get timing and energy data, using the collected data in combination with the annotated code to create a model in an Embedded Domain Specific Language (EDSL) based in the Idris programming language, and then uses Idris's built-in proof system to create a certificate that the specified properties do or do not hold. As part of the evaluation, this project presents a collection of programs and models to demonstrate the correctness and usefulness of the framework.

Contents

Chapter 1

Introduction

For certain system-critical or embedded systems with limited resources, it is often desirable to be able to express and reason about extra-functional properties of programs like time taken or energy used. What is even more desirable would be if the extra-functional properties could be proven to hold or not. This project aims to allow programmers to do this through the use of Embedded Domain Specific Languages (EDSLs) and the Idris programming language and type system.

Embedded systems are becoming increasingly common. From wireless handsets, smart cards, and routers to scientific sensors, health monitors, and satellites, embedded systems are an increasing part of our daily lives, and control more and more essential parts of it. With more devices connected to the internet than people since approximately 2009 and an estimated 50 billion connected devices by 2020 [16], there is an increasing need to provide certain guarantees for these systems. Since embedded systems often have limited resources, e.g. battery or processing power, being able to reason about how much time and energy programs take would be very useful for the programmers of embedded systems. One way to reason about these are EDSLs.

A DSL is a programming language which is designed for a specific purpose. The benefit to DSLs is that a programmer should be able to much quicker develop the program they want, as the DSL is intended specifically for that task/domain, than they would using a general purpose language [18]. However, since designing a language is difficult and time-consuming building a language *on top of* another language allows us to use constructs from the "base" language (e.g. operators, variable declarations) and use them in our DSL, thereby speeding up its development [19].

DSLs can be used for various purposes. From network protocols [3] to concurrency or systems programming [9, 10]. Using IDRIS as the host language for an EDSL means that the resulting DSL are well-typed. This means we can use the IDRIS type-checker and built-in proof system to prove that our DSL is correct and that the timing and energy properties hold. The built-in proof system is based on the IVOR proof engine [6, 7]. Using existing work by the international TEAMPLAY project [11], which is part of the EU Horizon 2020 funding programme, this project explores the ability to express and prove timing and energy properties using EDSLs (specifically the TEAMPLAY framework) and the IDRIS programming language [7].

The primary objectives of this project are to design a formal framework for assumptions about extra-functional properties of programs by extending the TEAMPLAY framework to have a complete set of operators; write basic C programs which capture that the operators in the framework function as intended; come up with C programs which use the framework beyond just the operators; and to evaluate how good the TEAMPLAY project's existing framework is for these purposes.

The secondary objectives of this project are to model "real-world" C programs using the framework, and in doing so come up with a collection of programs which showcase how the framework can be used.

The ternary objectives for this project are to explore the modelling of more advanced coding concepts like nested `for` loops, to automate the translation from C code to the framework, and to support open-ended assertions where the value of one or more of the variables are unknown.

Chapter 2

Context Survey

With numerous examples of software-based critical faults, e.g. the malfunction of NASA's "Demonstration of Autonomous Rendezvous Technology (DART)" in 2005 [15], the erroneous administration of radiation doses at Panama's National Oncology Institute in 2001 [5], or the infamous explosion of the Ariane 5 flight 501 in 1996 [4], reasoning about and asserting extra-functional properties of software is an important problem where unforeseen circumstances or uncaught edge-cases can have disastrous consequences. Some work on modelling extra-functional properties of programs has already been done. Model-checking, Domain Specific Languages (DSLs) using Haskell and IDRIS, and also simulators are all examples of tools used to verify software.

2.1 Haskell-based DSLs

Feldspar [2] is a language for reasoning about digital signal processing (DSP) algorithms and is built on Haskell. Feldspar's main building blocks are `value`, `ifThenElse`, `while`, and `parallel` [2]. These capture the building-blocks of simple C programs and introduces its own unique function `parallel` for describing parts of the program where each 'sub-computation' can be done independently [2]. Feldspar's aim is to help build sound DSP algorithms and then use the high-level description to generate fast, low-level C code [2].

Hume [17] is a different DSL which focuses on modelling embedded systems. In Hume, the programmer can reason about `boxes`, `wires`, `devices`, `datatypes`, `functions`, and `exceptions` [17]. Through these constructs, the programmer can model embedded systems as coarse- or fine-grained as they want, and Hume then estimates the stack, heap, and space/time requirements [17].

2.2 Idris-based DSLs

Several IDRIS-based DSLs exist and demonstrate the benefits of using IDRIS for implementing the DSLs. State-aware Embedded DSLs (EDSLs) for systems programming [10] allows the programmer to reason about network transfer or correct file-access. This lets them construct safe, low-level protocols in a higher-level language, and the type-checker can help them spot problems by construction [10] rather than by trial-and-error. Another example uses IDRIS for concurrent programming [9]. By representing the state of a concurrent program and the locking and unlocking operations as types, the resulting EDSL allows the programmer to reason about the thread-safety of their program when constructing it, rather than when testing it. DSLs can also be used to create correct-by-construction network protocols where the embedded types ensure that the protocol operates correctly as long as it type-checks [3].

2.3 Non-DSL Approaches

Alternatively to DSLs, a simulator could be used to measure extra-functional properties. Simulators model the hardware, so virtual readings of time and energy can be taken when running the program. The benefit is that there is no need to have the actual hardware and, depending on the power of the simulator and computer being used to run it, many different types of hardware can be simulated.

Another alternative could be to use a model-checker. Tools like UPPAAL [23] or PRISM [22] verify and validate timing properties of programs given as Petri Nets or timed automata. These tools are more general and can be applied to various programs, not just those designed for embedded systems. This makes them very powerful, but also increases their complexity.

The [MC]SQUARE model-checker/simulator hybrid is specifically aimed at C code for embedded systems [25]. It works similar to a simulator to construct its models, i.e. it compiles and analyses the assembler code, and from this checks that the program's functionality is correct.

2.4 Limitations of the existing work

The Haskell-based DSLs do not have a built-in, straightforward way of producing proofs for the written programs. Although existing research has used Haskell in combination with proof-assistants [21], this focuses on mathematical theories and proofs rather than proofs for extra-functional properties of programs. Having proofs for these is useful, as it means the programmer has a formal validation of the properties rather than a measurement or probable guarantee.

Simulation mainly focus on the designing the architectures themselves [1, 24] or target the energy consumption of the entire embedded system [26] rather than a specific program or piece of the software and the readings gotten are based on a simulation, not the real hardware which could behave differently. Simulation is also orders of magnitude slower than running the actual hardware, even when sacrifices are made in terms of accuracy (for example, using a functional simulator instead of a cycle-accurate simulator).

Model-checkers are slow due to the fact that they exhaustively explore the state-space of the given model/program. A well-known problem for model-checkers is the state-explosion problem [13, 14, 27]: Due to exhaustively searching the state-space, only small programs can be modelled as adding states exponentially grows the search space [13, 14], leading to the model-checker needing more memory than the computer can provide. One workaround is to break the program up into smaller parts which can then be modelled. However, each of these smaller parts are then susceptible to another limiting factor of model-checkers: They require the programmers to manually model the software or part of program to analyse. Since this has to be done in the model-checking software, this often implies learning a new, likely unfamiliar, tool. As such, getting from the program to the model is slow and prone to human error, even more so when multiple models are required for parts of one program.

2.5 Benefits of using Idris

Using a strongly-typed language like IDRIS which supports user-defined types and dependent types where the type depends on a value or property means that when we have modelled the problem as a collection of dependent types, we can guarantee the program works as intended through the type-checker. If our model of the problem is incorrect, the type-checker will prevent the model from even compiling. This lets us be very specific about which parts of our framework do what, and be certain that they *do* do those things. Furthermore, the built-in proof system allows for conditions in the resulting models to be formally guaranteed to hold without having to bring in an external tool. We can design our framework around supporting provable properties. Existing research using IDRIS [3, 9, 10] show that these features can be leveraged to create powerful prototyping and formal verification methods. However, the existing work is not in the area of time and energy guarantees.

Chapter 3

Example use-case

An example of a real-world use-case would be to assert that encrypting something using the AES encryption method does not require more than a set amount of energy. This could be to guarantee that the encryption will not heavily impact the embedded device's battery life. For this, I used an implementation of AES written in C [20]. The parts of the code which would be interesting to annotate and assert are the parts that do the AES cipher encryption:

Listing 3.1: The part of the AES encryption cipher we are interested in: the rounds

```
⋮
add_round_key(state, w, 0);

for (r = 1; r < Nr; r++) {
    sub_bytes(state);
    shift_rows(state);
    mix_columns(state);
    add_round_key(state, w, r);
}

sub_bytes(state);
shift_rows(state);
add_round_key(state, w, Nr);
⋮
```

This example will evolve throughout this dissertation, as more parts of the project are presented and explained.

Chapter 4

Idris and Dependent Types

This chapter aims to give the reader enough of an overview of IDRIS and dependent types to be able to understand the work done in the later chapters. A complete explanation of IDRIS is beyond the scope of this chapter and project. The chapter borrows a lot of examples and explanations from the IDRIS book ("Type-Driven Development with Idris") [8].

4.1 Types

Listing 4.1: In IDRIS, the type of a function is specified using ':'

```
anInt : Int
anInt = 10

aString : String
aString = "foo"

aBool : Bool
aBool = False
```

Types classify values. In programming, we often come across types like `Int`, `String`, or `Bool`, which could be values like `10`, `"foo"`, or `False` respectively.

Listing 4.2: Mismatching types

```
aString : String
aString = 10


-- Compiler error --

  |
2 | aString = 10
  |           ~~
When checking right hand side of aString with expected type
String

String is not a numeric type
```

Types in IDRIS are checked at compile time, meaning that if the types of the functions in a program do not match (for example, passing an `Int` where a `String` is required) then the program will not compile and the compiler will give a type-error.

Listing 4.3: Values are not automatically cast

```
anInt : Int
anInt = 10

half : Double -> Double
half x = x / 2


> half anInt
(input):1:1-10:When checking an application of function Main.half:
        Type mismatch between
                Int (Type of anInt)
        and
                Double (Expected type)
```

IDRIS is strongly typed, so the compiler will not automatically cast values or parameters. This means you cannot pass an `Int` to a function which requires a `Double` and have it automatically work, because `Int` is a different type, a different 'category' from `Double`.

4.2 Type Variables and Dependent Types

A different example of a type is a list of values. In languages like Java or Python, lists are parametrised over a type.

Listing 4.4: The types of different lists in IDRIS

```
[1, 2, 3, 4, 5] : List Integer

["a", "b", "c"] : List String

[True, False]   : List Bool
```

In IDRIS this is still the case. You can have lists of `String`, lists of `Int`, or lists of `Bool` and these are different types. In general, you can have any `List elem` where `elem` is a *type variable* representing the type of the elements of the list. IDRIS provides an even more specific type of lists: `Vect`

Listing 4.5: Example `Vect` types

```
[1, 2, 3, 4, 5] : Vect 5 Int

["a", "b", "c"] : Vect 3 String

[True, False]   : Vect 2 Bool
```

A `Vect`, short for "vector", is a list with a specific length. In general, you can have any `Vect n elem`. Here, `elem` is the same as in `List`, and `n` is the length of the list. Since the value of `n` *depends* on the number of elements in the list, we refer to types like `Vect n elem` as *dependent types* because its precise type depends on other values. Another example of a dependent type is `Matrix m n elem`, i.e. a matrix of `m` rows and `n` columns, with elements of type `elem`.

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix} \qquad \begin{pmatrix} \text{``1''} & \text{``2''} & \text{``3''} \\ \text{``4''} & \text{``5''} & \text{``6''} \end{pmatrix}$$

Here, the first matrix would have type `Matrix 3 2 Int` and the second matrix would have type `Matrix 2 3 String`.

Listing 4.6: Types are *first-class*

```
ty : Type
ty = Bool

StringOrInt : Bool -> Type
StringOrInt x = case x of
                    True => Int
                    False => String
```

In IDRIS, types are *first-class*. This means that they can be used without any restrictions: we can assign them to variables (`ty`), or have functions return them (`StringOrInt`). Types are used everywhere, in particular when defining functions.

4.3 Functions

In Listing 4.3 I defined a function `half`. Functions in IDRIS consist of a *type declaration* and a *function definition*.

Listing 4.7: The `half` function from Listing 4.3

```
half : Double -> Double
half x = x / 2
```

The first line of the `half` function is its type declaration and the second line is the function definition. These can each be broken down into further parts. A function definition consists of:

- a function name, `half`
- a function type, `Double -> Double`

A function definition is a number of equations with:

- a right hand side, `half x`
- a left hand side, `x / 2`

Listing 4.8: A different definition of `half`

```
half' : Double -> Double
half' 0.0 = 0.0
half' n = n / 2
```

In this different definition of `half`, `half'`, there are multiple functions definitions. A function can have varying definitions depending on its input. This is known as *pattern matching* and is used extensively in IDRIS. Here, it is not massively useful, as all it is doing is not bothering with the division, if the input is 0.

Listing 4.9: A function with a function as an argument

```
twice : (a -> a) -> a -> a
twice f x = f (f x)
```

Functions in IDRIS are first-class, so we can use them as arguments to other functions. The `twice` function's first argument is a function `f` which takes something of type `a` and produces something of type `a`. The second argument `x` is then something of type `a`. By not specifying a type and instead using a type variable, `twice` can be used on any type, as long as `x` has the same type as in the input to `f` and that `f` does not change the type of `x`.

In addition to being first-class, functions in IDRIS are *pure* meaning that they do not have side effects (e.g. they do not do console I/O and they do not throw exceptions). This in turn means that for the same input, a function will always give the same output [8].

4.4 Local variables

In function declarations we might not want to clutter the equation and instead have local variables which clearly describe what the various parts are. In IDRIS you can use the keywords '`let`' and '`in`' to define such variables.

Listing 4.10: A program which returns the longer of two given words

```
longer : String -> String -> String
longer word1 word2 =
  let
    len1 = length word1
    len2 = length word2
  in
    if len1 > len2 then word1 else word2
```

The variables `len1` and `len2` exist only in the statement after the `in` keyword. If we were to write the program in Listing 4.10 without a `let/in` definition it would look like this:

Listing 4.11: Listing 4.10 without using `let/in`

```
longer : String -> String -> String
longer word1 word2 =
  if (length word1) > (length word2) then word1 else word2
```

This is still mostly readable, but less so than Listing 4.10.

4.5 Data Types

In addition to the given types, we can define our own data types. This is useful when we want to capture an aspect of the program we are writing which is likely to not be pre-defined in the IDRIS prelude. Data types are defined using the `data` keyword. A data type is defined by a type constructor and one or more data constructors [8]. Some of the included types are defined as data types, for example `List`.

Listing 4.12: `List` as defined in the IDRIS prelude

```
data List : (elem : Type) -> Type where
    Nil  : List elem
    (::) : (x : elem) -> (xs : List elem) -> List elem
```

A `List` of elements of type `Type` can be constructed by either constructing the empty list `Nil`, or an `x` of type `elem` (the type of all elements in the list) followed by a list of more elements of the same type.

Listing 4.13: `Bool` as defined in the IDRIS prelude

```
data Bool = False
          | True
```

Data types can be constructed with either equals and vertical bars (see the definition of `Bool` just above) or as functions, like in the definition of `List`. They are mostly interchangeable, but the `data ... -> Type where` syntax is more flexible and general, at the cost of also being more verbose.

Data type definitions can also be recursive.

4.6 Natural Numbers

Listing 4.14: Natural numbers as defined in the IDRIS prelude

```
data Nat = Z | S Nat
```

Natural numbers in IDRIS are also a data record. They base case is the constant 0 (`Z` in Idris) and the other constructor is the *successor function* `S`. From these two, all the natural numbers can be constructed:

$$\begin{array}{rl} \texttt{Z} & \mapsto 0 \\ \texttt{S Z} & \mapsto 1 \\ \texttt{S (S Z)} & \mapsto 2 \\ \text{etc} & \\ \vdots & \end{array}$$

The benefit of this is that it allows us to pattern match on numbers.

Listing 4.15: Pattern matching on `Nat`s

```
natFact : Nat -> Nat
natFact Z = 1
natFact (S k) = (S k) * fact k
```

By pattern matching on the possible constructors for `Nat`, we have captured the base case, and the recursive step for calculating the factorial of a number using the `natFact` function. Pattern matching also helps in terms of decidability.

4.7 The `mutual` declaration

Occasionally, there might be a situation where we need to define functions or types which rely on each other. This is problematic because in IDRIS, something must be defined before you can use it. The `mutual` declaration allows us to explicitly define things which depend on each other, thereby 'bypassing' the declaration order.

Listing 4.16: Mutually declaring `odd` and `even`

```
mutual
    odd : Nat -> Bool
    odd Z     = False
    odd (S k) = even k

    even : Nat -> Bool
    odd Z     = True
    odd (S k) = odd k
```

If we wanted to define two functions `odd` and `even` which determined the parity of a natural number, we could define them by saying that zero is even, and a number bigger than that has the opposite parity of its predecessor. However, in doing so, `odd` relies on the existence of the `even` function and vice-versa, so both would have to be defined before each other. Since it is impossible to define two things at the same time, we use a `mutual` block to indicate that the functions rely on each other and that only when the entire `mutual` block has been defined will the functions work correctly.

4.8 `Maybe` – Handling uncertainty

Listing 4.17: The `Maybe` data type

```
data Maybe : (a : Type) -> Type where
    Nothing : Maybe a
    Just : (x : a) -> Maybe a
```

The `Maybe` data type captures the concept of failure or uncertainty of a function. Either, the function will have `Just` a value `x` or `Nothing`. We could use this to implement taking the nth element from a list.

Listing 4.18: List indexing using `Maybe`

```
getIndex : Nat -> List a -> Maybe a
getIndex _ Nil = Nothing
getIndex Z (x :: xs) = Just x
getIndex (S k) (x :: xs) = getIndex k xs
```

Since we could either be given the empty list or run out of bounds, we use `Maybe` to reflect that the index may not be valid. If the index exists, we return `Just x` where `x` is the nth element, and if we run out of bounds we return `Nothing`.

Chapter 5

Design

The first section (5.1) focuses on describing existing work done by the TeamPlay project [11], apart from the annotation of the AES example which was done by me. The remaining sections (5.2, 5.3, and 5.4) describe extensions of the TeamPlay system, implemented by me. In section 5.2, I will also cover the more advanced Idris concepts that I had to familiarise myself with in order to be able to implement the extensions I did. Similar to Chapter 4, the explanations and examples given when covering the concepts borrow a lot from the Idris book [8].

5.1 The TeamPlay Project

5.1.1 Assertions and Contracts

When writing C programs for embedded systems, the programmer might want to capture extra-functional properties like the time taken or the energy consumed. Furthermore, the programmer may want to have provable guarantees that these properties of the program hold. This is done through contracts which can be proven using the Idris proof system.

The Contract Specification Language (CSL) defined by the TeamPlay project [11] lets the programmer annotate existing C code to express properties like the worst time cost of a loop or function, using `__teamplay_worst_time`, and assert these, using `__teamplay_assert`. This results in a program whose extra-functional properties can be certified to hold. Since the CSL functions adhere to the C99 function naming scheme, the programmer does not need to change tools, libraries, compiler, or learn a new language. Instead, the annotations seamlessly integrate with the existing code, without disturbing its operation in any way. This means we can compile the code and measure it on the embedded systems to gain data concerning its time and energy properties. Using the data and the annotated

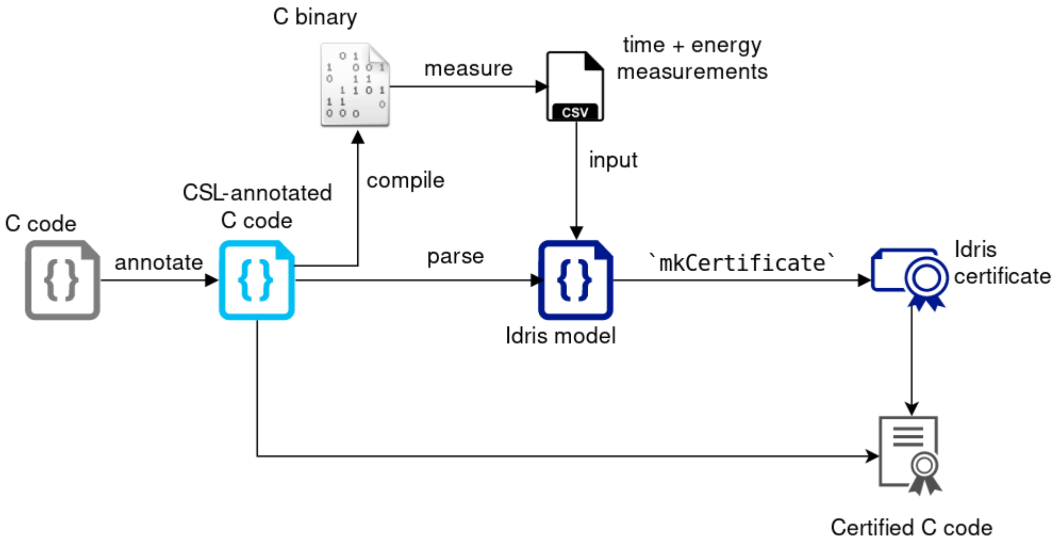

Figure 5.1: The process of creating a certified C program

C code as input to a parser, the annotations are extracted and used to construct a model in an IDRIS-based DSL. Since the parser is currently work in progress, the annotations are assumed to exist and work.

The resulting model is passed through the `mkCertificate` function which constructs a contract/certificate specifying which properties do or do not hold, by using the IDRIS proof system. The combination of this certificate and the annotated C code forms 'certified C code', i.e. C code whose extra-functional properties have been proven to hold.

C operator	IDRIS-based **DSL equivalent**
`==`	`Eq`
`!=`	`NEq`
`<=`	`LTE`
`<`	`LT`
`>=`	`GTE`
`>`	`GT`
`&&`	`And`
`||`	`Or`
`!`	`Not`

Table 5.1: C to DSL mappings

If we were to look at the AES example again, we could express that the rounds should take at most 50 energy units by annotating it as follows:

Listing 5.1: Annotating the AES code to express an energy requirement

```
⋮
__teamplay_worst_energy(addRoundKey0);
add_round_key(state, w, 0);

for (r = 1; r < Nr; r++) {
    __teamplay_worst_energy_acc(subBytesAcc);
    sub_bytes(state);

    __teamplay_worst_energy_acc(shiftRowsAcc);
    shift_rows(state);

    __teamplay_worst_energy_acc(mixColumnsAcc);
    mix_columns(state);

    __teamplay_worst_energy_acc(addRoundKeyAcc);
    add_round_key(state, w, r);
}

__teamplay_worst_energy(subBytes);
sub_bytes(state);

__teamplay_worst_energy(shiftRows);
shift_rows(state);

__teamplay_worst_energy(addRoundKeyNr);
add_round_key(state, w, Nr);

__teamplay_assert(addRoundKey0 + subBytesAcc +
                  shiftRowsAcc + mixColumnsAcc +
                  addRoundKeyAcc + subBytes +
                  shiftRows + addRoundKeyNr
                  <= 50)
⋮
```

The annotations describe the names of the variables that the energy measurements will be stored in, and some of them express that this measurement is accumulated across loops. At the end, the sum of the variables is required to be less than or equal to a limit, 50.

Listing 5.2: The `Assertion` data type

```
data Assertion   : Type where
    MkAssertion : BooleanExpression -> Assertion
```

Contracts/Certificates consists of one or more `Assertion`. These are created through the `MkAssertion` constructor. It takes a `BooleanExpression` and returns an `Assertion` based on it. `BooleanExpressions` are created by applying operators to `NumericExpressions` or existing `BooleanExpressions`.

5.1.2 Numeric Operators and Numeric Expressions

The numeric operators all take two arguments, their evaluation, a proof concerning what the operator evaluates to, and returns a `BooleanExpression`. The operands are `NumericExpressions`. A `NumericExpression` is defined by the following grammar:

⟨*NumericExpression*⟩	::=	; a literal
		‘`Lit`’ ⟨*digit*⟩$^+$
	|	; a variable
		‘`Var`’ ⟨*identifier*⟩
	|	; a parenthesised expression
		‘`NParen`’ ⟨*NumericExpression*⟩
	|	; addition
		‘`Plus`’ ⟨*NumericExpression*⟩ ⟨*NumericExpression*⟩
	|	; subtraction
		‘`Sub`’ ⟨*NumericExpression*⟩ ⟨*NumericExpression*⟩
	|	; multiplication
		‘`Mul`’ ⟨*NumericExpression*⟩ ⟨*NumericExpression*⟩
	|	; division
		‘`Div`’ ⟨*NumericExpression*⟩ ⟨*NumericExpression*⟩
	|	; modulo
		‘`Mod`’ ⟨*NumericExpression*⟩ ⟨*NumericExpression*⟩

`NumericExpressions` are evaluated given an environment `Env`. An environment is a mapping from variables to natural numbers, which is used to look up the value of a variable when given its identifier. The idea is that this would be provided by the CSV file containing the measurements.

Listing 5.3: The type of the `eval` function

```
eval : Env -> NumericExpression -> Nat
```

The `eval` function uses this to evaluate a `NumericExpression`: Given an `Env` and a `NumericExpression`, the `eval` function returns a natural number. This is the result of that `NumericExpression` over that environment. To use the evaluation in a proof, we use the `Evald` type.

Listing 5.4: `Evald` as defined in the IDRIS model

```
data Evald : NumericExpression -> Nat -> Type where
    MkEvald : (x : NumericExpression) -> (y : Nat) -> Evald x y
```

Listing 5.4 shows that `Evald` is a data type constructed from a `NumericExpression` and `Nat`. This is because `eval` only evaluates the expression, it does not provide link the evaluated value and the original expression. The `Evald` type is a relation between the numeric expression, `x`, and its evaluation, `y`.

When applying a numeric operator to some `NumericExpressions`, we get a `BooleanExpression`.

5.1.3 Boolean Operators and Boolean Expressions

Boolean expressions result from either a numeric operator or a boolean operator. The grammar for constructing boolean expressions is:

$\langle \textit{BooleanExpression} \rangle$ ::= ; a paranthesised expression
 'BParen' $\langle \textit{BooleanExpression} \rangle$
 | ; the negation of an expression
 'Not' $\langle \textit{BooleanExpression} \rangle$
 | ; the conjunction of two expressions
 'And' $\langle \textit{BooleanExpression} \rangle$ $\langle \textit{BooleanExpression} \rangle$
 | ; the disjunction of two expressions
 'Or' $\langle \textit{BooleanExpression} \rangle$ $\langle \textit{BooleanExpression} \rangle$
 | ; testing two numbers for equality
 'Eq' $\langle \textit{NumericExpression} \rangle$ $\langle \textit{NumericExpression} \rangle$
 | ; testing two numbers for inequality
 'NEq' $\langle \textit{NumericExpression} \rangle$ $\langle \textit{NumericExpression} \rangle$
 | ; whether one number is strictly less than the other
 'LT' $\langle \textit{NumericExpression} \rangle$ $\langle \textit{NumericExpression} \rangle$
 | ; whether one number is less than or equal to the other
 'LTE' $\langle \textit{NumericExpression} \rangle$ $\langle \textit{NumericExpression} \rangle$
 | ; whether one number is strictly greater than the other
 'GT' $\langle \textit{NumericExpression} \rangle$ $\langle \textit{NumericExpression} \rangle$
 | ; whether one number is greater than or equal to the other
 'GTE' $\langle \textit{NumericExpression} \rangle$ $\langle \textit{NumericExpression} \rangle$

Similar to `NumericExpressions`, `BooleanExpressions` are evaluated over an environment.

Listing 5.5: The type of the `beval` function

```
beval : Env -> BooleanExpression -> Bool
```

Given an `Env` and a `BooleanExpression`, the `beval` function returns a boolean. This boolean is the result of evaluating that `BooleanExpression` over that environment. Like with `NumericExpressions`, to use the evaluation in a proof, we have the `BEvald` type.

Listing 5.6: BEvald as defined in the IDRIS model

```
data BEvald : BooleanExpression -> Nat -> Type where
    MkBEvald : (x : BooleanExpression) -> (y : Bool) -> BEvald x y
```

`BEvald` works like `Evald` except for boolean expressions and values. It is a data type constructed from a `BooleanExpression` and a `Bool`. Like `eval`, the `beval` function only evaluates the value of the boolean expression, it does not link the expression to the corresponding boolean. The `BEval` type is a relation between the boolean expression, `x`, and the boolean it evaluates to, `y`.

5.1.4 The LTE operator

The less-than-or-equals operator, `LTE`, is the only operator defined in the IDRIS prelude, as the other similar operators (i.e. `LT`, `GTE`, and `GT`) can be defined from it.

Listing 5.7: LT can be defined based on LTE

```
isLT : (m : Nat) -> (n : Nat) -> Dec (LTE (S m) n)
isLT m n = isLTE (S m) n
```

Strictly less-than, `LT`, can be defined from less-than-or-equals as follows: if the successor of a number m is LTE to another number n, then m is strictly less-than n; $(m+1) \leq n \Rightarrow m < n$.

Listing 5.8: GTE can be defined based on LTE

```
isGTE : (m : Nat) -> (n : Nat) -> Dec (LTE n m)
isGTE m n = isLTE n m
```

Greater-than-or-equals, `GTE`, can be defined from less-than-or-equals by simply swapping the operands: if a number n is LTE to a number m then m is greater-than-or-equal to n; $n \leq m \Rightarrow m \geq n$.

Listing 5.9: GT can be defined based on LTE

```
isLT : (m : Nat) -> (n : Nat) -> Dec (LTE (S n) m)
isLT m n = isLTE (S n) m
```

Finally, strictly greater-than, `GT`, can be defined as a 'combination' of the definitions of `LT` and `GTE`: if the successor of a number n is LTE to a number m then m is strictly greater-than to n; $(n+1) \leq m \Rightarrow m > n$.

The operators `And`, `Eq`, `LT`, `LTE`, `GT`, and `GTE` were already implemented. In order to have a complete set of operators, I implemented `NEq`, `Or`, and `Not`.

5.1.5 Modelling C programs

In order to model and prove properties of C programs, the TeamPlay project defines the `CLang` data-type.

Listing 5.10: The `CLang` data type

```
data CLang : Type where
    BlockTime   : (name : String) -> Nat -> CLang -> CLang
    StmtTime    : (name : String) -> Nat -> CLang -> CLang
    BlockEnergy : (name : String) -> Nat -> CLang -> CLang
    StmtEnergy  : (name : String) -> Nat -> CLang -> CLang
    Assert      : (Env -> Assertion) -> CLang -> CLang
    Halt        : CLang
```

A `CLang` consists of a series block or statement time measurements, and/or a series of block or statement energy measurements, and/or a series of assertions. Since time and energy measurements are numeric values, they are represented by natural numbers, each associated to a variable with a `name`. An `Assert` block is a function from an environment to an `Assertion`, i.e. something which will evaluate the `Assertion` given an environment. The `Halt` constructor indicates the end of the series.

5.2 Inequality (NEq)

Eq had already been implemented by the TeamPlay project, using the built-in (=) data type and the DecEq interface, as part of the D1.1 deliverable [11].

5.2.1 The (=) data type

Listing 5.11: The concept of an equality data type

```
data (=) : a -> b -> Type where
    Refl : x = x
```

A very useful pre-defined data type is the (=) data type. It is not quite defined as a data type since '=' is a reserved symbol, instead it is part of the Idris syntax. It is still a data type, just defined at a lower level of the language than the data construct. Its constructor Refl is short for 'reflexive'. Reflexivity is a mathematical property which, roughly, states that if two elements are reflexive, they are the same element. In Idris, Refl is the constructor for propositional equality; the = data type.

Listing 5.12: Examples of reflexivity

```
Idris> the ("World" = "World") Refl
Refl : "World" = "World"


Idris> the (True = True) Refl
Refl : True = True


Idris> the (1 + 2 + 3 = 6) Refl
Refl : 6 = 6
```

At the Idris prompt, we can use the the function to create some example instances of Refl. The last example shows that Idris evaluates the sides of a type before trying to construct the Refl.

Listing 5.13: Things that are not reflexive

```
Idris> the ("Foo" = "Bar") Refl
(input):1:1-24:When checking argument value to function
Prelude.Basics.the:
    Type mismatch between
        x = x (Type of Refl)
    and
        "Foo" = "Bar" (Expected type)

    Specifically:
        Type mismatch between
            "Foo"
        and
            "Bar"


Idris> the (True = False) Refl
(input):1:1-23:When checking argument value to function
Prelude.Basics.the:
    Type mismatch between
        x = x (Type of Refl)
    and
        True = False (Expected type)

    Specifically:
        Type mismatch between
            True
        and
            False


Idris> the (2 = 3) Refl
(input):1:1-16:When checking argument value to function
Prelude.Basics.the:
    Type mismatch between
        3 = 3 (Type of Refl)
    and
        2 = 3 (Expected type)

    Specifically:
        Type mismatch between
            3
        and
            2
```

If we try to instantiate `Refl` using elements which are not reflexive, the type-checker complains and tells us that it does not make sense and so we cannot create a `Refl` from things which are not reflexive. Successful application of the `Refl` constructor proves that things are equal, since it can only be called if the things truly are equal. In order to prove the opposite, that things cannot be equal, we use a different data type.

5.2.2 The TyNEq data type

Because there is no built-in `NEq` data type, I had to define one. This was tricky since in order to prove that two numbers are not equal, we have to be able to describe exactly why they cannot possibly be equal. This is captured in the `TyNEq` data-type.

Listing 5.14: The data-type used for capturing inequality

```
data TyNEq : Nat -> Nat -> Type where
    MkNEqL   : TyNEq Z (S k)
    MkNEqR   : TyNEq (S k) Z
    MkNEqRec : TyNEq k j -> TyNEq (S k) (S j)
```

The constructors for the `TyNEq` data type describe the different ways numbers can be not equal:

- **The first number is zero and the second is not** – in this case the numbers are not equal, specifically the left number is zero. Since no number can have zero as its successor, the numbers are not equal.
- **The second number is zero and the first is not** – in this case, the numbers are not equal, specifically the right number is zero. Since no number can have zero as its successor, the numbers are not equal.
- **The numbers are not equal** – in this case, the successors of the numbers must also be not equal. For example, $2 \neq 5 \rightarrow 3 \neq 6 \rightarrow 4 \neq 7$ etc.

Using the `TyNEq` data type, we can then define `NEq`.

Listing 5.15: The definition of `NEq`

```
data BooleanExprssion : Type where
    ⋮

    NEq :  (x : NumericExpression)
        -> (y : NumericExpression)
        -> Evald x x'
        -> Evald y y'
        -> Dec (TyNEq x' y')
        -> BooleanExpression
    ⋮
```

NEq takes two numeric expressions, proofs of what they evaluate to, and a decidable proof of whether the numeric expressions are inequal. From this, NEq returns a BooleanExpression which will evaluate to True if the numeric expressions were not equal, and False otherwise. For the decidability part, Dec (TyNEq x' y'), we need some proof functions which either prove that two numbers must be unequal or prove that they cannot possibly be.

5.2.3 The Void data type and the void function

The IDRIS prelude has a 'Void' data type. It expresses the impossibility of something happening.

Listing 5.16: The Void type has no constructors

```
data Void : type where
```

Since Void has no constructors, it is impossible to directly create an instance of Void. Therefore, if a function returns an instance of Void, this means that its arguments resulted in something which is impossible to create. From a logical point of view, the arguments to the function express a contradiction, and as such Void represents something which can only be constructed if we accept that the contradiction is true.

Listing 5.17: Zero cannot be the successor of a natural number

```
zeroNotSuc : (0 = S k) -> Void
zeroNotSuc Refl impossible
```

The first argument to the zeroNotSuc function is a reflexive equality expressing that zero is the successor of some natural number k. This is impossible and as such we cannot construct the Refl, which means that the function could return a Void. The 'impossible' keyword tells the IDRIS type checker that the pattern must not type check. Since we could not possibly have a Refl : 0 = S k, this holds.

Listing 5.18: Invalid use of the `impossible` keyword

```
boolRefl : (b : Bool) -> (b = b)
boolRefl True = Refl
boolRefl False impossible
```

Compiler error:

```
  |
3 | boolRefl False impossible
  |                ~~~~~~~~~~
boolRefl False is a valid case
```

The `boolRefl` function here simply shows that booleans are reflexive. If we try to use the `impossible` keyword for the `False` case, the type checker complains and tells us that it *is* a valid case.

Listing 5.19: The successor of a natural number cannot be zero

```
sucNotZero : (S k = 0) -> Void
sucNotZero Refl impossible
```

Similar to stating that zero cannot be the successor of a number, we can state that no natural number `k` can have zero as its successor. If we could create a `Refl : S k = 0`, we could create an instance of `Void`. Actually, if we could create an instance of `Void`, we would be able to create an instance of any type. Or logically: if we accept a contradiction to be true, we can prove anything.

Listing 5.20: The `void` function

```
void : Void -> a
```

The pre-defined `void` function captures this: given an instance of `Void`, the `void` function returns an instance of any type. This may seem like a peculiar thing to have: a function whose input cannot exist. However, if we can say that certain things *cannot* happen, we can use that to say more precisely what *can* happen.

Listing 5.21: Applying a function to an impossibility does not make it possible

```
noRec : (contra : (k = j) -> Void) -> (S k = S j) -> Void
noRec contra Refl = contra Refl
```

The `noRec` function takes a proof that two natural numbers are not equal and proves that their successors must then also be not equal. Given a `contra` which is a function from `(k = j)` to `Void` and a `Refl` representing `(S k = S j)`, we can use the `contra` function to return an instance of `Void`. Hence, the `Refl : S k = S j` cannot exist.

5.2.4 Decidability

Decidability is more specific than `Maybe` (Section 4.8). Instead of saying that we have `Just` the value or `Nothing`, decidability allows us to express that we can always *decide* whether a property holds or not for certain values.

Listing 5.22: `Dec` as defined in the Idris prelude

```
data Dec : (prop : Type) -> Type where
    Yes : (prf : prop) -> Dec prop
    No  : (contra : prop -> Void) -> Dec prop
```

The definition of `Dec` may seem similar to that of `Maybe`, specifically

```
Yes (prf : prop) -> Dec prop
```

seems very similar to

```
Just : (x : a) -> Maybe a
```

and that is because they are. Both `a` and `prop` represent the type of the element that may or may not be there. However, contrary to `Maybe`'s '`Nothing`' which simply is the absence of a value, `Dec`'s '`No`' holds a value: '`contra`'. The type of `contra` is `prop -> Void`, i.e. it is a proof that no value of the required type can exist (because if it did, we could return an instance of `Void`). This is a much stronger statement than `Nothing`. Instead of saying "the value is not there" we have said "here is why the value can never be there".

Using `Dec` and Listings 5.17, 5.19, and 5.21, we can decidably prove equality of natural numbers.

Listing 5.23: Proving equality of natural numbers

```
decEq : (a : Nat) -> (b : Nat) -> Dec (a = b)
decEq Z Z = Yes Refl
decEq Z (S j) = No zeroNotSuc
decEq (S k) Z = No sucNotZero
decEq (S k) (S j) = case decEq k j of
                        Yes prf    => Yes (cong prf)
                        No contra => No (noRec contra)
```

The `decEq` function models decidable equality of natural numbers. It takes two natural numbers `a` and `b` and decidably produces whether `a` is reflexive to `b`, `Dec (a = b)`. We do this by pattern matching on the input:

- zero and zero – are equal, and it is trivial to construct a `Refl` showing this

- zero and the successor of a number – are not equal as zero cannot be the successor of a natural number, so we return `No`, followed by Listing 5.17 which proves this
- the successor of a number and zero – are not equal as no number can have zero as its successor, so we return `No` followed by Listing 5.19 which proves this
- the successor of a number and the successor of another number – are equal if and only if the predecessors are equal, so we recurse on the predecessors and test if they are equal. If they are, we use the `cong` function on the proof. The `cong` function simply guarantees that equality respects function application. If the predecessors are not equal, we use the `noRec` function to prove that their successors can also not be equal.

Specifying impossible inputs allows us to refine which inputs *are* valid. Since this is something we commonly want to do, IDRIS provides several constructs for facilitating this.

5.2.5 Expressing the impossible

In Listing 5.17 and 5.19 we used the `impossible` keyword which tells the type checker that a pattern must not type check. We also used the `Void` data type to express that something cannot exist, and the `void` function to express that if `Void` coulb be instantiated, we could do anything.

Listing 5.24: The `Uninhabited` interface

```
interface Uninhabited t where
    uninhabited : t -> Void
```

An interface can be thought of as a classification of types. They provide methods which must be given in an implementation of the interface using a certain type. The `Uninhabited` interface is a generalisation of types which cannot be constructed.

Listing 5.25: 2 cannot equal 3

```
implementation Uninhabited (2 = 3) where
    uninhabited Refl impossible
```

If a type cannot exist (like `Refl : 2 = 3`), we can provide an implementation of the `Uninhabited` interface for it.

Listing 5.26: The `uninhabited` function

```
uninhabited : Uninhabited t => t -> Void
```

The `uninhabited` function, which is required by an implementation of `Uninhabited`, takes a thing which implements the `Uninhabited` interface and produces an instance of `Void`, i.e. if `uninhabited` can ever be given its argument `t`, we must have constructed a contradiction.

Listing 5.27: The `absurd` function

```
absurd : Uninhabited t => (h : t) -> a
absurd h = void (uninhabited h)
```

Using `uninhabited` and `void`, the Idris prelude defines the `absurd` function. Recalling that `uninhabited` constructs an instance of `Void` and that `void` constructs anything from this, `absurd` essentially states that the existence of any instance of something which implements `Uninhabited` is absurd as it would mean anything, any type, could be created.

Listing 5.28: Example pre-defined implementations of `Uninhabited`

```
implementation Uninhabited (True = False) where
    uninhabited Refl impossible

implementation Uninhabited (False = True) where
    uninhabited Refl impossible
```

Idris comes with many implementations of `Uninhabited` already defined. Two examples of this are the proofs that `True` cannot be `False`, and `False` cannot be `True`

Listing 5.29: Using `absurd`

```
isTrue : (b : Bool) -> Dec (b = True)
isTrue True  = Yes Refl
isTrue False = No absurd

isFalse : (b : Bool) -> Dec (b = False)
isFalse True  = No absurd
isFalse False = Yes Refl
```

One usage of `Uninhabited` could be to decidably determine whether a boolean is `True` or `False`. If the boolean *is* the correct instance, then we simply return a `Yes Refl`. And if the boolean is *not* the correct instance, we return `No absurd` which shows that it cannot be the same boolean because if it was, we could construct any type. `No absurd` shows that we have had a contradiction.

5.2.6 Impossible inequalities

With ways of expressing impossibility, we can define which inequalities are impossible to be true and why.

Listing 5.30: Not equals cannot be constructed on 0 0

```
implementation Uninhabited (TyNEq Z Z) where
    uninhabited MkNEqL   impossible
    uninhabited MkNEqR   impossible
    uninhabited MkNEqRec impossible
```

If we try to construct a `TyNEq` from 0 0, i.e. try to prove $0 \neq 0$, this is impossible: `MkNEqL` requires only the left argument to be zero, `MkNEqR` requires only the right argument to be zero, and `MkNEqRec` requires the successors to be not equal (i.e. $1 \neq 1$, $2 \neq 2$, etc.). So `TyNEq 0 0` is uninhabited as we cannot use any of its constructors to create that type. For non-zero equal numbers, we need to take a similar approach to `MkNEqRec` (Listing 5.14).

Listing 5.31: Proving inequality of numbers is impossible

```
succNEqImpossible : (contra : TyNEq k j -> Void) ->
                    TyNEq (S k) (S j) -> Void
succNEqImpossible contra (MkNEqRec x) = contra x
```

The `succNEqImpossible` function takes a proof that `TyNEq k j` cannot be constructed and proves that in that case, `TyNEq (S k) (S j)` can also not be constructed. What this does is maintain that if two numbers are *not* not equal, i.e. that they *are* equal, then so are their successors.

With constructors for the valid `TyNEq` cases, and proofs why the invalid cases are impossible, we can construct the decidability rules for `TyNEq`. These will allow us to construct the `Dec (TyNEq x' y')` part from Listing 5.15.

Listing 5.32: Decidability rules for `NEq`

```
isNEq : (n1 : Nat) -> (n2 : Nat) -> Dec (TyNEq n1 n2)
isNEq Z Z         = No absurd
isNEq Z (S k)     = Yes MkNEqL
isNEq (S k) Z     = Yes MkNEqR
isNEq (S k) (S j) = case isNEq k j of
                         Yes prf   => Yes (MkNEqRec prf)
                         No contra => No (succNEqImpossible contra)
```

If the `isNEq` is given two zeros as its arguments, then it is absurd to construct an inequality (as described in Listing 5.30). If either the first or second argument is zero and the other is not, then the inequality is trivial as no natural number can have `Z` as its successor. In the final case, where both numbers are non-zero, we recurse on their predecessors. If the predecessors are not equal (by one of them being zero and the other not), then the numbers themselves must also be not equal. On the other hand, if it was impossible to prove the predecessors unequal, we use `succNEqImpossible` to maintain/prove that it is also impossible for the successors to be unequal.

With the `eval` function and the `Evald` type giving us the numbers, and the decidability rules through the implementation of the `Uninhabited` interface and using the `succNEqImpossible` function, we have all the parts needed for the `NEq` constructor. Hence, we can prove inequality between numbers.

Listing 5.33: Evaluating `NEq` over an environment

```
beval : (env : Env) -> (b : BooleanExpression) -> Bool
⋮
beval env (NEq x y x' y' (Yes prf))   = True
beval env (NEq x y x' y' (No contra)) = False
⋮
```

Given an environment and an inequality which has been proven to hold, the inequality evaluates to `True`. On the other hand, if the inequality has been proven impossible to hold, then it evaluates to `False`.

5.3 Boolean Disjunction (`Or`)

The boolean operators are made up of different stages: Constructors for boolean expressions which evaluate to true, the operators keywords, an evaluation of the boolean expression, and a proof of what the boolean expression evaluates to. Boolean conjunction (`And`) was already implemented. I used it as a starting point for how to implement `Or`, as they are somewhat similar.

Listing 5.34: The constructors for true `And` and `Or` statements

```
mutual
    data TyAnd : Bool -> Bool -> Type where
        MkAnd  : TyAnd True True

    data TyOr : Bool -> Bool -> Type where
        MkOr  : TyOr True True
        MkOrL : TyOr True False
        MkOrR : TyOr False True
```

There is only one case where `And` evaluates to true, i.e. `And True True`. This is reflected in that the only constructor for `TyAnd` has to have both arguments be `True`. With this idea in mind, the constructors for `Or` can be implemented. `TyOr` can be constructed when the expression evaluates to true, i.e. in three cases:

- both arguments are `True`, `Or True True`
- the left argument is `True`, `Or True False`
- the right argument is `True`, `Or False True`

With constructors for `TyOr`, we can then define the `Or` operator similar to the `And` operator.

Listing 5.35: The definitions of `And` and `Or`

```
data BooleanExpression : Type where
    ⋮

    And :  (x : BooleanExpression)
        -> (y : BooleanExpression)
        -> BEvald x x'
        -> BEvald y y'
        -> Dec (TyAnd x' y')
        -> BooleanExpression

    Or  :  (x : BooleanExpression)
        -> (y : BooleanExpression)
        -> BEvald x x'
        -> BEvald y y'
        -> Dec (TyOr x' y')
        -> BooleanExpression
    ⋮
```

`And` takes 5 arguments: two boolean expressions, the proofs of what these evaluated to, and decidable proof of whether those arguments would cause the `And` to evaluate to true. From the constructors of `TyAnd` (Listing 5.34), this can only be constructed when both arguments are `True`, i.e. when the `And` would evaluate to `True`. Based on this, `And` returns a new `BooleanExpression` (which can then be evaluated using the `beval` function). `Or` takes the same first 4 arguments. The difference being that its 5^{th} argument is a decidable proof of whether those arguments would cause the `Or` to evaluate to `True`. From the constructors of `TyOr` (Listing 5.34), this can only be constructed if one or both of the arguments is `True`, i.e. the `Or` would evaluate to `True`. Based on its arguments, `Or` then returns a new `BooleanExpression` (which can then be evaluated using the `beval` function).

Both `And` and `Or` rely on `Dec`, i.e. that a proof that it cannot evaluate to `True` exists.

Listing 5.36: Impossible `And` cases

```
implementation Uninhabited (TyAnd False True) where
    uninhabited MkAnd impossible

implementation Uninhabited (TyAnd True False) where
    uninhabited MkAnd impossible

implementation Uninhabited (TyAnd False False) where
    uninhabited MkAnd impossible
```

Since `MkAnd` requires both arguments to be `True`, an implementation of `Uninhabited` can be given for the other three cases. In each of the cases, the `And` would evaluate to `False`. With an implementation of `Uninhabited`, we can provide a proof for each case that it is impossible to construct a `MkAnd` and as such, the expression must evaluate to `False`. For `Or` there is only one case which evaluates to `False`.

Listing 5.37: The impossible `Or` case

```
implementation Uninhabited (TyOr False False) where
    uninhabited MkOr  impossible
    uninhabited MkOrL impossible
    uninhabited MkOrR impossible
```

`MkOr` requires at least one of the arguments to be `True` (see Listing 5.34). As such, the case where we would have a `TyOr False False` is impossible. None of the constructors provided for `TyOr` can be used on `False False`, so a `TyOr` cannot be created. This means the `Or` cannot evaluate to `True`, i.e. we have a proof that it must evaluate to `False`. Using these proofs of when a `TyOr` can be constructed, we can define the decidability rules for `Or`.

Listing 5.38: Decidability rules for `Or`

```
isOr : (b1 : Bool) -> (b2 : Bool) -> Dec (TyOr b1 b2)
isOr False False = No absurd
isOr True False  = Yes MkOrL
isOr False True  = Yes MkOrR
isOr True True   = Yes MkOr
```

When at least one of the arguments to `isOr` is `True`, it is possible to construct a `TyOr` using the appropriate constructor. If both arguments are `False`, constructing a `TyOr` is absurd since none of the constructors can be applied to `False False` (as defined in Listing 5.37). Hence, given two boolean expressions, a proof of what they evaluate to, and using the decidability rules, we can construct a `BooleanExpression` using the `Or` constructor, i.e. we can prove boolean disjunction.

Listing 5.39: Evaluating `Or` over an environment

```
beval : (env : Env) -> (b : BooleanExpression) -> Bool
⋮
beval env (Or x y x' y' (Yes prf))    = True
beval env (Or x y x' y' (No contra))) = False
⋮
```

The evaluation of a disjunction over an environment is `True` given a disjunction which has been proven to hold, and `False` given a disjunction which has been proven to be impossible to hold.

5.4 Boolean Negation (`Not`)

As with the other operators, we need to have a data type to capture correct things. In this case, which negations are valid.

Listing 5.40: The case where `Not` would evaluate to `True`

```
data TyBNot : Bool -> Type where
    MkBNot  : TyBNot False
```

The only case where boolean negation evaluates to `True` is when negating `False`. Therefore, it is only possible to construct a `TyBNot` if the given `BooleanExpression` was `False`. Similarly, there is only one case where boolean negation evaluates to `False`.

Listing 5.41: The `uninhabited` case for constructing a `TyBNot`

```
implementation Uninhabited (TyBNot True) where
    uninhabited MkBNot impossible
```

The `MkBNot` constructor only works if the argument is `False`, so constructing a `TyBNot` given `True` is impossible.

Listing 5.42: Decidability rules for `Not`

```
isNot : (b : Bool) -> Dec (TyBNot b)
isNot False = Yes MkBNot
isNot True  = No absurd
```

If the argument is False, the negation is True, so a TyBNot is constructed. Otherwise, if the argument is True, then it is absurd to construct a TyBNot as the negation will evaluate to False. This, combined with the given boolean expression and a proof of what it evaluates to, allows us to prove boolean negation.

Listing 5.43: beval for Not

```
beval : (env : Env) -> (b : BooleanExpression) -> Bool
    ⋮
beval env (Not x x' (Yes prf))    = True
beval env (Not x x' (No contra)) = False
    ⋮
```

Since we can prove negation, we can also evaluate it. Given an environment Env and a Not-expression which has been proven to hold true, its evaluation is the boolean 'True'. Otherwise, if it has been proven to be impossible to hold, then its evaluation is the boolean 'False'.

Chapter 6

Evaluation

In order to be able to use the existing operators, and the ones I had implemented, in code, I needed to make sure that they worked, i.e. that they did not return anything unexpected (e.g., `Eq` should not evaluate to true for 1 and 3, and `And False False` should not evaluate to `True`). This is the described in the first section (6.1) of this chapter. With working operators, it should then be possible to construct slightly more complex C programs, i.e. programs which do more than evaluate the operators, and annotate and model them using the framework. This is described in the second section of this chapter (6.2). In the third section of this chapter (6.3), some example of real-world use-cases and models are provided, demonstrating that the framework can be used with these programs.

The parser taking the CSV file and the annotated C code is currently work-in-progress by the TeamPlay project in St Andrews. The compiler and measurement tools are a part of the TeamPlay project being worked on in Germany by TUHH: the Hamburg University of Technology. As such, the models had to be constructed manually and with made-up values. This was somewhat challenging as I had to keep track of which parts of the model depended on what and reason about what the annotations would translate to. All of this should be automated when the parser is complete. However, since the functionality of the framework is unaffected by the lack of these tools, as its main concern for constructing certificates is that there are values in expressions and that the expressions are correct, the evaluation of the framework remains valid.

6.1 Operators

In order to evaluate the operators, I constructed multiple examples which tested specific input cases/categories and asserted that they functioned as expected in each case. For numeric operators, I tested the following input cases:

- zero and zero – they form the base case of natural numbers so for all recursive operations to be correct, the base case has to be correct
- zero and one or one and zero – since one is the immediate successor of zero, there is only the need for one recursive step, so if this case works, adding more recursive steps is likely to work
- numbers greater than zero and one – these require multiple recursive steps and so if they work, all numbers should work as we then know that recursion works

For boolean operators, I tested the possible combinations of `True` and `False` for the operator and compared the results with the corresponding truth-table for the corresponding logical operator.

Because I am purely evaluating the operators, there are no variables used, only literals. This results in all the environments in the Idris outputs being empty.

6.1.1 The `Eq` Operator

Listing 6.1: A C program which requires $0 = 0$

```
void main(void)
{
    __teamplay_assert(0 == 0);
}
```

The C program itself may not seem very interesting. However, it is a structurally correct C program. Its purpose is to test the base case of equality between natural numbers: that zero equals zero. This equality assertion would be similar for larger, more complex programs using the annotation.

```
import Darknet

mutual
  eq_0_0 : CLang
  eq_0_0 = Assert eq_0_0_assert
         $ Halt

  eq_0_0_assert : Env -> Assertion
  eq_0_0_assert env =
    let
      x = Lit 0
      y = Lit 0
      x' = eval env x
      y' = eval env y
      prf = decEq x' y'
    in
      MkAssertion (Eq x y (MkEvald x x') (MkEvald y y') prf)
```

Transforming the C annotation into IDRIS, we get the model detailed above. The model '`eq_0_0` is a `CLang` consisting of an assertion named '`eq_0_0_assert`' and nothing more (`Halt`). This assertion contains two literals `x` and `y` both carrying the value 0, their evaluation `x'` and `y'` over a given environment `env`, and a decidable proof `prf` that the numbers are equal. Using these, we can use `MkAssertion` to construct an instance of `Assertion` (Listing 5.2).

```
Idris> mkCertificate eq_0_0


([],
 [MkAssertion (Eq (Lit 0)
                  (Lit 0)
                  (MkEvald (Lit 0) 0)
                  (MkEvald (Lit 0) 0)
                  (Yes Refl))]) : (List (String, Nat), List Assertion)
```

Using the `mkCertificate` function, we construct a certificate based on the `Assertion` made. We can see that the evaluation of the literals has worked correctly from `MkEvald (Lit 0) 0`. The final argument to the `MkAssertion` is a `Yes Refl`, so the `Eq` operator worked for zero and zero: It found that they were equal, which is correct.

Listing 6.2: A C program which requires 1 = 1

```c
void main(void)
{
    __teamplay_assert(1 == 1);
}
```

Again, the actual C program is structurally sound. The assertion tests that 1 equals 1, as one is the immediate successor of zero and hence only requires one recursive step.

```idris
import Darknet

mutual
  eq_1_1 : CLang
  eq_1_1 = Assert eq_1_1_assert
           $ Halt

  eq_1_1_assert : Env -> Assertion
  eq_1_1_assert env =
    let
      x = Lit 1
      y = Lit 1
      x' = eval env x
      y' = eval env y
      prf = decEq x' y'
    in
      MkAssertion (Eq x y (MkEvald x x') (MkEvald y y') prf)
```

The resulting Idris model 'eq_1_1' contains an assertion that 1 equals 1 (Assert eq_1_1_assert) and nothing more (Halt). This assertion in turn contains the two literals (x and y) both carrying a value of 1 (Lit 1), their evaluation over the given environment (x' and y'), and a decidable proof 'prf' that the evaluations are equal.

```
Idris> mkCertificate eq_1_1


([],
 [MkAssertion (Eq (Lit 1)
                  (Lit 1)
                  (MkEvald (Lit 1) 1)
                  (MkEvald (Lit 1) 1)
                  (Yes Refl))]) : (List (String, Nat), List Assertion)
```

When passing `eq_1_1` through the `mkCertificate` function, we see that the evaluations were correct (from `MkEvald (Lit 1) 1`). The final argument to `MkAssertion` contains a `Yes Refl`, so the model has confirmed that $1 = 1$ and as such, that `Eq` works for 1.

Listing 6.3: A C program which requires $0 = 1$

```
void main(void)
{
    __teamplay_assert(0 == 1);
}
```

With $0 = 0$ and $1 = 1$ working, I evaluated that the operator was working correctly for 0 and 1 by testing that $0 = 1$ and $1 = 0$ did not produce a `Yes Refl`.

```
import Darknet

mutual
  eq_0_1 : CLang
  eq_0_1 = Assert eq_0_1_assert
         $ Halt

  eq_0_1_assert : Env -> Assertion
  eq_0_1_assert env =
    let
      x = Lit 0
      y = Lit 1
      x' = eval env x
      y' = eval env y
      prf = decEq x' y'
    in
      MkAssertion (Eq x y (MkEvald x x') (MkEvald y y') prf)
```

This time, the Idris model contains two numerically different literals, their evaluation, and a decidable proof of whether they are equal or not.

```
Idris> mkCertificate eq_0_1
```

```
([],
 [MkAssertion (Eq (Lit 0)
                  (Lit 1)
                  (MkEvald (Lit 0) 0)
                  (MkEvald (Lit 1) 1)
                  (No ZnotS))]) : (List (String, Nat), List Assertion)
```

The output from the `mkCertificate` function over `eq_0_1` shows the literals were evaluated correctly. The proof argument this time contains `No ZnotS`. This is the built-in version of the proof that zero cannot be the successor of a natural number (Listing 5.17). Given that 0 and 1 were the arguments to `Eq`, the operator has worked correctly: zero and one are not equal, because zero cannot be the successor of any natural number.

The details of the vice-versa of this, i.e. $1 = 0$ not returning `Yes Refl`, can be found in Appendix A, Section A.1. The output from `mkCertificate` function is slightly unusual, and as such is included here.

```
Idris> mkCertificate eq_1_0
```

```
([],
 [MkAssertion (Eq (Lit 1)
                  (Lit 0)
                  (MkEvald (Lit 1) 1)
                  (MkEvald (Lit 0) 0)
                  (No (negEqSym ZnotS)))]) :
(List (String, Nat), List Assertion)
```

However, the certificate produced is slightly unexpected: it still contains `ZnotS`, this time combined with a function '`negEqSym`'. The `negEqSym` function has the following type:

`negEqSym :  ((a = b) -> Void) -> (b = a) -> Void`

Essentially, the `negEqSym` proves that if two numbers are not equal, then swapping them around does not change that fact; It proves/states that the negation of equality is *symmetric*. Hence, `ZnotS` can also be used to prove that $1 = 0$ is false as it is symmetric to proving $0 = 1$ is false. So the `Eq` operator works correctly when asserting $1 = 0$.

Listing 6.4: A C program which requires 3 = 3

```c
void main(void)
{
    __teamplay_assert(3 == 3);
}
```

With both zero and one working, the remaining case is to test that `Eq` works with multiple recursive calls, i.e. any natural number greater than 1.

```idris
import Darknet

mutual
  eq_3_3 : CLang
  eq_3_3 = Assert eq_3_3_assert
         $ Halt

  eq_3_3_assert : Env -> Assertion
  eq_3_3_assert env =
    let
      x = Lit 3
      y = Lit 3
      x' = eval env x
      y' = eval env y
      prf = decEq x' y'
    in
      MkAssertion (Eq x y (MkEvald x x') (MkEvald y y') prf)
```

The `eq_3_3` model contains two '3'-literals, their evaluation over the given environment, and a decidable proof that the two evaluations are equal.

```
Idris> mkCertificate eq_3_3


([],
 [MkAssertion (Eq (Lit 3)
                  (Lit 3)
                  (MkEvald (Lit 3) 3)
                  (MkEvald (Lit 3) 3)
                  (Yes Refl))]) : (List (String, Nat), List Assertion)
```

Both evaluations returned 3 as expected and the final argument to `MkAssertion` is a `Yes Refl`. Hence, the `Eq` operator works for numbers greater than 0 and 1; It works in general.

Listing 6.5: A C program which requires $1 = 3$

```c
void main(void)
{
    __teamplay_assert(1 == 3);
}
```

Finally, to make sure that the operator disproves incorrect equalities for numbers greater than 1, I used the C program above.

```idris
import Darknet

mutual
  eq_1_3 : CLang
  eq_1_3 = Assert eq_1_3_assert
         $ Halt

  eq_1_3_assert : Env -> Assertion
  eq_1_3_assert env =
    let
      x = Lit 1
      y = Lit 3
      x' = eval env x
      y' = eval env y
      prf = decEq x' y'
    in
      MkAssertion (Eq x y (MkEvald x x') (MkEvald y y') prf)
```

The model contains the literals '1' and '3', their evaluation over the given environment, and a decidable proof of their equality.

```
Idris> mkCertificate eq_1_3


([],
 [MkAssertion (Eq (Lit 1)
                  (Lit 3)
                  (MkEvald (Lit 1) 1)
                  (MkEvald (Lit 3) 3)
                  (No (\h => ZnotS (succInjective 0 2 h))))]) :
(List (String, Nat), List Assertion)
```

Both evaluations match with the literals. The final argument in the `MkAssertion` argument uses a lambda function and the `succInjective` function. It has the type:

`succInjective : (left : Nat) -> (right : Nat) -> (p : S left = S right) -> left = right`

It takes two numbers and proves that if the successor of the left number is reflexive to the successor of the right number, then the numbers themselves must be equal. The argument `h` is a proof that `0 = S k`, so if it exists, then we could create a `Void`. This is a contradiction, so the equality is impossible. Swapping the operands results in almost the same proof.

```
([],
 [MkAssertion (Eq (Lit 3)
                  (Lit 1)
                  (MkEvald (Lit 3) 3)
                  (MkEvald (Lit 1) 1)
                  (No (\h => ZnotS (sym (succInjective 2 0 h)))))]) :
(List (String, Nat), List Assertion)
```

The only difference between this output is that it uses the `sym` function. Like with $0 = 1$ and $1 = 0$, this proves that numbers not being equal is symmetrical, so the same proof that they are not equal can be used. These two examples ($1 = 3$ and $3 = 1$), combined with the example in Listing 6.4, shows that the `Eq` function (and by extension, the `==` operator) works as intended for numbers greater than 0 and 1: if they are equal, then `Eq` proves it, and if they are not equal, then it correctly produces a proof as to why this is.

6.1.2 The `NEq` Operator

Listing 6.6: A C program requiring $0 \neq 1$

```
void main(void)
{
    __teamplay_assert(0 != 1);
}
```

The C program for testing inequality is syntactically correct. Any more complex program which was trying to assert an inequality at some point would use the same annotation and operator.

```
import Darknet

mutual
  neq_0_1 : CLang
  neq_0_1 = Assert neq_0_1_assert
          $ Halt

  neq_0_1_assert : Env -> Assertion
  neq_0_1_assert env =
    let
      x = Lit 0
      y = Lit 1
      x' = eval env x
      y' = eval env y
      prf = isNEq x' y'
    in
      MkAssertion (NEq x y (MkEvald x x') (MkEvald y y') prf)
```

The resulting IDRIS model contains the literals 0 and 1, their evaluation, and a decidable proof of whether they are equal or not. These are passed as arguments to the `NEq` function, which is in turn used to construct an `Assertion` using `MkAssertion`.

```
Idris> mkCertificate neq_0_1
```

```
([],
 [MkAssertion (NEq (Lit 0)
                   (Lit 1)
                   (MkEvald (Lit 0) 0)
                   (MkEvald (Lit 1) 1)
                   (Yes MkNEqL))]) :
(List (String, Nat), List Assertion)
```

Constructing the certificate using the `mkCertificate` function shows that the last argument to `NEq` was a `Yes MkNEqL`, i.e. that the numbers are not equal, specifically because the left number was smaller than the right number, `MkNEqL`.

```
Idris> mkCertificate neq_1_0


([],
 [MkAssertion (NEq (Lit 1)
                   (Lit 0)
                   (MkEvald (Lit 1) 1)
                   (MkEvald (Lit 0) 0)
                   (Yes MkNEqR))]) :
(List (String, Nat), List Assertion)
```

Switching the operands around does not change the C program or the model much (see Appendix A.2, Listing A.3). However, the resulting certificate is slightly different: the numbers are still shown to be not equal (the certificate contains a `Yes`) but this time specifically because the right number was smaller than the left number, `MkNEqR`. For the arguments 0 and 1 the `NEq` function evaluates correctly that the numbers are not equal, regardless of which side of the operator they are on.

Listing 6.7: A C program requiring $0 \neq 0$

```c
void main(void)
{
    __teamplay_assert(0 != 0);
}
```

Since the program requires 0 to be not equal to 0, the `NEq` operator should evaluate to false.

```
import Darknet

mutual
  neq_0_0 : CLang
  neq_0_0 = Assert neq_0_0_assert
            $ Halt

  neq_0_0_assert : Env -> Assertion
  neq_0_0_assert env =
    let
      x = Lit 0
      y = Lit 0
      x' = eval env x
      y' = eval env y
      prf = isNEq x' y'
    in
      MkAssertion (NEq x y (MkEvald x x') (MkEvald y y') prf)
```

The IDRIS model contains two literals, both carrying the value 0, their evaluation, and a decidable proof of whether they are not equal.

```
Idris> mkCertificate neq_0_0


([],
 [MkAssertion (NEq (Lit 0)
                   (Lit 0)
                   (MkEvald (Lit 0) 0)
                   (MkEvald (Lit 0) 0)
                   (No absurd))]) :
(List (String, Nat), List Assertion)
```

Constructing a certificate shows that the numbers are not unequal (`No`). The proof that they cannot be unequal is the `absurd` function, which can be used thanks to the implementation of `Uninhabited` that we provided in Section 5.2, Listing 5.30.

```
Idris> mkCertificate neq_1_1


([],
 [MkAssertion (NEq (Lit 1)
                   (Lit 1)
                   (MkEvald (Lit 1) 1)
                   (MkEvald (Lit 1) 1)
                   (No (succNEqImpossible absurd)))]) :
(List (String, Nat), List Assertion)
```

Changing the operands to 1, i.e. having a program which requires the immediate successor of 0 to be not equal to itself (Appendix A.2, Listing A.4) results in a similar certificate. The proof that this cannot be is still '`absurd`', but passed through the `succNEqImpossible` function (detailed in Listing 5.31) which proves that if two numbers are proven to not be unequal, then their successors must also not be unequal.

With these examples, the `NEq` operator has been shown to work for for the base case, 0, and its immediate successor, 1. For numbers greater than these, more recursive steps are required.

Listing 6.8: A C program requiring $2 \neq 5$

```c
void main(void)
{
    __teamplay_assert(2 != 5);
}
```

Both 2 and 5 are greater than 0 and 1, and so are not part of the base cases. Since they are higher successors, they should require more recursive steps to show that they are not equal. If it works, then the recursive step works and so any natural number would work.

```idris
import Darknet

mutual
  neq_2_5 : CLang
  neq_2_5 = Assert neq_2_5_assert
          $ Halt

  neq_2_5_assert : Env -> Assertion
  neq_2_5_assert env =
    let
      x = Lit 2
      y = Lit 5
      x' = eval env x
      y' = eval env y
      prf = isNEq x' y'
    in
      MkAssertion (NEq x y (MkEvald x x') (MkEvald y y') prf)
```

Constructing the model, we have the literals 2 and 5, their evaluation over a given environment, and a decidable proof of their inequality. This is passed to the `NEq` constructor to construct a `BooleanExpression` which is then passed to the `MkAssertion` constructor.

```
Idris> mkCertificate neq_2_5


([],
 [MkAssertion (NEq (Lit 2)
                   (Lit 5)
                   (MkEvald (Lit 2) 2)
                   (MkEvald (Lit 5) 5)
                   (Yes (MkNEqRec (MkNEqRec MkNEqL))))]) :
(List (String, Nat), List Assertion)
```

Creating a certificate based on the model shows that the literals have been correctly evaluated and associated. It was also possible to construct a proof that the numbers are not equal (the `Yes` part of the certificate). As can be seen by the repeated `MkNEqRec` constructor in the proof, the recursive step works; The left number is smaller (`MkNEqL`), specifically it is 3 smaller (1 smaller + 2 recursive steps). Swapping the operands and generating the resulting model (Appendix A.2, Listing A.5) should result in a similar result.

```
([],
 [MkAssertion (NEq (Lit 5)
                   (Lit 2)
                   (MkEvald (Lit 5) 5)
                   (MkEvald (Lit 2) 2)
                   (Yes (MkNEqRec (MkNEqRec MkNEqR))))]) :
(List (String, Nat), List Assertion)
```

Having swapped the order of the operands, the resulting certificate is very similar to the previous one. The literals have swapped order, and the two recursive steps are still there. However, this time the numbers are not equal specifically because right number has been shown to be smaller (`MkNEqR`). So the `NEq` operator can show that natural numbers greater than 0 and 1 are not equal. However, we still need to show that it can disprove the inequality between numbers greater than 0 and 1.

Listing 6.9: A C program requiring $3 \neq 3$

```
void main(void)
{
    __teamplay_assert(3 != 3);
}
```

To completely test that the operator works correctly for numbers greater than 0 and 1, the above C program requires that 3 be not equal to itself. This should evaluate to false. If it does, then since 3 requires multiple recursive steps, the operator can prove that two numbers greater than 0 and 1 cannot be unequal.

```
import Darknet

mutual
  neq_3_3 : CLang
  neq_3_3 = Assert neq_3_3_assert
          $ Halt

  neq_3_3_assert : Env -> Assertion
  neq_3_3_assert env =
    let
      x = Lit 3
      y = Lit 3
      x' = eval env x
      y' = eval env y
      prf = isNEq x' y'
    in
      MkAssertion (NEq x y (MkEvald x x') (MkEvald y y') prf)
```

The IDRIS model has two literals carrying the value 3, their evaluation over a given environment, and a decidable proof of whether they are not equal.

```
Idris> mkCertificate neq_3_3
```

```
([],
 [MkAssertion (NEq (Lit 3)
                   (Lit 3)
                   (MkEvald (Lit 3) 3)
                   (MkEvald (Lit 3) 3)
                   (No (succNEqImpossible (succNEqImpossible
                        (succNEqImpossible absurd)))))]) :
(List (String, Nat), List Assertion)
```

The resulting certificate shows that the literals were evaluated correctly and that the numbers were shown to not be unequal (the `No` part in the certificate). Similar to the example for $1 \neq 1$, the proof of this is the `absurd` function, but this time recursively passed through the `succNEqImpossible` function three times (due to the number being 3). This shows that the `NEq` operator works correctly in terms of proving that two natural numbers greater than 0 and 1 cannot be unequal.

Having shown that the `NEq` operator works correctly for the base case (0), its immediate successor (1), and natural numbers greater than these, we can conclude that the `NEq` operator works as intended.

6.1.3 The LTE Operator

Listing 6.10: A C program which requires $0 \leq 0$

```c
void main(void)
{
    __teamplay_assert(0 <= 0);
}
```

This C program captures the essential structure of a program requiring that a property be less than or equal to another. It has a function, two properties compared by the less-than-or-equals (LTE) operator, and the `assert` annotation.

```idris
module Examples.LTE

import Darknet

mutual
  lte_0_0 : CLang
  lte_0_0 = Assert lte_0_0_assert
          $ Halt

  lte_0_0_assert : Env -> Assertion
  lte_0_0_assert env =
    let
      x = Lit 0
      y = Lit 0
      x' = eval env x
      y' = eval env y
      prf = isLTE x' y'
    in
      MkAssertion (LTE x y (MkEvald x x') (MkEvald y y') prf)
```

Translating this to an IDRIS model we get two literals both carrying the value 0, their evaluation over a given environment, and a decidable proof of whether the first is less than or equal to the second. These are passed to the `LTE` constructor to create a `BooleanExpression` which is used to create an `Assertion`.

```
([],
 [MkAssertion (LTE (Lit 0)
                    (Lit 0)
                    (MkEvald (Lit 0) 0)
                    (MkEvald (Lit 0) 0)
                    (Yes LTEZero))]) :
(List (String, Nat), List Assertion)
```

The resulting certificate shows that the literals were evaluated correctly, and the `Yes` on the last line shows that we successfully proved that 0 is less than or equal to 0. The `LTEZero` proof is part of the IDRIS prelude and states that zero is the smallest natural number, so any natural number is less than or equal to it. This means that $0 \leq 1$ should be identical (Appendix A.3, Listing A.6)

```
Idris> mkCertificate lte_0_1
```

```
([],
 [MkAssertion (LTE (Lit 0)
                    (Lit 1)
                    (MkEvald (Lit 0) 0)
                    (MkEvald (Lit 1) 1)
                    (Yes LTEZero))]) :
(List (String, Nat), List Assertion)
```

The certificate created by asserting $0 \leq 1$ is identical (apart from the '1'-literal) due to it still involving zero on the left-hand side.

Listing 6.11: A C program which requires $1 \leq 1$

```c
void main(void)
{
    __teamplay_assert(1 <= 1);
}
```

Again, the program has the same core structure as any which would assert that one property was LTE to the other. This time both properties are 1 to test that the operator works with the immediate successor of 0.

```
import Darknet

mutual
  lte_1_1 : CLang
  lte_1_1 = Assert lte_1_1_assert
          $ Halt

  lte_1_1_assert : Env -> Assertion
  lte_1_1_assert env =
    let
      x = Lit 1
      y = Lit 1
      x' = eval env x
      y' = eval env y
      prf = isLTE x' y'
    in
      MkAssertion (LTE x y (MkEvald x x') (MkEvald y y') prf)
```

The resulting IDRIS model has two literals both carrying their the value 1, their evaluation over the given environment, and a decidable proof of the first being LTE to the other.

```
Idris> mkCertificate lte_1_1


([],
 [MkAssertion (LTE (Lit 1)
                   (Lit 1)
                   (MkEvald (Lit 1) 1)
                   (MkEvald (Lit 1) 1)
                   (Yes (LTESucc LTEZero)))]) :
(List (String, Nat), List Assertion)
```

Passing the model through the `mkCertificate` function, we see that it has successfully evaluated the literals and proven that 1 is LTE than itself, by applying `LTEZero` using the `LTESucc` function. It has type:
`LTESucc :  LTE left right -> LTE (S left) (S right)`
The `LTESucc` function takes a proof that two numbers are LTE to each other, and proves than in that case, so are their successors. So `Yes (LTESucc LTEZero)` says that since 0 is LTE to any natural number n, $(0+1)$ is LTE to any natural number $(n+1)$.

Listing 6.12: A C program requiring $1 \leq 3$

```c
void main(void)
{
    __teamplay_assert(1 <= 3);
}
```

For the case where zero is not on the left-hand side, recursive steps should be required.

```idris
module Examples.LTE

import Darknet

mutual
  lte_1_3 : CLang
  lte_1_3 = Assert lte_1_3_assert
          $ Halt

  lte_1_3_assert : Env -> Assertion
  lte_1_3_assert env =
    let
      x = Lit 1
      y = Lit 3
      x' = eval env x
      y' = eval env y
      prf = isLTE x' y'
    in
      MkAssertion (LTE x y (MkEvald x x') (MkEvald y y') prf)
```

The resulting model this time contains two non-zero literals (1 and 3 respectively), their evaluation over a given environment, and a decidable proof that the left is LTE to the right.

```
Idris> mkCertificate lte_1_3


([],
 [MkAssertion (LTE (Lit 1)
                   (Lit 3)
                   (MkEvald (Lit 1) 1)
                   (MkEvald (Lit 3) 3)
                   (Yes (LTESucc LTEZero)))]) :
(List (String, Nat), List Assertion)
```

The certificate shows that the evaluation successfully proved the comparison to be true. The specifics of the proof show that the left-hand side reached 0 before (or at the same time as) the right-hand side did, so we can apply the `LTEZero` proof to that side using the `LTESucc` function (in this case once, to reach 1).

```
([],
 [MkAssertion (LTE (Lit 3)
                   (Lit 3)
                   (MkEvald (Lit 3) 3)
                   (MkEvald (Lit 3) 3)
                   (Yes (LTESucc (LTESucc (LTESucc LTEZero)))))]) :
(List (String, Nat), List Assertion)
```

When having a left-hand side greater than 1 (Appendix A.3, Listing A.7) , we can more clearly see the recursive step. Here, the `LTESucc` function is applied three times to the `LTEZero` proof, thereby proving that the left-hand side '3' reached 0 before, or at the same time as, the right-hand side and therefore it is LTE to the right-hand side.

Listing 6.13: A C program requiring $1 \leq 0$

```c
void main(void)
{
    __teamplay_assert(1 <= 0);
}
```

To test that the `LTE` operator can prove that numbers cannot be LTE to each other, we first test with 0 and its immediate successor (1), as they should not require any recursive steps.

```
module Examples.LTE

import Darknet

mutual
  lte_1_0 : CLang
  lte_1_0 = Assert lte_1_0_assert
          $ Halt

  lte_1_0_assert : Env -> Assertion
  lte_1_0_assert env =
    let
      x = Lit 1
      y = Lit 0
      x' = eval env x
      y' = eval env y
      prf = isLTE x' y'
    in
      MkAssertion (LTE x y (MkEvald x x') (MkEvald y y') prf)
```

The resulting IDRIS model has the literals 1 and 0 respectively, their evaluation, and a decidable proof of whether the left is LTE to the right (i.e. $1 \leq 0$).

```
Idris> mkCertificate lte_1_0


([],
 [MkAssertion (LTE (Lit 1)
                   (Lit 0)
                   (MkEvald (Lit 1) 1)
                   (MkEvald (Lit 0) 0)
                   (No succNotLTEzero))]) :
(List (String, Nat), List Assertion)
```

Generating a certificate from the model, from the 'No' we can see that it successfully evaluated that 1 is not LTE than 0. The exact proof, `succNotLTEzero`, has the following type:
`succNotLTEzero :  Not (LTE (S m) 0)`
It states that no successor of any natural number `m` could be LTE to zero. So the `LTE` operator successfully works for 1 and its immediate predecessor 0, which is also the base case of natural numbers.

Listing 6.14: A C program requiring $3 \leq 1$

```c
void main(void)
{
    __teamplay_assert(3 <= 1);
}
```

Changing the false expression such that both numbers are greater than 0, we should get more recursive steps to prove that they cannot be LTE to each other.

```idris
module Examples.LTE

import Darknet

mutual
  lte_3_1 : CLang
  lte_3_1 = Assert lte_3_1_assert
          $ Halt

  lte_3_1_assert : Env -> Assertion
  lte_3_1_assert env =
    let
      x = Lit 3
      y = Lit 1
      x' = eval env x
      y' = eval env y
      prf = isLTE x' y'
    in
      MkAssertion (LTE x y (MkEvald x x') (MkEvald y y') prf)
```

The IDRIS model has the literals 3 and 1 instead of 1 and 0 respectively. Apart from that, it is identical to the previous.

```
Idris> mkCertificate lte_3_1


([],
 [MkAssertion (LTE (Lit 3)
                   (Lit 1)
                   (MkEvald (Lit 3) 3)
                   (MkEvald (Lit 1) 1)
                   (No (\x1 => succNotLTEzero (fromLteSucc x1))))]) :
(List (String, Nat), List Assertion)
```

The certificate makes use of the same `succNotLTEzero` proof to prove that 3 cannot be LTE to 1. The recursive step is the `fromLteSucc` function, which is

passed the argument `x1` from a lambda. The `fromLteSucc` function has the following type:
`fromLteSucc :  LTE (S m) (S n) -> LTE m n`
It takes a proof that the successors of two numbers are LTE to each other and concludes that in that case, so are their predecessors. If it was possible to construct this proof for 2 and 0 (the predecessors to 3 and 1), then `succNotLTEzero` states that it cannot exist. So it is never possible that 3 is LTE to 1.

With these examples, covering the base case (0), its immediate successor (1), and numbers larger than this (where recursion is used), we have showed that the `LTE` operator works as intended.

6.1.4 The `LT`, `GTE`, and `GT` operators

For the operators related to `LTE`, i.e. `LT`, `GTE`, and `GT`, the examples are identical to those for `LTE` apart from the operator used in the C code. The `LTE` operator is the crucial one to have proven to work correctly since it is used to define the similar operators (as explained in Section 5.1.4). Below is the output of a couple of examples from the other operators, to show that `LTE` really is used to define these. Since we have shown `LTE` to work correctly, all the related operators also work correctly. The complete examples for the `LT`, `GTE`, and `GT` operators can be found in appenices A.4, A.5, and A.6 respectively.

```
([],
 [MkAssertion (LT (Lit 0)
                  (Lit 1)
                  (MkEvald (Lit 0) 0)
                  (MkEvald (Lit 1) 1)
                  (Yes (LTESucc LTEZero)))]) :
(List (String, Nat), List Assertion)
```

The proof recurses on the successor of the left-hand side (i.e. `S Z`, 1), using the `LTESucc` and `LTEZero`, and thereby proves that since 1 is LTE to 1, then 0 must be LT 1.

```
([],
 [MkAssertion (LT (Lit 0)
                  (Lit 0)
                  (MkEvald (Lit 0) 0)
                  (MkEvald (Lit 0) 0)
                  (No succNotLTEzero))]) :
(List (String, Nat), List Assertion)
```

The proof that 0 cannot be LT 0 is done using the `succNotLTEzero` proof, i.e. since the successor of the left-hand side has to be LTE than 0 (which is impossible), then 0 cannot be LT 0.

```
([],
 [MkAssertion (GTE (Lit 0)
                   (Lit 0)
                   (MkEvald (Lit 0) 0)
                   (MkEvald (Lit 0) 0)
                   (Yes LTEZero))]) :
(List (String, Nat), List Assertion)
```

The proof for $0 \geq 0$ is `LTEZero`, i.e. since the right-hand side is zero and any number is LTE to zero, then the left-hand side has to be GTE to the right.

```
([],
 [MkAssertion (GTE (Lit 0)
                   (Lit 1)
                   (MkEvald (Lit 0) 0)
                   (MkEvald (Lit 1) 1)
                   (No succNotLTEzero))]) :
(List (String, Nat), List Assertion)
```

The proof that 0 cannot be GTE to 1 is done using `succNotLTEzero` on the right-hand side of the expression. The right-hand side contains a successor, and since no successor can be LTE to zero the right-hand side cannot be LTE to the left. And so the left-hand side cannot be GTE to the right.

```
([],
 [MkAssertion (GT (Lit 3)
                  (Lit 1)
                  (MkEvald (Lit 3) 3)
                  (MkEvald (Lit 1) 1)
                  (Yes (LTESucc (LTESucc LTEZero))))]) :
(List (String, Nat), List Assertion)
```

The proof of $3 > 1$ is done by proving that 2 is LTE to 3. This can be seen in the multiple recursive steps of `LTESucc` applied to the `LTEZero` proof. Since 2 reaches 0 before or at the same time as 3 and all numbers are LTE to 0, 2 must be LTE to 3 and so 3 must be GT 1 (see Section 5.1.4).

```
([],
 [MkAssertion (GT (Lit 3)
                  (Lit 3)
                  (MkEvald (Lit 3) 3)
                  (MkEvald (Lit 3) 3)
                  (No (\x1 => succNotLTEzero
                                (fromLteSucc (fromLteSucc
                                    (fromLteSucc x1))))))]) :
(List (String, Nat), List Assertion)
```

The proof that 3 cannot be GT 3 is done by showing that the successor of the right-hand side of the expression (i.e. `S 3`, 4) cannot be LTE to 3 (see Section 5.1.4). This is done similar to the example in Listing 6.14. Since 4 cannot be LTE to 3, 3 cannot be GT 3.

6.1.5 The `Or` Operator

Since 'or' operates on booleans, we needed two `BooleanExpressions`: one `True` and one `False` . We used $1 = 1$ for `True` and $0 = 1$ for `False`

Listing 6.15: A boolean expression which is true

```
module Examples

import Darknet

public export
b_true : Env -> BooleanExpression
b_true env =
  let
    x = Lit 1
    y = Lit 1
    x' = eval env x
    y' = eval env y
    prf = decEq x' y'
  in
    Eq x y (MkEvald x x') (MkEvald y y') prf
```

The '`b_true`' function is a boolean expression which evaluates to `True` over an environment. In this case, $1 = 1$.

Listing 6.16: A boolean expression which is false

```
module Examples.Not.False

import Darknet

public export
b_false : Env -> BooleanExpression
b_false env =
  let
    x = Lit 0
    y = Lit 1
    x' = eval env x
    y' = eval env y
    prf = decEq x' y'
  in
    Eq x y (MkEvald x x') (MkEvald y y') prf
```

The '`b_false`' function is a boolean expression which evaluates to `False` over an environment. In this case, $0 = 1$.

The truth-table for logical 'or' is:

a	b	a ∨ b
T	T	T
T	F	T
F	T	T
F	F	F

Listing 6.17: A C program where both 'or' operands happen to be true

```
void main(void)
{
    __teamplay_assert((1 == 1) || (1 == 1));
}
```

It is possible that a program would at some point assert that at least one of two properties held. And that both properties happened to hold. Such a program would have a similar assertion to the above.

```
module Examples.Or

import Darknet
import Examples.True

mutual
  or_t_t : CLang
  or_t_t = Assert or_t_t_assert
           $ Halt

  or_t_t_assert : Env -> Assertion
  or_t_t_assert env =
    let
      t1 = b_true env
      t2 = b_true env
      t1' = beval env t1
      t2' = beval env t2
      prf = isOr t1' t2'
    in
      MkAssertion (Or t1 t2 (MkBEvald t1 t1') (MkBEvald t2 t2') prf)
```

The resulting Idris model contains two boolean expressions which should evaluate to `True` (Listing 6.15), their evaluation over a given environment, and a decidable proof of whether the 'or' expression evaluates to true.

```
([],
 [MkAssertion (Or (Eq (Lit 1) (Lit 1) (MkEvald (Lit 1) 1)
                                       (MkEvald (Lit 1) 1)
                                       (Yes Refl))
                  (Eq (Lit 1) (Lit 1) (MkEvald (Lit 1) 1)
                                       (MkEvald (Lit 1) 1)
                                       (Yes Refl))
                  (MkBEvald (Eq (Lit 1)
                                (Lit 1)
                                (MkEvald (Lit 1) 1)
                                (MkEvald (Lit 1) 1)
                                (Yes Refl))
                            True)
                  (MkBEvald (Eq (Lit 1)
                                (Lit 1)
                                (MkEvald (Lit 1) 1)
                                (MkEvald (Lit 1) 1)
                                (Yes Refl))
                            True)
                  (Yes MkOr))]) :
(List (String, Nat), List Assertion)
```

When generating the certificate, we can see that the true boolean expression has been filled in where the `b_true` used to be, and that the expressions held true:

both `MkBevald`s have `True` at the end. The final argument to the `Or` function is a `Yes MkOr`, i.e. the operator has evaluated to `True` due to both arguments being true.

Listing 6.18: A C program where the left 'or' operand happens to be true

```
void main(void)
{
    __teamplay_assert((1 == 1) || (0 == 1));
}
```

Another possibility is that one of the properties could be true and the other false. This should still evaluate to true.

```
module Examples.Or

import Darknet
import Examples.True
import Examples.False

mutual
  or_t_f : CLang
  or_t_f = Assert or_t_f_assert
           $ Halt

  or_t_f_assert : Env -> Assertion
  or_t_f_assert env =
    let
      t1 = b_true env
      f2 = b_false env
      t1' = beval env t1
      f2' = beval env f2
      prf = isOr t1' f2'
    in
      MkAssertion (Or t1 f2 (MkBEvald t1 t1') (MkBEvald f2 f2') prf)
```

The resulting model now contains both a true and a false boolean expression (from Listing 6.15 and 6.16 respectively). Apart from that, it is unchanged.

```
([],
 [MkAssertion (Or (Eq (Lit 1) (Lit 1) (MkEvald (Lit 1) 1)
                                      (MkEvald (Lit 1) 1)
                                      (Yes Refl))
                  (Eq (Lit 0) (Lit 1) (MkEvald (Lit 0) 0)
                                      (MkEvald (Lit 1) 1)
                                      (No ZnotS))
                  (MkBEvald (Eq (Lit 1)
                                (Lit 1)
                                (MkEvald (Lit 1) 1)
                                (MkEvald (Lit 1) 1)
                                (Yes Refl))
                            True)
                  (MkBEvald (Eq (Lit 0)
                                (Lit 1)
                                (MkEvald (Lit 0) 0)
                                (MkEvald (Lit 1) 1)
                                (No ZnotS))
                            False)
                  (Yes MkOrL))]) :
(List (String, Nat), List Assertion)
```

From the certificate we can see that the boolean expressions were correctly evaluated: the first evaluated to `True` and the second to `False`. The final line before the type declaration is `Yes MkOrL`, so the operator has correctly showed that the expression evaluates to `True`: the left argument held true.

```
([],
 [MkAssertion (Or (Eq (Lit 0) (Lit 1) (MkEvald (Lit 0) 0)
                                      (MkEvald (Lit 1) 1)
                                      (No ZnotS))
                  (Eq (Lit 1) (Lit 1) (MkEvald (Lit 1) 1)
                                      (MkEvald (Lit 1) 1)
                                      (Yes Refl))
                  (MkBEvald (Eq (Lit 0)
                                (Lit 1)
                                (MkEvald (Lit 0) 0)
                                (MkEvald (Lit 1) 1)
                                (No ZnotS))
                            False)
                  (MkBEvald (Eq (Lit 1)
                                (Lit 1)
                                (MkEvald (Lit 1) 1)
                                (MkEvald (Lit 1) 1)
                                (Yes Refl))
                            True)
                  (Yes MkOrR))]) :
(List (String, Nat), List Assertion)
```

Swapping the true and false boolean expressions (Appendix A.7, Listing A.26), we get the above certificate. Again, the boolean expressions have been evaluated correctly: the first is `False` and the second is `True`. This time, the proof for 'or' is a `Yes MkOrR`, so the operator functioned correctly: The 'or' expression evaluates to `True` because the right argument held true.

Listing 6.19: A C program where both 'or' operands happen to be false

```
void main(void)
{
    __teamplay_assert((0 == 1) || (0 == 1));
}
```

Finally, it is possible that both properties did not hold. In this case, the operator should evaluate to `False`, i.e it should be impossible to construct one of the previous proofs.

```
module Examples.Or

import Darknet
import Examples.False

mutual
  or_f_f : CLang
  or_f_f = Assert or_f_f_assert
           $ Halt

  or_f_f_assert : Env -> Assertion
  or_f_f_assert env =
    let
      f1 = b_false env
      f2 = b_false env
      f1' = beval env f1
      f2' = beval env f2
      prf = isOr f1' f2'
    in
      MkAssertion (Or f1 f2 (MkBEvald f1 f1') (MkBEvald f2 f2') prf)
```

The corresponding IDRIS model's boolean expressions are both the ones that evaluate to `False` (Listing 6.16). Apart from that, it is identical to the previous ones.

```
([],
 [MkAssertion (Or (Eq (Lit 0) (Lit 1) (MkEvald (Lit 0) 0)
                                       (MkEvald (Lit 1) 1)
                                       (No ZnotS))
                  (Eq (Lit 0) (Lit 1) (MkEvald (Lit 0) 0)
                                       (MkEvald (Lit 1) 1)
                                       (No ZnotS))
                  (MkBEvald (Eq (Lit 0)
                                (Lit 1)
                                (MkEvald (Lit 0) 0)
                                (MkEvald (Lit 1) 1)
                                (No ZnotS))
                            False)
                  (MkBEvald (Eq (Lit 0)
                                (Lit 1)
                                (MkEvald (Lit 0) 0)
                                (MkEvald (Lit 1) 1)
                                (No ZnotS))
                            False)
                  (No absurd))]) :
(List (String, Nat), List Assertion)
```

The resulting certificate shows that both boolean expressions correctly evaluated to `False`. Furthermore, the final part is a '`No absurd`' which shows that the operator correctly determined that the 'or'-expression cannot be true, using the `Uninhabited` implementations for `TyOr` (Listing 5.37) to prove this.

All the cases evaluate to the value predicted from the truth-table. As such, we can conclude that the `Or` operator works correctly.

6.1.6 The `Not` Operator

The `True` and `False` boolean expressions used to evaluate `Not` are the same as in `Or`, i.e. Listings 6.15 and 6.16.

The truth-table for logical negation is:

p	¬p
F	T
T	F

Listing 6.20: A C program where the property that must not hold is false

```c
void main(void)
{
    __teamplay_assert(!(0 == 1));
}
```

If a program was trying to assert that a property did not hold, then it would be using the logical 'not' operator. In the desired case, this means the property itself evaluates to false. This situation would be similar to the one in the program above.

```
import Darknet
import Examples.False

mutual
  not_false : CLang
  not_false = Assert not_false_assert
            $ Halt

  not_false_assert : Env -> Assertion
  not_false_assert env =
    let
      f = b_false env
      f' = beval env f
      prf = isNot f'
    in
      MkAssertion (Not f (MkBEvald f f') prf)
```

The corresponding Idris model only has one boolean expression which should evaluate to `False` (Listing 6.16), its evaluation over a given environment, and a proof of whether its negation is true.

```
([],
 [MkAssertion (Not (Eq (Lit 0) (Lit 1) (MkEvald (Lit 0) 0)
                                        (MkEvald (Lit 1) 1)
                                        (No ZnotS))
                   (MkBEvald (Eq (Lit 0)
                                 (Lit 1)
                                 (MkEvald (Lit 0) 0)
                                 (MkEvald (Lit 1) 1)
                                 (No ZnotS))
                             False)
                   (Yes MkBNot))]) :
(List (String, Nat), List Assertion)
```

Making the certificate, we can see that the boolean expression was evaluated to `False` (in the `BEvald`) and that the final argument is a `Yes MkBNot`. This shows that the operator returns `True` when the input is `False`, as it should.

Listing 6.21: A C program where the property that must not hold is true

```
void main(void)
{
    __teamplay_assert(!(1 == 1));
}
```

In the undesired case, the property that must not hold does. This means that the expression should evaluate to `False`. This situation would be similar to the one in the program above.

```
import Darknet
import Examples.True

mutual
  not_true : CLang
  not_true = Assert not_true_assert
             $ Halt

  not_true_assert : Env -> Assertion
  not_true_assert env =
    let
      t = b_true env
      t' = beval env t
      prf = isNot t'
    in
      MkAssertion (Not t (MkBEvald t t') prf)
```

The resulting IDRIS model again only has one boolean expression. However, this time it should evaluate to `True` (Listing 6.15). Apart from this, the model is identical to the previous.

```
([],
 [MkAssertion (Not (Eq (Lit 1) (Lit 1) (MkEvald (Lit 1) 1)
                                       (MkEvald (Lit 1) 1)
                                       (Yes Refl))
                   (MkBEvald (Eq (Lit 1)
                                 (Lit 1)
                                 (MkEvald (Lit 1) 1)
                                 (MkEvald (Lit 1) 1)
                                 (Yes Refl))
                             True)
                   (No absurd))]) :
(List (String, Nat), List Assertion)
```

Looking at the certificate produced, we can see that the boolean expression correctly evaluated to `True` and that the final argument is a '`No absurd`'. This means that the operator has proved that the expression cannot evaluate to `True`, using the `Uninhabited` implementation of `Not` (Listing 5.41), and as such, the operator functions correctly when negating a `True` expression.

The results from the operator correspond to the truth-table for boolean negation. Therefore, the `Not` operator works as intended.

With a set of operators verified to be correct, I constructed and modelled some more complex examples.

6.2 Small Programs

This section shows how annotations and operators can be combined to assert extra-functional properties of constructs like `for`-loops and `if`-statements. Since these are very widely used in programming, being able to model and assert things about them is crucial if the framework is to be used on real-world programs.

6.2.1 Loop accumulation

Listing 6.22: A C program whose loop has a timing requirement.

```
void main(void) {
    __teamplay_worst_time(measured);
    for (int i = 0; i < 100; i++) {
        // ...
        // some operations that take some time
        // ...

        __teamplay_loop_time_acc(acc);
    }

    __teamplay_assert(acc <= measured);
}
```

The `__teamplay_worst_time` annotation states that the most time the block below (i.e. the loop) must take has been measured and stored in the variable '`measured`'. The `__teamplay_loop_time_acc` annotation indicates that the accumulated time taken after each iteration of the loop is stored in the variable '`acc`'. Finally, the `__teamplay_assert` annotation specifies that this program requires the accumulated time of the loop to be less-than-or-equal to the time we measured that block of code takes, `acc <= measured`.

This program makes use of variables in to annotations. These would appear in the CSV file containing the timing measurements. The environment should reflect this and contain both variables' names and their corresponding values.

```
module Examples.Advanced.Loop

import Darknet

mutual
  loop : CLang
  loop = BlockTime "measured" 3
         $ BlockTime "acc" 2  -- dummy value illustrating 0.02
                              -- time units over 100 iterations
         $ Assert loop_assert
         $ Halt

  loop_assert : Env -> Assertion
  loop_assert env =
    let
      p0 = Var "measured"
      p1 = Var "acc"
      p0' = eval env p0
      p1' = eval env p1
      prf = isLTE p1' p0'
    in
      MkAssertion
        (LTE p1 p0 (MkEvald p1 p1') (MkEvald p0 p0') prf)
```

In the corresponding IDRIS model, "`loop`", the '`__teamplay_worst_time`' annotation had a time of 3 time units, captured in a `BlockTime`. I modelled the '`__teamplay_loop_time_acc`' annotation as a `BlockTime` construct with the variable `acc` and a value of 2. The value illustrates 0.02 time units per iteration, times the 100 iterations that the loop had. Since `acc` was declared later than `measured`, it is the second property, which is correctly captured in the `prf`. The properties, values, and proof are passed to the `LTE` function which is in turn passed to the `MkAssertion` function. This completes the model of the measurements and assertions in the C program.

```
Idris> mkCertificate loop


([("acc", 2), ("measured", 3)],
 [MkAssertion (LTE (Var "acc")
                   (Var "measured")
                   (MkEvald (Var "acc") 2)
                   (MkEvald (Var "measured") 3)
                   (Yes (LTESucc (LTESucc LTEZero))))]) :
(List (String, Nat), List Assertion)
```

Creating the certificate for `loop`, we can see that the environment correctly contains the two variables '`acc`' and '`measured`' associated with the correct values

(2 and 3 respectively). From the two `MkEvald`s, we can see that the environment has been correctly used to evaluate the variables. Finally, the `Yes` part of the certificate shows us that the assertion held: the property `acc` $\leq$ `measured` held.

6.2.2 Branching

Listing 6.23: A C program which requires equal energy consumption between branches

```
void main(void)
{
    __teamplay_branch_energy(b1, b2);
    if (/*condition*/)
    {
        // ...
        // some operations which happen to use less energy
        // ...
    }
    else
    {
        // ...
        // some operations which happen to use more energy
        // ...
    }

    __teamplay_assert(b1 == b2);
}
```

The `__teamplay_branch_energy` annotation measures the energy consumption of the branches of an `if`-statement and stores them in two variables. The program then uses an assertion to require that the branches use the same amount of energy. However, for some reason, they do not. This would be represented in the measurements CSV-file passed to the model-builder.

```
module Examples.Advanced.If_then_else

import Darknet

mutual
  if_then_else : CLang
  if_then_else = BlockEnergy "b1" 2
               $ BlockEnergy "b2" 5
               $ Assert if_then_else_assert
               $ Halt

  if_then_else_assert : Env -> Assertion
  if_then_else_assert env =
    let
      p0 = Var "b1"
      p1 = Var "b2"
      p0' = eval env p0
      p1' = eval env p1
      prf = decEq p0' p1'
    in
      MkAssertion (Eq p0 p1 (MkEvald p0 p0') (MkEvald p1 p1') prf)
```

In the model, I have taken the `__teamplay_branch_energy` annotation to produce two `BlockEnergy` constructs with the variable names declared in the C file and the energy measurements of 2 and 5 energy units respectively. The proof, `prf`, is done using the built-in `decEq` function on the evaluated values `p0'` and `p1'`. The equality is captured by the `Eq` function, and the assertion by the `MkAssertion` function.

```
Idris> mkCertificate if_then_else


([("b2", 5), ("b1", 2)],
 [MkAssertion (Eq (Var "b1")
                  (Var "b2")
                  (MkEvald (Var "b1") 2)
                  (MkEvald (Var "b2") 5)
                  (No (\h =>
                         ZnotS (succInjective 0 3
                                 (succInjective 1 4 h)))))]) :
(List (String, Nat), List Assertion)
```

In the resulting certificate we can see the environment contains the same variables as used in the model. As a result, the variables have been correctly associated with their values in the `MkEvalds`. Since one of the energy measurements was 2 and the other 5, the equality did not hold. This can be seen in the `No` part of the certificate. The exact proof shows that this is because 0 cannot be the successor of any number (in this case 3) and hence the numbers cannot be equal. Furthermore, this implies (`succInjective`) that 1 cannot be the successor of 4, and that 2 cannot be the successor of 5, so the numbers cannot be equal.

6.2.3 Statements

```c
void main(void)
{
    // ...
    // function definitions
    // ...

    __teamplay_worst_time(rd_time);
    void* val = readVal();

    __teamplay_worst_time(wr_time);
    __teamplay_worst_energy(wr_energy);
    int err = writeVal(val);

    __teamplay_assert((wr_time > rd_time) && (wr_energy <= 10));
}
```

This C program has a timing constraint on its read and write statements. The write time must be greater than the read time. Also, the write energy must be LTE to 10.

```
module Examples.Advanced.Statements

import Darknet

mutual
  statements : CLang
  statements = StmtTime "rd_time" 3
             $ StmtTime "wr_time" 4
             $ StmtEnergy "wr_energy" 9
             $ Assert statements_assert
             $ Halt

  statements_assert : Env -> Assertion
  statements_assert env =
    let
      p0 = Var "rd_time"
      p1 = Var "wr_time"
      p2 = Var "wr_energy"
      p3 = Lit 10
      p0' = eval env p0
      p1' = eval env p1
      p2' = eval env p2
      p3' = eval env p3
      prf0 = isLTE (S p0') p1' -- wr_time > rd_time
      prf1 = isLTE p2' p3'     -- wr_energy <= 10
    in
      let
        bexpr0 = GT p1 p0
                   (MkEvald p1 p1') (MkEvald p0 p0')
                   prf0
        bexpr1 = LTE p2 p3
                    (MkEvald p2 p2') (MkEvald p3 p3')
                    prf1
        bexpr0' = beval env bexpr0
        bexpr1' = beval env bexpr1
        prf = isAnd bexpr0' bexpr1'
              -- ((wr_time > rd_time) && (wr_energy <= 10)
      in
        MkAssertion
          (And bexpr0 bexpr1
               (MkBEvald bexpr0 bexpr0')
               (MkBEvald bexpr1 bexpr1')
               prf)
```

The resulting Idris model is quite big despite the simple program. The expression was split up into two `let/in` statements to facilitate the reading. The mechanics are the same as the previous examples, with more components. The proof of the expression is omitted for brevity. It can be found in Appendix A.8.

6.3 Real programs

6.3.1 AES encryption

Listing 6.24: The annotated AES code from Chapter 5

```
⋮
__teamplay_worst_energy(addRoundKey0);
add_round_key(state, w, 0);

for (r = 1; r < Nr; r++) {
    __teamplay_worst_energy_acc(subBytesAcc);
    sub_bytes(state);

    __teamplay_worst_energy_acc(shiftRowsAcc);
    shift_rows(state);

    __teamplay_worst_energy_acc(mixColumnsAcc);
    mix_columns(state);

    __teamplay_worst_energy_acc(addRoundKeyAcc);
    add_round_key(state, w, r);
}

__teamplay_worst_energy(subBytes);
sub_bytes(state);

__teamplay_worst_energy(shiftRows);
shift_rows(state);

__teamplay_worst_energy(addRoundKeyNr);
add_round_key(state, w, Nr);

__teamplay_assert(addRoundKey0 + subBytesAcc +
                  shiftRowsAcc + mixColumnsAcc +
                  addRoundKeyAcc + subBytes +
                  shiftRows + addRoundKeyNr
                  <= 50)
⋮
```

```
module Examples.Real_Life.Aes

import Darknet

mutual
  aes : CLang
  aes = StmtEnergy "addRoundKey0" 2
        $ StmtEnergy "subBytesAcc" 10
        $ StmtEnergy "shiftRowsAcc" 3
        $ StmtEnergy "mixColumnsAcc" 5
        $ StmtEnergy "addRoundKeyAcc" 19
        $ StmtEnergy "subBytes" 1
        $ StmtEnergy "shiftRows" 7
        $ StmtEnergy "addRoundKeyNr" 3
        $ Assert aes_assert
        $ Halt


  aes_assert : Env -> Assertion
  aes_assert env =
    let
      p0 = Var "addRoundKey0"
      p1 = Var "subBytesAcc"
      p2 = Var "shiftRowsAcc"
      p3 = Var "mixColumnsAcc"
      p4 = Var "addRoundKeyAcc"
      p5 = Var "subBytes"
      p6 = Var "shiftRows"
      p7 = Var "addRoundKeyNr"
      p8 = Lit 50
      p8' = eval env p8
      sum = Plus (Plus (Plus (Plus (Plus
                 (Plus (Plus p7 p6)
                  p5) p4) p3) p2) p1) p0
      sum' = eval env sum
      prf = isLTE sum' p8'
    in
      MkAssertion (LTE sum p8 (MkEvald sum sum') (MkEvald p8 p8') prf)
```

Once again, the resulting IDRIS model is very big. This is due to the multiple variables being defined and evaluated. The sum of the numbers is a variable which is also evaluated. This is mostly for readability, but a sophisticated parser could perhaps do something similar. The proof that the expression holds is massive due to the numbers involved, but can be found in Appendix A.9.

Chapter 7

Conclusion

This dissertation has discussed and evaluated the use of dependent types as a way of proving extra-functional properties of programs for embedded systems. The existing work by the TeamPlay project has been extended to feature a complete set of operators, and these have all been evaluated and shown to work successfully. In evaluating the operators, multiple examples of how annotated programs are translated into Idris models have been given.

Programs having more complex features like branches or loops were successfully provided and modelled, demonstrating that the basic TeamPlay constructs can be combined in different ways to model larger parts of a program. Examples of a real-world program using the framework was provided and successfully modelled, demonstrating that the TeamPlay framework scales to the complexity of real-world programs. The C programs given as part of these show that the TeamPlay annotations seamlessly integrate with the code and are intuitively named.

There is still much work to be done. A parser for the TeamPlay-annotated C code and CSV files is currently work in progress by the part of the TeamPlay project at St Andrews, and would drastically improve the time it takes to construct the Idris models as it would completely automate this. Evaluating the project with real values would also be very useful. However, this is beyond the scope of division of the TeamPlay project at St Andrews. Even more complicated programs with nested loops or branches could be generated given a parser. Having these would strengthen the demonstration of the potential of the framework. Finally, expanding the framework to be able to handle open-ended assertions (i.e. assertions where one or more variables are unknown) would be useful as it would enable programmers to get a proof of how much (or how little) margin they have in terms of time or energy. However, doing this is complicated and would require functionality along the lines of a constraint solver for the framework.

Acknowledgements

I would like to thank my supervisors at the School of Computer Science for their help and guidance with the project. Specifically, I would like to thank Dr. Edwin Brady for lending me a copy of his book on Idris and helping me understand the details of the language, and Dr.s Chris Brown and Adam Barwell for their patience and help with introducing me to the TeamPlay project. I would also like to thank all the members of the TeamPlay project for allowing me to attend their research seminar in St Andrews.

Bibliography

[1] T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. Times b— a tool for modelling and implementation of embedded systems. In J.-P. Katoen and P. Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 460–464, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[2] E. Axelsson, K. Claessen, G. Dévai, Z. Horváth, K. Keijzer, B. Lyckegård, A. Persson, M. Sheeran, J. Svenningsson, and A. Vajdax. Feldspar: A domain specific language for digital signal processing algorithms. In *Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2010)*, pages 169–178, July 2010.

[3] S. Bhatti, E. Brady, K. Hammond, and J. McKinna. Domain specific languages (dsls) for network protocols (position paper). In *2009 29th IEEE International Conference on Distributed Computing Systems Workshops*, pages 208–213, June 2009.

[4] I. Board. Ariane 5 flight 501 failure, report by the inquiry board. *Paris, July*, 19, 1996.

[5] C. Borrás. Overexposure of radiation therapy patients in panama: problem recognition and follow-up measures. *Revista Panamericana de Salud Pública*, 20:173–187, 2006.

[6] E. Brady. Ivor, a proof engine. In Z. Horváth, V. Zsók, and A. Butterfield, editors, *Implementation and Application of Functional Languages*, pages 145–162, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[7] E. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23(5):552–593, 2013.

[8] E. Brady. *Type-Driven Development with Idris*. Manning, 2017.

[9] E. Brady and K. Hammond. Correct-by-construction concurrency: Using dependent types to verify implementations of effectful resource usage protocols. *Fundamenta Informaticae*, 102(2):145–176, 2010.

[10] E. Brady and K. Hammond. Resource-safe systems programming with embedded domain specific languages. In C. Russo and N.-F. Zhou, editors, *Practical Aspects of Declarative Languages*, pages 242–257, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[11] C. Brown. Report on code-level contracts for energy, time and security. https://gitlab.inria.fr/TeamPlay_Public/TeamPlay_Public_Deliverables/blob/master/D1.1.pdf. [Online; Accessed Mar 2019].

[12] A. Burgess, A. Whetter, G. Field, G. Markall, H. Oosenbrug, J. Pallister, J. Bennett, N. Grech, P. Langlois, and S. Cook. Beebs: Open benchmarks for energy measurements on embedded platforms. https://github.com/mageec/beebs. [Online; Accessed April 2019].

[13] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. *Progress on the State Explosion Problem in Model Checking*, pages 176–194. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[14] E. M. Clarke, W. Klieber, M. Nováček, and P. Zuliani. *Model Checking and the State Explosion Problem*, pages 1–30. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[15] S. Croomes. Overview of the dart mishap investigation results. *NASA Report*, pages 1–10, 2006.

[16] D. Evans. The internet of things. https://www.cisco.com/c/dam/en_us/about/ac79/docs/innov/IoT_IBSG_0411FINAL.pdf, April 2011. [Online; Accessed March 2019].

[17] K. Hammond and G. Michaelson. Hume: A domain-specific language for real-time embedded systems. In F. Pfenning and Y. Smaragdakis, editors, *Generative Programming and Component Engineering*, pages 37–56, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[18] P. Hudak. Modular domain specific languages and tools. In *Proceedings. Fifth International Conference on Software Reuse (Cat. No.98TB100203)*, pages 134–142, June 1998.

[19] P. Hudak et al. Building domain-specific embedded languages. *ACM Comput. Surv.*, 28(4es):196, 1996.

[20] D. Huertas. Aes algorithm implementation in c. https://github.com/dhuertas/AES. [Online; Accessed April 2019].

[21] M. Johansson, D. Rosén, N. Smallbone, and K. Claessen. Hipster: Integrating theory exploration in a proof assistant. In S. M. Watt, J. H. Davenport, A. P. Sexton, P. Sojka, and J. Urban, editors, *Intelligent Computer Mathematics*, pages 108–122, Cham, 2014. Springer International Publishing.

[22] M. Kwiatkowska, G. Norman, and D. Parker. Prism 4.0: Verification of probabilistic real-time systems. In G. Gopalakrishnan and S. Qadeer, editors, *Computer Aided Verification*, pages 585–591, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[23] K. G. Larsen, P. Pettersson, and W. Yi. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1):134–152, Dec 1997.

[24] J. Lee, J. Kim, C. Jang, S. Kim, B. Egger, K. Kim, and S. Han. Facsim: A fast and cycle-accurate architecture simulator for embedded systems. In *Proceedings of the 2008 ACM SIGPLAN-SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES '08, pages 89–100, New York, NY, USA, 2008. ACM.

[25] B. Schlich and S. Kowalewski. Model checking c source code for embedded systems. *International Journal on Software Tools for Technology Transfer*, 11(3):187–202, Jul 2009.

[26] T. Simunic, L. Benini, and G. De Micheli. Cycle-accurate simulation of energy consumption in embedded systems. In *Proceedings 1999 Design Automation Conference (Cat. No. 99CH36361)*, pages 867–872, June 1999.

[27] A. Valmari. *The state explosion problem*, pages 429–528. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998.

Appendix A

Evaluation

A.1 Eq operator

Listing A.1: A C program which requires $1 = 0$

```
void main(void)
{
    __teamplay_assert(1 == 0);
}
```

A correct C program asserting the vice-versa of the example in Section 6.1.1, Listing 6.3.

```
import Darknet

mutual
  eq_1_0 : CLang
  eq_1_0 = Assert eq_1_0_assert
           $ Halt

  eq_1_0_assert : Env -> Assertion
  eq_1_0_assert env =
    let
      x = Lit 1
      y = Lit 0
      x' = eval env x
      y' = eval env y
      prf = decEq x' y'
    in
      MkAssertion (Eq x y (MkEvald x x') (MkEvald y y') prf)
```

The resulting model '`eq_1_0`' consists of an assertion '`eq_1_0_assert`' and nothing more. This assertion contains the literals 1 and 0 (assigned to `x` and `y` respectively), their evaluation (`x'` and `y'`), and a decidable proof of whether they are equal.

Listing A.2: A C program which requires $3 = 1$

```
void main(void)
{
    __teamplay_assert(3 == 1);
}
```

A valid C program which asserts that if we swap the operands of Listing 6.5, then the equality should not hold.

```
import Darknet

mutual
  eq_3_1 : CLang
  eq_3_1 = Assert eq_3_1_assert
         $ Halt

  eq_3_1_assert : Env -> Assertion
  eq_3_1_assert env =
    let
      x = Lit 3
      y = Lit 1
      x' = eval env x
      y' = eval env y
      prf = decEq x' y'
    in
      MkAssertion (Eq x y (MkEvald x x') (MkEvald y y') prf)
```

The resulting model has the literals swapped, but apart from that it is identical to the model for Listing 6.5.

[to return to Eq: Section 6.1.1]

A.2 NEq operator

Listing A.3: A C program which requires $1 \neq 0$

```c
void main(void)
{
    __teamplay_assert(1 != 0);
}
```

A syntactically correct C program asserting the same as Listing 6.6 but with the operands swapped.

```idris
import Darknet

mutual
  neq_1_0 : CLang
  neq_1_0 = Assert neq_1_0_assert
          $ Halt

  neq_1_0_assert : Env -> Assertion
  neq_1_0_assert env =
    let
      x = Lit 1
      y = Lit 0
      x' = eval env x
      y' = eval env y
      prf = isNEq x' y'
    in
      MkAssertion (NEq x y (MkEvald x x') (MkEvald y y') prf)
```

The resulting IDRIS model. The literals 0 and 1 have been swapped compared to the model for Listing 6.6, but apart from that, the model is the same.

Listing A.4: A C program which requires $1 \neq 1$

```c
void main(void)
{
    __teamplay_assert(1 != 1);
}
```

A valid C program that requires 1 to be not equal to 1, similar to Listing 6.7 but with the immediate successor of 0.

```
import Darknet

mutual
  neq_1_1 : CLang
  neq_1_1 = Assert neq_1_1_assert
            $ Halt

  neq_1_1_assert : Env -> Assertion
  neq_1_1_assert env =
    let
      x = Lit 1
      y = Lit 1
      x' = eval env x
      y' = eval env y
      prf = isNEq x' y'
    in
      MkAssertion (NEq x y (MkEvald x x') (MkEvald y y') prf)
```

The resulting model contains two literals, both carrying the value 1, their evaluation, and a decidable proof of whether they are not equal.

Listing A.5: A C program which requires $5 \neq 2$

```c
void main(void)
{
    __teamplay_assert(5 != 2);
}
```

A correct C program which requires 5 to not be equal to 2, swapping the order of the operands from Listing 6.8.

```
import Darknet

mutual
  neq_5_2 : CLang
  neq_5_2 = Assert neq_5_2_assert
                $ Halt

  neq_5_2_assert : Env -> Assertion
  neq_5_2_assert env =
    let
      x = Lit 5
      y = Lit 2
      x' = eval env x
      y' = eval env y
      prf = isNEq x' y'
```

```
in
  MkAssertion (NEq x y (MkEvald x x') (MkEvald y y') prf)
```

The resulting IDRIS model. The numbers 2 and 5 have swapped order compared to the model for Listing 6.8, to reflect the same swap made in the C program.

[to return to NEq: Section 6.1.2]

A.3 LTE operator

Listing A.6: A C program which requires $0 \leq 1$

```
void main(void)
{
    __teamplay_assert(0 <= 1);
}
```

This C program has the same structure as the one in Listing 6.10, this time asserting $0 \leq 1$.

```
module Examples.LTE

import Darknet

mutual
  lte_0_1 : CLang
  lte_0_1 = Assert lte_0_1_assert
          $ Halt

  lte_0_1_assert : Env -> Assertion
  lte_0_1_assert env =
    let
      x = Lit 0
      y = Lit 1
      x' = eval env x
      y' = eval env y
      prf = isLTE x' y'
    in
      MkAssertion (LTE x y (MkEvald x x') (MkEvald y y') prf)
```

Translating this to an IDRIS model we get two literals carrying the values 0 and 1 respectively, their evaluation over a given environment, and a decidable proof of whether the first is less than or equal to the first.

Listing A.7: A C program requiring $3 \leq 3$

```
void main(void)
{
    __teamplay_assert(3 <= 3);
}
```

```
module Examples.LTE

import Darknet

mutual
  lte_3_3 : CLang
  lte_3_3 = Assert lte_3_3_assert
          $ Halt

  lte_3_3_assert : Env -> Assertion
  lte_3_3_assert env =
    let
      x = Lit 3
      y = Lit 3
      x' = eval env x
      y' = eval env y
      prf = isLTE x' y'
    in
      MkAssertion (LTE x y (MkEvald x x') (MkEvald y y') prf)
```

The corresponding IDRIS model.

[to return to LTE: Section 6.1.3]

A.4 LT operator

Listing A.8: A C program which requires $0 < 0$

```c
void main(void)
{
    __teamplay_assert(0 < 0);
}
```

```idris
module Examples.LT

import Darknet

mutual
  lt_0_0 : CLang
  lt_0_0 = Assert lt_0_0_assert
           $ Halt

  lt_0_0_assert : Env -> Assertion
  lt_0_0_assert env =
    let
      x = Lit 0
      y = Lit 0
      x' = eval env x
      y' = eval env y
      prf = isLTE (S x') y'
    in
      MkAssertion (LT x y (MkEvald x x') (MkEvald y y') prf)
```

The corresponding IDRIS model. Note that the proof, `prf`, is derived from the successor of the left-hand side.

```
([],
 [MkAssertion (LT (Lit 0)
                  (Lit 0)
                  (MkEvald (Lit 0) 0)
                  (MkEvald (Lit 0) 0)
                  (No succNotLTEzero))]) :
(List (String, Nat), List Assertion)
```

0 is not LT 0, since 1 cannot be LTE to 0.

[to return to LT, GTE, and GT: Section 6.1.4]

Listing A.9: A C program which requires $0 < 1$

```c
void main(void)
{
    __teamplay_assert(0 < 1);
}
```

```idris
module Examples.LT

import Darknet

mutual
  lt_0_1 : CLang
  lt_0_1 = Assert lt_0_1_assert
           $ Halt

  lt_0_1_assert : Env -> Assertion
  lt_0_1_assert env =
    let
      x = Lit 0
      y = Lit 1
      x' = eval env x
      y' = eval env y
      prf = isLTE (S x') y'
    in
      MkAssertion (LT x y (MkEvald x x') (MkEvald y y') prf)
```

The corresponding IDRIS model. Note that the proof, `prf`, is derived from the successor of the left-hand side.

```idris
([],
 [MkAssertion (LT (Lit 0)
                  (Lit 1)
                  (MkEvald (Lit 0) 0)
                  (MkEvald (Lit 1) 1)
                  (Yes (LTESucc LTEZero)))]) :
(List (String, Nat), List Assertion)
```

1 is LTE to 1, so 0 is LT 1.

[to return to LT, GTE, and GT: Section 6.1.4]

Listing A.10: A C program which requires $1 < 0$

```
void main(void)
{
    __teamplay_assert(1 < 0);
}
```

```
module Examples.LT

import Darknet

mutual
  lt_1_0 : CLang
  lt_1_0 = Assert lt_1_0_assert
           $ Halt

  lt_1_0_assert : Env -> Assertion
  lt_1_0_assert env =
    let
      x = Lit 1
      y = Lit 0
      x' = eval env x
      y' = eval env y
      prf = isLTE (S x') y'
    in
      MkAssertion (LT x y (MkEvald x x') (MkEvald y y') prf)
```

The corresponding Idris model. Note that the proof, `prf`, is derived from the successor of the left-hand side.

```
([],
 [MkAssertion (LT (Lit 1)
                  (Lit 0)
                  (MkEvald (Lit 1) 1)
                  (MkEvald (Lit 0) 0)
                  (No succNotLTEzero))]) :
(List (String, Nat), List Assertion)
```

Since 2 cannot be LTE to 0, 1 cannot be LT 0.

[to return to LT, GTE, and GT: Section 6.1.4]

Listing A.11: A C program which requires $1 < 3$

```c
void main(void)
{
    __teamplay_assert(1 < 3);
}
```

```idris
module Examples.LT

import Darknet

mutual
  lt_1_3 : CLang
  lt_1_3 = Assert lt_1_3_assert
           $ Halt

  lt_1_3_assert : Env -> Assertion
  lt_1_3_assert env =
    let
      x = Lit 1
      y = Lit 3
      x' = eval env x
      y' = eval env y
      prf = isLTE (S x') y'
    in
      MkAssertion (LT x y (MkEvald x x') (MkEvald y y') prf)
```

The corresponding IDRIS model. Note that the proof, `prf`, is derived from the successor of the left-hand side.

```
([],
 [MkAssertion (LT (Lit 1)
                  (Lit 3)
                  (MkEvald (Lit 1) 1)
                  (MkEvald (Lit 3) 3)
                  (Yes (LTESucc (LTESucc LTEZero))))]) :
(List (String, Nat), List Assertion)
```

Since 2 is LTE to 3, 1 is LT 3.

[to return to LT, GTE, and GT: Section 6.1.4]

Listing A.12: A C program which requires $3 < 1$

```
void main(void)
{
    __teamplay_assert(3 < 1);
}
```

```
module Examples.LT

import Darknet

mutual
  lt_3_1 : CLang
  lt_3_1 = Assert lt_3_1_assert
           $ Halt

  lt_3_1_assert : Env -> Assertion
  lt_3_1_assert env =
    let
      x = Lit 3
      y = Lit 1
      x' = eval env x
      y' = eval env y
      prf = isLTE (S x') y'
    in
      MkAssertion (LT x y (MkEvald x x') (MkEvald y y') prf)
```

The corresponding IDRIS model. Note that the proof, `prf`, is derived from the successor of the left-hand side.

```
([],
 [MkAssertion (LT (Lit 3)
                  (Lit 1)
                  (MkEvald (Lit 3) 3)
                  (MkEvald (Lit 1) 1)
                  (No (\x1 => succNotLTEzero (fromLteSucc x1))))]) :
(List (String, Nat), List Assertion)
```

Since 4 cannot be LTE 1, 3 cannot be LT 1.

[to return to LT, GTE, and GT: Section 6.1.4]

Listing A.13: A C program which requires $3 < 3$

```
void main(void)
{
    __teamplay_assert(3 > 3);
}
```

```
module Examples.LT

import Darknet

mutual
  lt_3_3 : CLang
  lt_3_3 = Assert lt_3_3_assert
             $ Halt

  lt_3_3_assert : Env -> Assertion
  lt_3_3_assert env =
    let
      x = Lit 3
      y = Lit 3
      x' = eval env x
      y' = eval env y
      prf = isLTE (S x') y'
    in
      MkAssertion (LT x y (MkEvald x x') (MkEvald y y') prf)
```

The corresponding IDRIS model. Note that the proof, `prf`, is derived from the successor of the left-hand side.

```
([],
 [MkAssertion (LT (Lit 3)
                  (Lit 3)
                  (MkEvald (Lit 3) 3)
                  (MkEvald (Lit 3) 3)
                  (No (\x1 => succNotLTEzero
                                 (fromLteSucc (fromLteSucc
                                     (fromLteSucc x1)))))))]) :
(List (String, Nat), List Assertion)
```

Since 4 cannot be LTE to 3, 3 cannot be LT 3.

[to return to LT, GTE, and GT: Section 6.1.4]

A.5 GTE operator

Listing A.14: A C program which requires $0 \geq 0$

```
void main(void)
{
    __teamplay_assert(0 >= 0);
}
```

```
module Examples.GTE

import Darknet

mutual
  gte_0_0 : CLang
  gte_0_0 = Assert gte_0_0_assert
          $ Halt

  gte_0_0_assert : Env -> Assertion
  gte_0_0_assert env =
    let
      x = Lit 0
      y = Lit 0
      x' = eval env x
      y' = eval env y
      prf = isLTE y' x'
    in
      MkAssertion (GTE x y (MkEvald x x') (MkEvald y y') prf)
```

The corresponding IDRIS model. Note that the proof, `prf`, has the arguments swapped compared to the order they appeared in.

```
([],
 [MkAssertion (GTE (Lit 0)
                   (Lit 0)
                   (MkEvald (Lit 0) 0)
                   (MkEvald (Lit 0) 0)
                   (Yes LTEZero))]) :
(List (String, Nat), List Assertion)
```

Since 0 is LTE to 0, 0 is GTE to 0.

[to return to LT, GTE, and GT: Section 6.1.4]

Listing A.15: A C program which requires $0 \geq 1$

```c
void main(void)
{
    __teamplay_assert(0 >= 1);
}
```

```idris
module Examples.GTE

import Darknet

mutual
  gte_0_1 : CLang
  gte_0_1 = Assert gte_0_1_assert
          $ Halt

  gte_0_1_assert : Env -> Assertion
  gte_0_1_assert env =
    let
      x = Lit 0
      y = Lit 1
      x' = eval env x
      y' = eval env y
      prf = isLTE y' x'
    in
      MkAssertion (GTE x y (MkEvald x x') (MkEvald y y') prf)
```

The corresponding IDRIS model. Note that the proof, `prf`, has the arguments swapped compared to the order they appeared in.

```idris
([],
 [MkAssertion (GTE (Lit 0)
                   (Lit 1)
                   (MkEvald (Lit 0) 0)
                   (MkEvald (Lit 1) 1)
                   (No succNotLTEzero))]) :
(List (String, Nat), List Assertion)
```

Since 1 cannot be LTE to 0, 0 cannot be GTE to 1.

[to return to LT, GTE, and GT: Section 6.1.4]

Listing A.16: A C program which requires $1 \geq 0$

```c
void main(void)
{
    __teamplay_assert(1 >= 0);
}
```

```idris
module Examples.GTE

import Darknet

mutual
  gte_1_0 : CLang
  gte_1_0 = Assert gte_1_0_assert
          $ Halt

  gte_1_0_assert : Env -> Assertion
  gte_1_0_assert env =
    let
      x = Lit 1
      y = Lit 0
      x' = eval env x
      y' = eval env y
      prf = isLTE y' x'
    in
      MkAssertion (GTE x y (MkEvald x x') (MkEvald y y') prf)
```

The corresponding IDRIS model. Note that the proof, `prf`, has the arguments swapped compared to the order they appeared in.

```
([],
 [MkAssertion (GTE (Lit 1)
                   (Lit 0)
                   (MkEvald (Lit 1) 1)
                   (MkEvald (Lit 0) 0)
                   (Yes LTEZero))]) :
(List (String, Nat), List Assertion)
```

Since 0 is LTE to 1, 1 is GTE to 0.

[to return to LT, GTE, and GT: Section 6.1.4]

Listing A.17: A C program which requires $3 \geq 1$

```
void main(void)
{
    __teamplay_assert(3 >= 1);
}
```

```
module Examples.GTE

import Darknet

mutual
  gte_3_1 : CLang
  gte_3_1 = Assert gte_3_1_assert
          $ Halt

  gte_3_1_assert : Env -> Assertion
  gte_3_1_assert env =
    let
      x = Lit 3
      y = Lit 1
      x' = eval env x
      y' = eval env y
      prf = isLTE y' x'
    in
      MkAssertion (GTE x y (MkEvald x x') (MkEvald y y') prf)
```

The corresponding Idris model. Note that the proof, `prf`, has the arguments swapped compared to the order they appeared in.

```
([],
 [MkAssertion (GTE (Lit 3)
                   (Lit 1)
                   (MkEvald (Lit 3) 3)
                   (MkEvald (Lit 1) 1)
                   (Yes (LTESucc LTEZero)))]) :
(List (String, Nat), List Assertion)
```

Since 1 is LTE to 3, 3 is GTE to 1.

[to return to LT, GTE, and GT: Section 6.1.4]

Listing A.18: A C program which requires $3 \geq 1$

```c
void main(void)
{
    __teamplay_assert(1 >= 3);
}
```

```idris
module Examples.GTE

import Darknet

mutual
  gte_1_3 : CLang
  gte_1_3 = Assert gte_1_3_assert
          $ Halt

  gte_1_3_assert : Env -> Assertion
  gte_1_3_assert env =
    let
      x = Lit 1
      y = Lit 3
      x' = eval env x
      y' = eval env y
      prf = isLTE y' x'
    in
      MkAssertion (GTE x y (MkEvald x x') (MkEvald y y') prf)
```

The corresponding IDRIS model. Note that the proof, `prf`, has the arguments swapped compared to the order they appeared in.

```
([],
 [MkAssertion (GTE (Lit 1)
                   (Lit 3)
                   (MkEvald (Lit 1) 1)
                   (MkEvald (Lit 3) 3)
                   (No (\x1 => succNotLTEzero
                                (fromLteSucc x1))))]) :
(List (String, Nat), List Assertion)
```

Since 3 cannot be LTE to 1, 1 cannot be GTE to 3.

[to return to LT, GTE, and GT: Section 6.1.4]

Listing A.19: A C program which requires $3 \geq 3$

```c
void main(void)
{
    __teamplay_assert(3 >= 3);
}
```

```idris
module Examples.GTE

import Darknet

mutual
  gte_3_3 : CLang
  gte_3_3 = Assert gte_3_3_assert
          $ Halt

  gte_3_3_assert : Env -> Assertion
  gte_3_3_assert env =
    let
      x = Lit 3
      y = Lit 3
      x' = eval env x
      y' = eval env y
      prf = isLTE y' x'
    in
      MkAssertion (GTE x y (MkEvald x x') (MkEvald y y') prf)
```

The corresponding IDRIS model. Note that the proof, `prf`, has the arguments swapped compared to the order they appeared in.

```
([],
 [MkAssertion (GTE (Lit 3)
                   (Lit 3)
                   (MkEvald (Lit 3) 3)
                   (MkEvald (Lit 3) 3)
                   (Yes (LTESucc (LTESucc
                             (LTESucc LTEZero)))))]) :
(List (String, Nat), List Assertion)
```

Since 3 is LTE to 3, 3 is GTE to 3.

[to return to LT, GTE, and GT: Section 6.1.4]

A.6 GT operator

Listing A.20: A C program which requires $0 > 0$

```c
void main(void)
{
    __teamplay_assert(0 > 0);
}
```

```idris
module Examples.GT

import Darknet

mutual
  gt_0_0 : CLang
  gt_0_0 = Assert gt_0_0_assert
           $ Halt

  gt_0_0_assert : Env -> Assertion
  gt_0_0_assert env =
    let
      x = Lit 0
      y = Lit 0
      x' = eval env x
      y' = eval env y
      prf = isLTE (S y') x'
    in
      MkAssertion (GT x y (MkEvald x x') (MkEvald y y') prf)
```

The corresponding Idris model. Note that the proof, `prf`, is derived from the successor of one of the arguments, and that it has the arguments swapped compared to the order they appeared in.

```
([],
 [MkAssertion (GT (Lit 0)
                  (Lit 0)
                  (MkEvald (Lit 0) 0)
                  (MkEvald (Lit 0) 0)
                  (No succNotLTEzero))]) :
(List (String, Nat), List Assertion)
```

Since 1 cannot be LTE to 0, 0 cannot be GT 0.

[to return to LT, GTE, and GT: Section 6.1.4]

Listing A.21: A C program which requires $1 > 0$

```
void main(void)
{
    __teamplay_assert(1 > 0);
}
```

```
module Examples.GT

import Darknet

mutual
  gt_1_0 : CLang
  gt_1_0 = Assert gt_1_0_assert
           $ Halt

  gt_1_0_assert : Env -> Assertion
  gt_1_0_assert env =
    let
      x = Lit 1
      y = Lit 0
      x' = eval env x
      y' = eval env y
      prf = isLTE (S y') x'
    in
      MkAssertion (GT x y (MkEvald x x') (MkEvald y y') prf)
```

The corresponding Idris model. Note that the proof, `prf`, is derived from the successor of one of the arguments, and that it has the arguments swapped compared to the order they appeared in.

```
([],
 [MkAssertion (GT (Lit 1)
                  (Lit 0)
                  (MkEvald (Lit 1) 1)
                  (MkEvald (Lit 0) 0)
                  (Yes (LTESucc LTEZero)))]) :
(List (String, Nat), List Assertion)
```

Since 1 is LTE to 1, 1 is GT 0.

[to return to LT, GTE, and GT: Section 6.1.4]

Listing A.22: A C program which requires $0 > 1$

```c
void main(void)
{
    __teamplay_assert(0 > 1);
}
```

```idris
module Examples.GT

import Darknet

mutual
  gt_0_1 : CLang
  gt_0_1 = Assert gt_0_1_assert
           $ Halt

  gt_0_1_assert : Env -> Assertion
  gt_0_1_assert env =
    let
      x = Lit 0
      y = Lit 1
      x' = eval env x
      y' = eval env y
      prf = isLTE (S y') x'
    in
      MkAssertion (GT x y (MkEvald x x') (MkEvald y y') prf)
```

The corresponding IDRIS model. Note that the proof, `prf`, is derived from the successor of one of the arguments, and that it has the arguments swapped compared to the order they appeared in.

```
([],
 [MkAssertion (GT (Lit 0)
                  (Lit 1)
                  (MkEvald (Lit 0) 0)
                  (MkEvald (Lit 1) 1)
                  (No succNotLTEzero))]) :
(List (String, Nat), List Assertion)
```

Since 2 cannot be LTE to 0, 0 cannot be GT 1.

[to return to LT, GTE, and GT: Section 6.1.4]

Listing A.23: A C program which requires $3 > 1$

```c
void main(void)
{
    __teamplay_assert(3 > 1);
}
```

```idris
module Examples.GT

import Darknet

mutual
  gt_3_1 : CLang
  gt_3_1 = Assert gt_3_1_assert
           $ Halt

  gt_3_1_assert : Env -> Assertion
  gt_3_1_assert env =
    let
      x = Lit 3
      y = Lit 1
      x' = eval env x
      y' = eval env y
      prf = isLTE (S y') x'
    in
      MkAssertion (GT x y (MkEvald x x') (MkEvald y y') prf)
```

The corresponding IDRIS model. Note that the proof, `prf`, is derived from the successor of one of the arguments, and that it has the arguments swapped compared to the order they appeared in.

```
([],
 [MkAssertion (GT (Lit 3)
                  (Lit 1)
                  (MkEvald (Lit 3) 3)
                  (MkEvald (Lit 1) 1)
                  (Yes (LTESucc (LTESucc LTEZero))))]) :
(List (String, Nat), List Assertion)
```

Since 1 is LTE to 3, 3 is GT 1.

[to return to LT, GTE, and GT: Section 6.1.4]

Listing A.24: A C program which requires $1 > 3$

```
void main(void)
{
    __teamplay_assert(1 > 3);
}
```

```
module Examples.GT

import Darknet

mutual
  gt_1_3 : CLang
  gt_1_3 = Assert gt_1_3_assert
           $ Halt

  gt_1_3_assert : Env -> Assertion
  gt_1_3_assert env =
    let
      x = Lit 1
      y = Lit 3
      x' = eval env x
      y' = eval env y
      prf = isLTE (S y') x'
    in
      MkAssertion (GT x y (MkEvald x x') (MkEvald y y') prf)
```

The corresponding IDRIS model. Note that the proof, `prf`, is derived from the successor of one of the arguments, and that it has the arguments swapped compared to the order they appeared in.

```
([],
 [MkAssertion (GT (Lit 1)
                  (Lit 3)
                  (MkEvald (Lit 1) 1)
                  (MkEvald (Lit 3) 3)
                  (No (\x1 => succNotLTEzero (fromLteSucc x1))))]) :
(List (String, Nat), List Assertion)
```

Since 3 cannot be LTE to 2, 1 cannot be GT 3.

[to return to LT, GTE, and GT: Section 6.1.4]

Listing A.25: A C program which requires $3 > 3$

```c
void main(void)
{
    __teamplay_assert(3 > 3);
}
```

```idris
module Examples.GT

import Darknet

mutual
  gt_3_3 : CLang
  gt_3_3 = Assert gt_3_3_assert
           $ Halt

  gt_3_3_assert : Env -> Assertion
  gt_3_3_assert env =
    let
      x = Lit 3
      y = Lit 3
      x' = eval env x
      y' = eval env y
      prf = isLTE (S y') x'
    in
      MkAssertion (GT x y (MkEvald x x') (MkEvald y y') prf)
```

The corresponding IDRIS model. Note that the proof, **prf**, is derived from the successor of one of the arguments, and that it has the arguments swapped compared to the order they appeared in.

```
([],
 [MkAssertion (GT (Lit 3)
                  (Lit 3)
                  (MkEvald (Lit 3) 3)
                  (MkEvald (Lit 3) 3)
                  (No (\x1 => succNotLTEzero
                               (fromLteSucc (fromLteSucc
                                   (fromLteSucc x1))))))]) :
(List (String, Nat), List Assertion)
```

Since 4 cannot be LTE to 3, 3 cannot be GT 3.

[to return to LT, GTE, and GT: Section 6.1.4]

A.7 `Or` operator

Listing A.26: A C program where the right 'or' operand happens to be true

```
void main(void)
{
    __teamplay_assert((0 == 1) || (1 == 1));
}
```

The above C program has the same concept as Listing 6.18, but with the `True` and `False` expression swapped.

[to return to Or: Section 6.1.5]

A.8 Statements proof

```
([("wr_energy", 9), ("wr_time", 4), ("rd_time", 3)],
 [MkAssertion (And (GT (Var "wr_time")
                       (Var "rd_time")
                       (MkEvald (Var "wr_time") 4)
                       (MkEvald (Var "rd_time") 3)
                       (Yes (LTESucc (LTESucc (LTESucc
                            (LTESucc LTEZero))))))
                  (LTE (Var "wr_energy")
                        (Lit 10)
                        (MkEvald (Var "wr_energy") 9)
                        (MkEvald (Lit 10) 10)
                        (Yes (LTESucc (LTESucc (LTESucc
                             (LTESucc (LTESucc (LTESucc
                             (LTESucc (LTESucc (LTESucc
                                LTEZero))))))))))
                  (MkBEvald (GT (Var "wr_time")
                                (Var "rd_time")
                                (MkEvald (Var "wr_time") 4)
                                (MkEvald (Var "rd_time") 3)
                                (Yes (LTESucc (LTESucc
                                     (LTESucc (LTESucc
                                       LTEZero))))))
                            True)
                  (MkBEvald (LTE (Var "wr_energy")
                                 (Lit 10)
                                 (MkEvald (Var "wr_energy") 9)
                                 (MkEvald (Lit 10) 10)
                                 (Yes (LTESucc (LTESucc
                                       (LTESucc (LTESucc
                                       (LTESucc (LTESucc
                                       (LTESucc (LTESucc (LTESucc
                                           LTEZero))))))))))
                            True)
                  (Yes MkAnd))]) :
(List (String, Nat), List Assertion)
```

[to return to Statements: Section 6.2.3]

A.9 AES proof

```
([("addRoundKeyNr", 3), ("shiftRows", 7), ("subBytes", 1),
  ("addRoundKeyAcc", 19), ("mixColumnsAcc", 5),
  ("shiftRowsAcc", 3),
  ("subBytesAcc", 10),
  ("addRoundKey0", 2)],
 [MkAssertion (LTE (Plus (Plus (Plus (Plus
  (Plus (Plus (Plus (Var "addRoundKeyNr")
   (Var "shiftRows"))
   (Var "subBytes"))
   (Var "addRoundKeyAcc"))
   (Var "mixColumnsAcc"))
   (Var "shiftRowsAcc"))
   (Var "subBytesAcc"))
   (Var "addRoundKey0"))
   (Lit 50)
   (MkEvald (Plus (Plus (Plus (Plus (Plus (Plus
    (Plus (Var "addRoundKeyNr")
    (Var "shiftRows"))
    (Var "subBytes"))
    (Var "addRoundKeyAcc"))
    (Var "mixColumnsAcc"))
    (Var "shiftRowsAcc"))
    (Var "subBytesAcc"))
    (Var "addRoundKey0"))
    50)
    (MkEvald (Lit 50) 50)
    (Yes (LTESucc (LTESucc (LTESucc (LTESucc
         (LTESucc (LTESucc (LTESucc (LTESucc
         (LTESucc (LTESucc (LTESucc (LTESucc
         (LTESucc (LTESucc (LTESucc (LTESucc
         (LTESucc (LTESucc (LTESucc (LTESucc
         (LTESucc (LTESucc (LTESucc (LTESucc
         (LTESucc (LTESucc (LTESucc (LTESucc
         (LTESucc (LTESucc (LTESucc (LTESucc
         (LTESucc (LTESucc (LTESucc (LTESucc
         (LTESucc (LTESucc (LTESucc (LTESucc
         (LTESucc (LTESucc (LTESucc (LTESucc
         (LTESucc (LTESucc (LTESucc (LTESucc
         (LTESucc (LTESucc LTEZero))))))))))
         )))))))))))))))))))))))))))))))))))
         )))))))]) :
(List (String, Nat), List Assertion)
```

[to return to AES: Section 6.3.1]

Appendix B

User manual

- Install IDRIS with help from https://github.com/idris-lang/Idris-dev/wiki/Installation-Instructions
- `cd` into the '`Idris_Stuff`' directory
- start IDRIS with the example you would like to load by running the command
  `idris Path/To/Example.idr`
  (note that the `Examples/True` and `Examples/False` files may need to be loaded before any of the examples that use them)
- create a certificate by running the
  `mkCertificate <var-name>`
  command in IDRIS, e.g. if Examples/Loop/Loop.idr is loaded:
  `mkCertificate loop`
- either quit by typing '`:q`' (and repeat the IDRIS startup with a different module) or load another module by running the
  `:m Path.To.Module`
  command in IDRIS

B.1 Notes on code supplied

- The code supplied in the `Darknet.idr` file is written by the TEAMPLAY project apart from the bits mentioned in sections 5.2, 5.3, 5.4.
- The C code in the `Real_Life` Examples sub-folder is retrieved from GitHub, from [20, 12].
- The code in all the other Examples sub-folders is entirely my own work.

Appendix C

Ethics

UNIVERSITY OF ST ANDREWS
TEACHING AND RESEARCH ETHICS COMMITTEE (UTREC)
SCHOOL OF COMPUTER SCIENCE
PRELIMINARY ETHICS SELF-ASSESSMENT FORM

This Preliminary Ethics Self-Assessment Form is to be conducted by the researcher, and completed in conjunction with the Guidelines for Ethical Research Practice. All staff and students of the School of Computer Science must complete it prior to commencing research.

This Form will act as a formal record of your ethical considerations.
Tick one box

☐ **Staff Project**
☐ **Postgraduate Project**
☒ **Undergraduate Project**

Title of project

Improving Trust and Security through First-Class Certificates on Probabilistic Software Behaviour

Name of researcher(s)

Thomas Ekström Hansen

Name of supervisor (for student research)

Prof. Kevin Hammond

OVERALL ASSESSMENT (to be signed after questions, overleaf, have been completed)

Self audit has been conducted **YES** ☒ **NO** ☐

There are no ethical issues raised by this project

Signature Student or Researcher

TEHans

Print Name

Thomas Ekström Hansen

Date

2018-11-16

Signature Lead Researcher or Supervisor

K Hammond

Print Name

Kevin Hammond

Date

2018-11-16

This form must be date stamped and held in the files of the Lead Researcher or Supervisor. If fieldwork is required, a copy must also be lodged with appropriate Risk Assessment forms.

The School Ethics Committee will be responsible for monitoring assessments.

Computer Science Preliminary Ethics Self-Assessment Form

Research with human subjects

Does your research involve human subjects or have potential adverse consequences for human welfare and wellbeing?

YES ☐ **NO** ☒

If YES, full ethics review required
For example:
Will you be surveying, observing or interviewing human subjects?
Will you be analysing secondary data that could significantly affect human subjects?
Does your research have the potential to have a significant negative effect on people in the study area?

Potential physical or psychological harm, discomfort or stress

Are there any foreseeable risks to the researcher, or to any participants in this research?

YES ☐ **NO** ☒

If YES, full ethics review required
For example:
Is there any potential that there could be physical harm for anyone involved in the research?
Is there any potential for psychological harm, discomfort or stress for anyone involved in the research?

Conflicts of interest

Do any conflicts of interest arise?

YES ☐ **NO** ☒

If YES, full ethics review required
For example:
Might research objectivity be compromised by sponsorship?
Might any issues of intellectual property or roles in research be raised?

Funding

Is your research funded externally?

YES ☐ **NO** ☒

If YES, does the funder appear on the 'currently automatically approved' list on the UTREC website?

YES ☐ **NO** ☐

If NO, you will need to submit a Funding Approval Application as per instructions on the UTREC website.

Research with animals

Does your research involve the use of living animals?

YES ☐ **NO** ☒

If YES, your proposal must be referred to the University's Animal Welfare and Ethics Committee (AWEC)

University Teaching and Research Ethics Committee (UTREC) pages
http://www.st-andrews.ac.uk/utrec/