Universiteit Leiden
The Netherlands

Opleiding Informatica

Formalised semantics

of Lustre

Lennard Schaap

Supervisors:
Henning Basold
Marcello Bonsangue

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

06/07/2022

Abstract

Formal verification of synchronous programs has received growing attention since the emergence of synchronous programming languages like Lustre. The goal of this research is to explore the formal verification of Lustre program with the proof assistant Agda, which is based on Martin Löfs Type theory. To achieve this, the syntax and semantics of Lustre are formalised in Agda to model a Lustre program. Subsequently, a Linear Temporal Logic proof system is implemented in Agda which can be used to write formal specifications about Lustre programs. The result of this research enables the writing of formal proofs concerning properties of Lustre nodes.

Contents

1 Introduction

Lustre is a *synchronous data flow language*, which is designed for programming *reactive systems* [HCRP91]. Reactive systems are computing systems which continuously interact with the physical environment [Ber89]. This is opposed to an *interactive system*, which can make the environment wait.

Real-time programming is essential in industrial systems, where real-time programs read data from the real world (through sensors) and construct outputs based on this data. The computation of those outputs, also called the *response time*, has to be within some fixed time constraint too keep up with incoming signals. Real-time constraints ask for the predictability of performance.

The *synchronous hypothesis* assumes that the program is able to react before any further event occurs. Reactive systems are mostly used for automatic process control, monitoring and signal processing. Most reactive-systems are highly *critical*, so the ability of a real-time program to give guarantees about its computation within a certain time frame (*dependability*) is crucial. Think for example of the consequences of a design error (bug) in an air traffic control system. *Safety* and thus *program correctness* is a major concern for most real-time programs. Which raises the need for *formal verification* of those programs [MBAK11]. *Formal methods* describe the specification and verification of systems using mathematical (logical) formalisms.

Lustre is designed as a synchronous data flow language to make computations with real-time *data flows*. It's designed to interact with the environment in parallel. To achieve parallelism, time constraints must be imperatively satisfied and verified as an important part of the program's correctness. Synchronous languages such as Lustre, Esterel [BCG88] or Signal [GBBG86] allow programmers to assume that the program reacts instantaneously to external events by giving 'ideal' real-time programming primitives. Because of this, correctness proofs can be written using existing verification methods.

In 1993 Lustre was used as the core language of the industrial environment SCADE, which is used today in industry[1].

In this thesis, I will present a method of formalising the semantics of Lustre in a type theory with the help of the proof assistant Agda [BDN09]. With this formalisation, I will present proofs about properties of Lustre programs.

This motivates my research question: *How can formalisation of the semantics of Lustre be achieved in Agda and how can Agda be used to prove properties of Lustre programs?*

[1] https://www.ansys.com/products/embedded-software/ansys-scade-suite

1.1 Thesis overview

In this thesis I will explain the process of program verification in section 2. I will discuss formal systems (section 2.2.1), formal verification and validation (section 2.3) and methods for proving program correctness (section 2.3.2). In section 3, I will discuss type systems and the correspondence between type theories and typed functional programming languages. I will discuss Agda (section 3.3), a programming language which is based on Martin Löf's intuitionistic type theory (section 3.2.1). In section 4, I will discuss the syntax and semantics of the Lustre language, and why the formal definitions of these can be used to write proofs about them in Agda. In section 5, I will discuss related efforts to formally verify Lustre programs. In section 6 I will explain how Agda programs can be written by construction in type theory. Finally, I will discuss my implementation of the verification of Lustre's syntax and semantics in section 7.

This bachelor thesis is the result of my bachelor project under the supervision of H. Basold from the Leiden Institute of Advanced Computer Science (LIACS).

2 Program verification

The process of program verification is to ensure the program is performing the task it was intended to execute. To express this task, the programmer specifies the task using some kind of specification language.

To quote Alan Turing: *"The programmer should make a number of definite assertions which can be checked individually and from which the correctness of the whole program easily follows."* [Tur49].

2.1 Why program verification?

The importance of software verification depends on the program but its importance becomes clear when *safety-critical* systems are considered. A safety-critical system is a system which malfunction results in harm to the system itself, its environment or people. For example, a system failure in a nuclear reactor could result in severe damage to the environment and people. Programs that are used in these kind of systems must be verified to make sure no software related failures can happen.

In recent years, synchronous languages have received growing attention since their emergence. Lustre, Esterel and Signal are now widely used to program real-time, safety-critical applications such as nuclear power plant management and Airbus flight control systems.

This interest has in turn inspired researchers to make significant contributions to the problem of verifying synchronous programs and several program specification based methods for proving program correctness have been designed. I will discuss some of these methods in section 5.

The use for program verification can be emphasised with examples of fatal safety-critical software flaws in (recent) history:

- The case of Therac-25 [LT93] between 1985 and 1987, where six accidents caused patients to receive massive overdoses of radiation due to a race condition.
- The Ariane 5 [Maz96] rocket in 1996, which was destroyed in less than a minute after launch due to a bug in the guidance program.
- The Mars Climate Orbiter [Orb99], where failure to use metric units in the coding caused the orbiter to lose trajectory.

These incidents highlight the need for formal verification in both safety-critical systems rather than relying on empirical analysis and software testing methods.

A recent example of formal verification being used in industry is the seL4 (secure embedded L4 microkernel) verification project [KAE+14]. In the project it's proven that the seL4 OS kernel implements its specifications correctly. Systems based on seL4 have been used in aviation [CGB+18] and telecommunication [2].

2.2 Definitions

2.2.1 Formal systems

A *formal system* consists of a formal language over a finite alphabet of symbols (strings) together with postulates (a *syntax* that states which strings are in the language) and inference rules that distinguish some of the strings in the language as theorems of that language [Poh89]. These strings are called *well-formed*.

Formal logic is a formal system which has two types of well-formed expressions: terms (atoms) and formulas. *Axioms* are special formulas that are taken to be theorems. The inference rules are a set of premises and a conclusion to those premises. With these, we can decide whether some arbitrary string (formula) is well-formed if there is a way to express the strings in terms of the axioms, formulas and inference rules. *Semantics* define the interpretation of these formulas [Sch96]. The syntax of a language is the set of rules that define which arrangement of symbols denote well-formed strings.

A *proof system* is often presented as a collection of inference rules in the form:

$$\frac{\Gamma}{\psi}$$

where Γ is a set of premises and ψ is the conclusion to that set of premises. In the case when $\Gamma = \emptyset$, ψ can be concluded without premise (axiom).

Natural deduction is a proof system originally described by Gentzen and Jaśkowski [HP14] which expresses natural reasoning. A *proof* of some *judgement* $\Gamma \vdash \phi$ holds if there is a *deduction* of $\Gamma \vdash \phi$.

[2] https://gdmissionsystems.com/products/cross-domain-solutions/hypervisor

A deduction is a finite *proof tree* in which nodes are labeled with formulas. The root of the tree is the *conclusion* and the leaves are the *assumptions*. A proof tree is built up using inference rules on the assumptions to reach a new formula. This process can be repeated by using inference rules on the new formulas to reach the conclusion.

Truth ($\models$) is a semantic notion of a formula. A formula is *true* if and only if every interpretation that satisfies the assumptions also satisfies the conclusion.

A proof system is *sound* if everything that is provable is true, which means $A \models B$ can b concluded from $A \vdash B$. A proof system is *complete* if everything that is true has a proof, which means $A \vdash B$ can be concluded from $A \models B$.

A *completeness theorem* states for a proof system P and set of formulas S, that a formula is provable in P if and only if it is true in P.

Formal language syntax is often described by a *context-free grammar* (CFG) formalism. In CFG's, well-formed expressions can be formed from *terminals* (symbols in the language), *variables* (a set of other symbols, which each represent a language) and a *start symbol* (the starting variable).

Production rules (inference rules) are in the form:

$$\textit{Variable} \rightarrow \textit{String of variables and terminals}$$

Strings that can be derived from this CFG are considered well-formed.

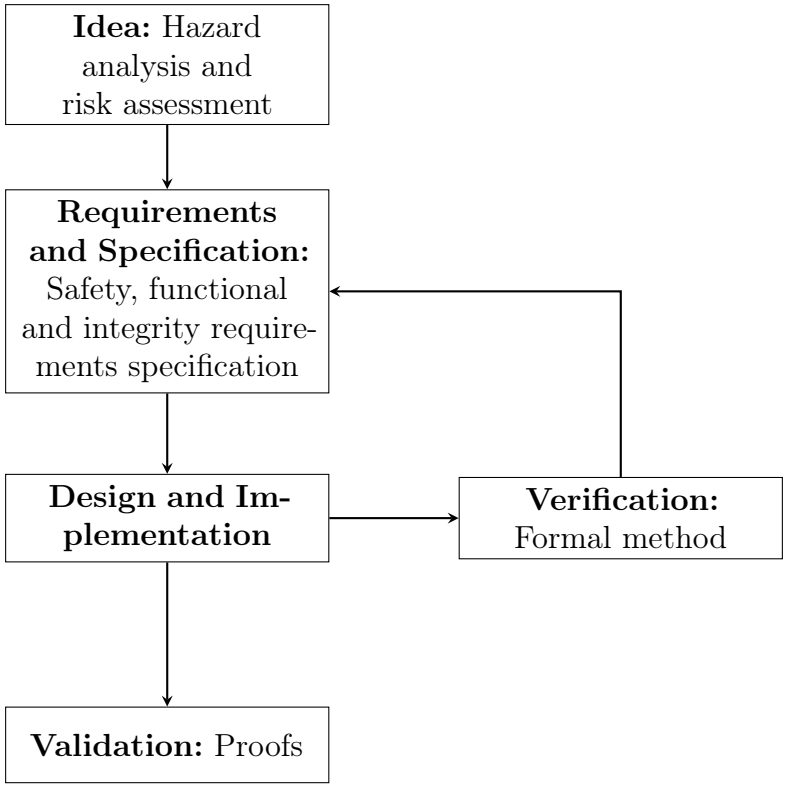
Programming language syntax is often written in Backus-Naur Form (BNF), in which variables are written between angle brackets ($\langle\rangle$). $\rightarrow$ is written as ::= and | is used for 'or', which is an abstraction for a list of productions with the same left side.

A formula Φ is *satisfiable* (SAT) if there exists an interpretation that evaluates the formula to true. Φ is considered *valid* if all interpretations of Φ evaluate to true. Φ is *decidable* if its validity can effectively be determined. A formal system is called *complete* if every valid formula can be derived from that system.

A *system* in this context refers to some abstract machine, for example a program (algorithm) or electronic circuit that transforms input signals into output signals. A system is *causal* if the output at time t only depends of the input at t.
A causal system is *memoryless* if the output at time t only depends on the input at time t, otherwise, the system must have memory. The configuration of a system for a given instant of time is defined as its *state*.

2.3 Formal verification and validation

Every software engineering method is based on a recommended development process which often has the following stages: Analysis, specification, design, coding, unit testing, integration and maintenance [KS98]. To visualise where formal methods fit into software development of a safety-critical system, imagine a process like this:



A formal method in this case refers to a mathematical approach to describe the properties the system must conform to. In order to formally verify the system, the requirements, specification and system design must also be written in a formal way. This means that program is described in a fully formal specification language with precise semantics.

Formal methods can eliminate ambiguity in software specifications by translating informal specifications about properties of the program into a formal specification language. *Verification* of a program involves mathematically asserting the program meets the formally stated requirements and specification by using a formal method to describe the program. The *validation* of the program means to *prove* that the program is functioning the way it was intended.

If the design and implementation of the program is written in a formal way, proofs that confirm certain properties of the program can be written.

2.3.1 Program abstraction

Most program verification methods use some kind of *abstraction* to prove certain properties of the program. This means approximating the program by forgetting certain information and turning it into a model for which proofs can be more easily constructed. This abstraction must of course conserve the class of properties we want to write proofs about. When programs are abstracted, we can conclude that if some behaviour does not occur in the abstraction, it will certainly not exist in the real program.

2.3.2 Program correctness

For interactive systems, if the input specifications to run program P (*precondition*) hold and the program *terminates* for those inputs and the desired properties are satisfied (*post condition*), P is said to be *partially correct*.

There exists a distinction between partial correctness and total correctness for interactive systems. An interactive program is called partially correct if it returns the right output *when* it terminates and *totally correct* if for every input, the halts and returns the right output. Because the stop-characteristic of a program is related to the halting problem, this is not decidable.

For real-time systems, program correctness depends both *logical* and *temporal* correctness [OVW98]. Logical correctness depends on the results of the computations. Temporal correctness depends on the *time at which the results are computed*. Temporal incorrectness often concerns computations that violate the specified time constraints (outputs are produced too early or their computation is overdue). A terminating real-time program would be *incorrect*.

2.4 Proving program correctness

The simplest way of proving the correctness of a program would be to extensively simulate all possible inputs and verify the outputs. However, non-exhaustive simulation (testing) can miss events and exhaustive simulation is too computationally expensive for systems with a large amount of reachable states.

Other methods for program verification include *theorem provers* [Bun70], *proof checkers* [BG01], *term rewriting systems* [Klo00] and *model checking* [CMCHG96].

Formal proofs are interesting because they can be generated by a computer or generated interactively with a human, which I will discuss in sections 3.3 and 7. A formal proof can also in turn be checked by the computer. When a program and its proof are constructed in parallel, deeper understanding of the program can be achieved.

To prove a program's correctness it must be described using a formal language (section 2.3). One can argue that any logical or functional program is its own specification and therefore correct by definition. However, the distinction between the specification and implementation of a program must be taken into account. Correctness depends on both correctness of the specification and the consistency of the implementation with its specification.

2.4.1 Formal specification languages

Formal Specification Languages, based on a formal mathematical logic, are generally designed to specify what needs to be computed (descriptive), but often not how the computation is actually done. Most of these systems are based on axiomatic set theory or higher-order logic.

There are (including others) multiple kinds of formal specification languages:

- *Model oriented* specification languages (VDM, Z, OCL, B) support the specification of systems by construction of a mathematical model of the system based on sets, relationships and predicates.
- *Algebraic* specification languages (OBJ) are descriptive. Signatures of operations define the syntax, and axioms define semantics.
- *Constructive* specification languages, usually type theories (Agda [BDN09], Coq [C.12]) concern themselves with (recursive) functions which are effectively computable.

The benefits of writing a formal specification is that the specification is unambiguous and verifiable. Properties of the program can be proven or tested automatically. Despite the obvious advantages, there are a lot of disadvantages such as cost and exploding complexity on larger systems.

Work on constructive logical systems such as type theories have become widespread since Martin Löf's intuitionistic type theory proved to have special relevancy to computing. I will discuss type theories in the next section.

3 Type systems

According to the Curry-Howard correspondence, there exists a correspondence between programs and mathematical proofs. This means, among other things, that a proof of functional correctness in a constructive logic corresponds to a certain program in lambda calculus. This lays a mathematical foundation for programs as proofs.

Essentially the correspondence can be explained by two different perspectives of typing judgments of the form $a : A$. Here a can be read as a term of type A, but also a can be read as a proof of the formula A [Geu08].

The correspondence has sparked the design of formal systems that act both as a proof system and a typed functional programming language. Examples include Per Martin Löf's intuitionistic type theory and Alonzo Church's typed λ-calculus. This typed lambda calculus has led to the development of interactive theorem proving software (Coq [C.12], Agda [BDN09], NuPrl [CAB+96], Alf [PN93] and Lego [Pol96]) in which programs can be formalised and proofs about them be checked.

3.1 Type theory

The main way types differ from sets is that sets are collections of objects which are in turn also sets, whereas types can be seen as collections of objects of the same nature [Geu08]. The original motivation for type theory was to provide a basis for constructive mathematics in Bertand Russels *Principia Mathematica* [Rus03]. It was introduced as a way to avoid Russel's paradox, which arises in set theory.

In a system of type theory, we say term α has a certain type A in some context Γ. We can write this as $\Gamma \vdash \alpha : A$.

Type theory describes how objects are *constructed* (recall section 2.4.1). Because it's a formal language, it describes what can and cannot be constructed (what types are well-formed and what types are not).

Many functional program languages use some kind of *type system* which can usually provide a *type checking* algorithm verifying the constraints of types which gives guarantees about properties of type safety properties for all possible inputs. For example, Haskell is a typed language, where functions are assigned with a type expressing what type the inputs must be and a type for the output.

C, C++ or Java are examples of typed languages. However, all of these language are not *total*, which means that a program p of type T will not always terminate. Moreover, these programs can raise exceptions when, for example, inputs of the wrong type are given. Languages based on type theory are total when a program p of type T will always terminate with an output of type T.

3.2 Dependent type theory

In *dependent type theory*, types can depend on *terms* or put differently, elements of other types [BD08]. For instance, a type A^n of vectors of length n can be defined, where n is parameterised by $\mathbb{N}$. We say that A^n is a *family of types* indexed by n.

The type [A] of lists of elements of type A is a *parameterised* type, and is not usually called dependent because it depends on a type rather than a term. However, in dependent type theories there exist a type of *small types* (universe), so that type [A] is considered a type of a list of elements of a given small type A.

A *context* is a finite ordered list of typing declarations which is defined (mutually) inductively [Geu08]:

- An empty list is a valid context.
- If Γ is a valid context, α is a term not already in Γ, and A is a valid type in Γ, then $\Gamma, x : A$ is also a valid context.

When considering dependent types, the type of functions and pairs must be generalized in dependent type theory. A function which type of output is dependent on its output is a *dependent function* and its type is the *dependent function type* (Π). For example, $\Gamma, \alpha : A \vdash B\ a$ is the type of the functions taking an argument α of type A, returning $B\ a$, a family of types indexed by elements in A. Or, if $\Gamma, a : A \vdash B\ \alpha$, then $\Gamma \vdash (\Pi\alpha : A)\ (B\ \alpha)$.

The *function type*, written as $A \to B$ is a special case of Π where the dependent type A does not depend on B.

The *dependent pair type* (Σ) can be considered as an ordered pair where the type of the second term is dependent on the first. Like the previous example, if $\Gamma, \alpha : A \vdash B\ \alpha$ is a type, then there exists a dependent pair type $\Gamma \vdash (\Sigma\alpha : A)\ (B\ \alpha)$.

The *product type*, written as $A \times B$ is a special case of Σ where the A does not depend on B.

Because of the Curry-Howard correspondence, a logical specification using dependent types can be expressed. There is a one-to-one correspondence between product types to conjunctions, and sum types to disjunctions. Howard and de Bruijn introduced dependent product types corresponding to universal quantification and dependent pair types corresponding to existential quantification [BD08].

There is an correspondence between propositions and types in a dependent type theory. There is also an correspondence between the proof of a proposition in constructive predicate logic and terms of the corresponding type. This also means that formulas and function can be expressed and proved by a verifying algorithm.

3.2.1 Martin Löf's intuitionistic type theory

The Curry-Howard interpretation of proposition as types was the basis for Per Martin Löf's intuitionistic type theory (MLTT) [ML98]. The theory was primarily intended as a foundation for constructive mathematics. Because of the proposition as types interpretation, it can also be used as a programming language [ML82].

For any context Γ, if A is a (non parametrised) type, $\Gamma \vdash A : Type$. If $\alpha : A$ is a typing declaration in Γ, then we write $\Gamma \vdash \alpha : A$ meaning α is a well-typed term of type A in context Γ.

$\Gamma \vdash x : A$ is a typing declaration of x in Γ. In this theory, there are *atomic* types called $\top$ (top, also called the trivial type) and $\bot$ (bot, or the empty type).

- $\bot$ has no constructors, which means there is no way to construct something of type $\bot$.
- $\top$ has a single constructor (tt), it is trivial to make something of type $\top$.

The semantics of dependent function types, dependent pair types, function types and product types are as the same as in section 3.2. We write $\Gamma \vdash A : True$ for the notion that A is a nonempty type. If $\Gamma \vdash \alpha : A$, then $\Gamma \vdash A : True$. This corresponds with $\Gamma \vdash A$ in first-order logic. The typing rules are used as the inference rules for *intuitionistic first-order logic*.

For example, modus ponens can be described like this:

$$\frac{\Gamma \vdash A \to B : True,\ \Delta \vdash A : True}{\Gamma, \Delta \vdash A : True}$$

3.3 Agda

Agda is a dependently typed programming language and proof assistant based on MLTT [BDN09]. In contrast to proof assistants such as Coq, Agda is primarily developed as a programming language rather than a proof assistant. Programming proofs in Agda consists of defining types and recursive functions. Unlike Coq, Agda has no proof automation to aid in the construction of proofs.

However, proofs in Agda are written in a functional programming style, which is easy to understand for people familiar with existing functional languages. Using the functional programming style, proofs can be constructed from *smaller* proofs (programs). This corresponds to writing mathematical proofs to prove mathematical theorems constructively and to run such proofs as algorithms.

Because the semantics of streams are based on coinductive types (section 6.3), which are easily expressible in Agda, I've chosen Agda to formalise the semantics of a Lustre program.

4 Lustre

LUSTRE is a *synchronous data flow language*, designed for programming reactive systems [HCRP91]. As discussed in the introduction of this thesis, reactive systems react to real-time external events. The synchronous hypothesis states that the program is able to react within a given time interval.

The time interval between 'events' has to be set to the time constraints introduced by the dynamics of the environment [3]. If it is possible to verify the synchronous hypothesis, the ideal programming primitives offered by Lustre can be seen as a sufficient abstraction.

The object code of most synchronous programs is structured as a finite automaton in which the transitions are labeled with a *linear* piece of code that corresponds to an elementary reaction of the program [HCRP91]. Because the code is linear and not cyclic, the execution time of a program can be calculated for given hardware. The synchronous hypothesis can be tested like this.

A way of giving semantics to synchronous programs is by viewing them as functions that transform a stream of inputs to a stream of outputs. A Lustre program is built up of a network of nodes which are analogous to these functions. Nodes operate on *flows* of data. A *flow* in Lustre refers to a pair of an infinite sequence of values of the same type (*stream*) and a *clock*, representing a sequence of times.

A Lustre program P is functional, and maps its input flows to output flows. If we assume the synchronous hypotheses, all functions must satisfy *causality* and *predictability of execution time*. Satisfying causality means at any time instance t, the output of P can only depend on the input before or at t. Satisfying the predictability of execution time means loops and recursive functions can't be implemented unless there is an upper bound on the execution time.

The functional behaviour of a lustre program P does not depend on the time instance t. This means the functional validation of P can be done independently from the machine the program is designed for.

[3] https://www-verimag.imag.fr/DIST-TOOLS/SYNCHRONE/lustre-v6/doc/lv6-ref-man.pdf

4.1 Example of a node in Lustre

Lustre equations are mainly structured using a network of *nodes*. A Lustre node is made of input and output declarations. The body of a node is made of a list of assertions and equations:

```
node example(x:bool; y:bool) returns (a:bool; b:bool);
let
    a = x fby y and not pre x;
    b = true fby not a;
tel
```

The full specification of the Lustre syntax can be found in the Lustre V6 Reference Manual [3]. I will lay out a reasonable subset of this syntax for the scope of this thesis.

This simplified syntax of the *interface* of a node (for the purposes of this thesis) is illustrated below in BNF.

```
<Node> ::= node <Identifier>(<Params>) returns (<Params>) ; <Body>
```

Parameters are a list of typed identifiers:

```
<Params> ::= [ <TypedIds> ]

<TypedIds> ::= <Identifier> : <Type>

<Type> ::= bool
         | int
         | real
```

Where the body is made up of an equation list:

```
<Body> ::= let [ <Equation> ] tel
```

Equations can define outputs and values of local variables:

```
<Equation> ::= <Left> = <Expression> ;
```

A subset of the syntax of expressions is as follows:

```
<Expression> ::= <Constant>
               | <Variable>
               | not <Expression>
               | - <Expression>
               | pre <Expression>
               | current <Expression>
               | <Expression> when <ClockExpr>
               | <Expression> fby <Expression>
               | <Expression> -> <Expression>
```

```
              | <Expression> and <Expression>
              | <Expression> or <Expression>
              | <Expression> xor <Expression>
              | <Expression> + <Expression>
              | <Expression> - <Expression>
              | <Expression> * <Expression>
              | if <Expression> then <Expression> else <Expression>
```

A Clock expression refers to an identifier:

```
<ClockExpr> ::= <Identifier>
              | not <Identifier>
```

4.1.1 Types and operators

Predefined types that can be used are Booleans, integers and reals, expressed by `bool`, `int` and `real` respectively. A stream can only be of a certain type. In the example, x and y denote Boolean flows. Output a in the example is defined by the equation `a = x fby y and not pre x`.

The `and` and `not` operators are regular (Boolean) operators. In the synchronous data flow model, these operators work point-wise on flows. For example, if we consider stream X = $(x_0, x_1, ..., x_n, ...)$ and stream Y = $(y_0, y_1, ..., y_n, ...)$, X `or` Y denotes the stream $(x_0 \vee y_0, x_1 \vee y_1, ..., x_n \vee y_n, ...)$. In this example, it's assumed that X and Y are on the same clock, which will be discussed in section 4.1.2.

Due to the synchronous nature of flows, temporal operators can be used. Lustre has four temporal operators: `fby` (followed by), `pre` (previous), `when` and `current`:

- The `fby` operator is a binary operator which defines a flow that is equal to the left expression at time 0 of the clock, and equal to the right expression at time > 0. For example, if we consider flow X = $(x_0, x_1, ..., x_n)$ and flow Y = $(y_0, y_1, ..., y_n)$, X `fby` Y is the flow $(x_0, y_1, ..., y_n)$.
- The `pre` operator is a unary operator which defines a flow that is at every time step equal to the value of the flow at the previous time step. For example, if we consider the flow X = $(x_0, x_1, ..., x_n, ...)$, `pre` X is the flow $(nil, x_0, ..., x_n, ...)$, were *nil* means the flow is undefined at that time instance.
- The `when` operator is a binary operator which defines a flow that is only defined when the right (Boolean) expression is *true*.
- The `current` operator is a unary operator which defines a flow that is equal to the last defined value (at time step t or previous) of the right expression.

The `when` and `current` operators will be more thoroughly explained in the next section.

4.1.2 Sequence operators and clocks

The `when` and `current` operators are called *sequence* operators, which can manipulate sequences, their behaviour, given two arbitrary streams X and Y, is shown in table 1.

Global Clock	0	1	2	3	4	5	6	7	...
X	T	T	T	F	F	T	F	F	...
Y	F	F	T	T	F	F	T	T	...
X when Y			T	F			F	F	...
current (X when Y)	nil	nil	T	F	F	F	F	F	...

Table 1: `when` and `current`

Note that the gaps between the values of X `when` Y are not *nil*. The flow X `when` Y denotes the stream `(T, F, F, F, ...)`. This means that this stream is on a different clock, in this case clock Y. In other words, X `when` Y does not have *the same notion of time* as X and Y.

As stated before, a Lustre flow consists of a stream and its corresponding clock. Arbab and Rutten present flows as *timed data streams* defined by a pair of a data stream and a time stream of type $\mathbb{R}_+$. Lustre's clocks use natural numbers to represent discrete time intervals [HCRP91] but replacing the positive real numbers with natural numbers keeps the model largely the same [AR02]. Different flows can therefore have different clocks, meaning that if we consider two flows which streams are equivalent, they do not have to be equal.

Streams that follow the global clock are defined at each time step t. A clock can be considered to be `true` at t if it is defined at that instant. The global clock can be viewed as a Boolean stream which is defined (true) at every t. Constant flows follow the global clock. For example, the constant flow `true` denotes an infinite sequence of Booleans (*true*, *true*, ...). This global clock is not necessarily bound to a physical time interval. The basic clock tick should be considered as the minimal amount of time in which a program cannot discriminate external events in order to satisfy causality.

A clock can also be defined by a non-constant flow, which in turn has its own clock. This means clocks can be nested. For example, we can model a millisecond as a clock based on the global clock which *true* value corresponds with a millisecond signal from a real clock.

Lustre's *clock calculus* consists of associating a clock with each stream of the program and checking whether an operator applies to correctly clocked streams. When arithmetic is applied to flows with different clocks, we have to consider that either causality or the bounded memory condition [Cas92] may be violated.

4.1.3 The data flow model

The Lustre data flow model is based on a block diagram description. Block diagrams are often used for specifying and programming real-time systems [Cas94]. A system is made up of a network of operators acting in parallel and in time with their rate of input. A graphical representation of the block diagram for the system of equations `a = x fby y and not pre x` and `b = true fby not a` from the example in section 4.1 is illustrated below:

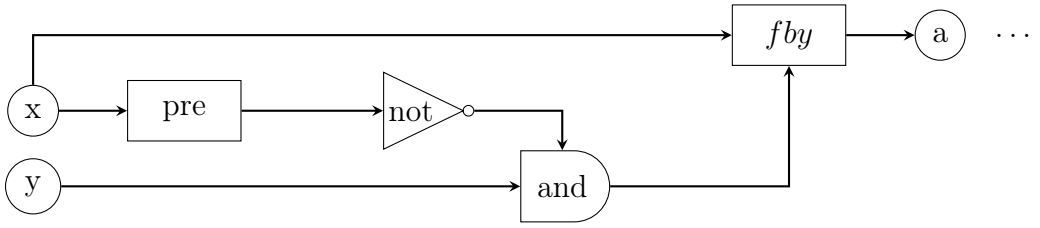


Figure 1: Schematic of the equation a = x fby y and not pre x

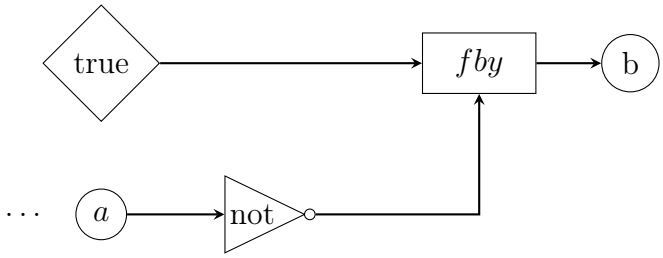


Figure 2: Schematic of the equation b = true fby not a

Consider also the following block diagram containing a feedback loop which represents a node producing the fibonacci sequence[4]:

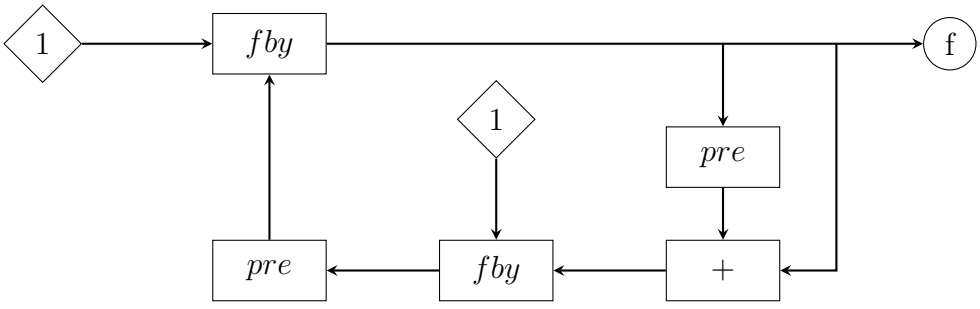


Figure 3: Schematic of the equation f = 1 fby (pre (1 fby (f + pre f)))

The order of the arguments of the `fby` operator is not specified by the diagram, but can be inferred from the equation. The block diagrams show that outputs can be used as inputs for other equations or assertions. Furthermore, constant flows such as *true* can be used with the *fby* operator. Because data flow model is a functional model, it is suited for formal verification.

[4]https://homepage.divms.uiowa.edu/ tinelli/classes/181/Spring08/Lustre-tutorial/Luke/Fibonacci.html

4.1.4 Synchronous data flow

The combination of the synchronous- and data flow model can be expressed by adding a time dimension to the data flow model. This way, a flow can be described by a stream of values associated with a global clock. A flow takes the i^{th} value at time i of the clock. If we take arbitrary streams x and y, the outputs a and b will be as follows:

Global Clock	0	1	2	3	4	5	6	7	...
x	T	F	T	F	T	F	T	F	...
y	T	T	T	T	F	F	F	F	...
pre x	nil	T	F	T	F	T	F	T	...
not pre x	nil	F	T	F	T	F	T	F	...
y and not pre x	nil	F	T	F	F	F	F	F	...
a (x fby y and not pre x)	T	F	T	F	F	F	F	F	...
not a	F	T	F	T	T	T	T	T	...
b (true fby not a)	T	T	F	T	T	T	T	T	...

4.1.5 Multiple equations

`Example` has two outputs. Lustre allows multiple definitions to be written. To call the node, we can write `(a, b) = example(x, y);` which defines a and b to be the first and second result of the call respectively. A model of the behaviour of this node is illustrated below. Eqx stands for the equational definition of its corresponding output variable:

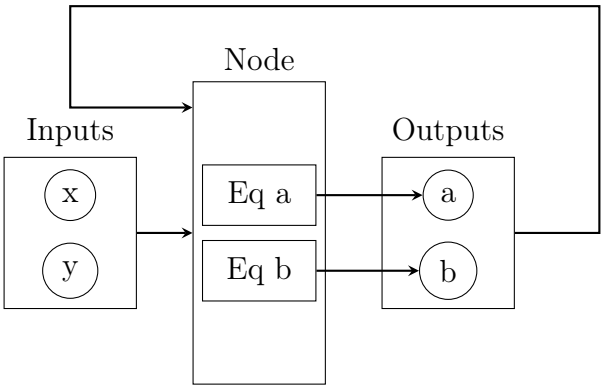


Figure 4: Model of the behaviour of `example`

As described before, an equation can use both input and output flows. For every instance of the clock, the value of the output flows associated with that clock cycle are computed. Equations are considered in order.

If a flow is not defined at a certain time, its value is considered `nil` (undefined). If the value of an output flow at a certain cycle is `nil`, the program is not well-formed and should be rejected. In the example, the outputs a and b are defined at every clock tick and should therefore be accepted.

In general, for an arbitrary number of inputs, outputs and equations, the behaviour of a node can be modelled like this:

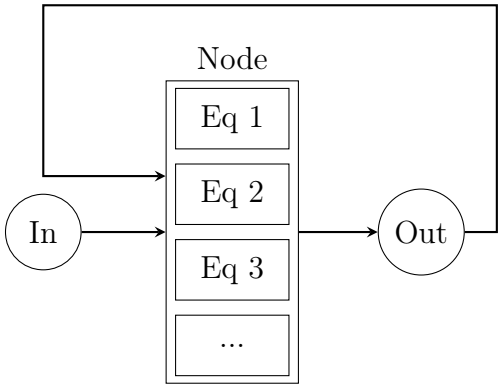


Figure 5: Model of the behaviour of a node

4.2 A system of nodes

A Lustre program typically consists of a network of nodes. A node can be used as a function by another node in an equation. For example, consider the two nodes `system` and `plus`:

```
node system(x:int; y:int) returns (z:int);
let
    z = x fby plus(x, y);
tel

node plus(a:int; b:int) returns (c:int);
let
    c = a + b;
tel
```

The behaviour of a system of nodes can be modelled like this:

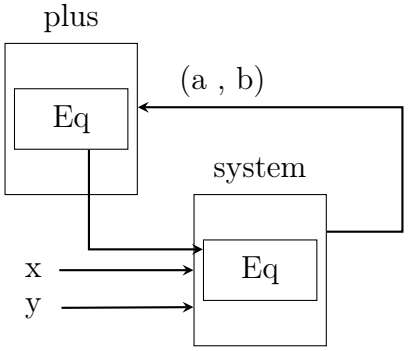


Figure 6: Model of the behaviour of a system of nodes

In the data flow model, the block diagram of these nodes is shown in the figure below:

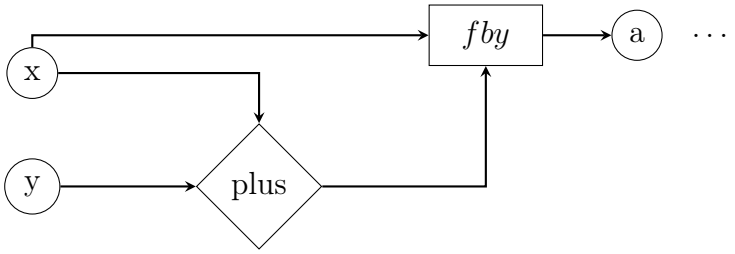


Figure 7: Schematic of `a = x fby plus(x, y)`

4.3 Verification of Lustre programs

As noted in the introduction, Lustre programs are used in reactive systems, which often are safety-critical. This means that program verification is crucial. The safety of a critical application often depends on a set of properties that the program must fulfill.

We want to make sure a written Lustre program is both syntactically well-formed and meets its formal specification. We do not have to prove the total correctness of the program. This is automatically ensured in the form of productivity for streams. At each time instant t, the computation of the value of the stream at time t (σ_t) will terminate due to the synchronous hypothesis.

The goal is thus to check whether the program satisfies the expected properties. Since we consider dynamic systems, we talk about the relations between the input and output sequences and are interested in the classification of certain *temporal properties*. There is a lot of theory about the classification of temporal properties [MP90]. I will discuss temporal logic in the next section.

4.4 Temporal logic

To reason about the behaviour of Lustre programs, a logic is needed that allows reasoning about temporal properties. *Temporal logic* is used for reasoning about properties of languages *over time*. *Linear temporal logic* is such a temporal logic that allows such reasoning in terms of *paths* which are similar to Lustre's flows.

The syntax of LTL is as follows:

$$\phi ::= true \mid a \mid \phi \ \wedge \ \psi \mid \neg \ \phi \mid \ \circ \phi \mid \phi \ \cup \ \psi$$

Where a is an atomic proposition, meaning the smallest possible formula in propositional logic which either evaluates to true or false. $\circ$ the 'next' operator (ϕ is true at the next time instance) and $\cup$ the 'until' operator (ϕ is true at least until ψ becomes true).

There are two key operators in temporal logic:

- *Eventually* ($\diamond\phi$) : ϕ will become true at some point in the future
- *Always* ($\Box\phi$) : ϕ will always be true

In Lustre, flows are defined as a pair of stream and a clock. Because flows can be associated with each other by their clocks and flows are well-defined at each time instant, it's useful to be able to reason about linear time properties. LTL has specific operators to specify linear time properties, and can be used to reason about them. Each instance of time has a well-defined successor and no branches are possible. This makes LTL a suitable logic to reason about temporal properties of Lustre programs.

Consider the execution of a Lustre program P at global clock time t. For the verification of this program, the difference between *safety properties* and *liveness properties* has to be considered.

- A safety property states that nothing (bad) will ever happen.
- A liveness property states that something (good) will eventually happen.

A safety property can be verified in temporal logic by proving $\Box\phi$ holds and a liveness property can be verified by proving $\diamond\phi$ holds.

4.4.1 Static verification

The first important issue of verifying Lustre programs is checking their well-formedness. Notable things to check are:

- Any output variable must belong to one and only one equational definition.
- No recursive node calls can be made (Lustre allows only static networks to be described).
- No *Clock inconsistencies*.
- No outputs yielding *nil*.
- No *cyclic definitions*.

4.4.2 Recursive node calls

A recursive node call is a node calling itself in a system of nodes (section 4.2). Recall that recursive node calls have to be avoided to let the synchronous hypothesis hold (section 4).

4.4.3 Undefined output

Lustre does not allow outputs to be undefined at any point in time, so any program that can output a flow defined as *nil* at any point in time, is not well-formed [HCRP91].

4.4.4 Clock consistency

To explain what clock consistency means for a Lustre program, consider the following node:

```
node inconsistent(a:int) returns (x:bool; y:int);
let
x = false -> not pre x;
y = a + (a when x);
tel
```

In the second equation, the computation of y needs both the n^{th} and $2n^{th}$ values of a. Adding something at time n with something at time $2n$ runs into an inconsistency. In other words, any binary operator has to apply to operands sharing the same clock. This means the clock calculus of each equation has to be checked statically.

4.4.5 Clock calculus

To check whether two flows operate on the same clock means to check the equality of the Boolean streams. However, static equality checking of Boolean streams is undecidable, so other rules have to be applied to compare the clocks of two distinct flows. These rules are formally described in [CPHP87]. Two Boolean streams define the same clock if and only if these can be unified by means of syntactical substitutions [HCRP91] Consider the following example:

```
x = a + b;
y = a when (y > b);
u = a when (a + b > b);
v = a when (b < y);
```

Here, `y` and `u` are on the same clock, while `v` is on a different clock.

4.4.6 Cyclic definitions

Any cycle in the system of equations (visualised in the block diagram description) should at least contain one `pre` operator. Consider the equation $x = (2 * x) + 2$. The computation of this equation will result in a flow of *nil* values, which will be rejected. I will further discuss how proofs of each of these properties can be written in the next section.

4.5 Dynamic verification

If the program is well-formed, we can check if the program meets its formal specification. This means we have to make sure that for every input, we get the output we desire.

Consider the following Lustre program which models a stopwatch. `time_unit` specifies the unit of time and is set to the global clock, `start_stop` indicates the activation of the start/stop button, `time` indicates the time displayed on the screen and `running` indicates whether the stopwatch is currently running. Let's say every clock cycle in Lustre is one unit of time for the stopwatch:

```
node stopwatch (time_unit: bool; reset: bool; start_stop: bool)
returns (time: int; running: int);
let
    time = 0 -> if reset then 0
           else if running and time_unit then pre(time) + 1
           else pre(time);
    running = false -> if start_stop then not pre(running)
                       else pre(running);
tel
```

One possible execution of the program is illustrated below:

Global Clock	0	1	2	3	4	5	6	7	8	9	...
time_unit	T	T	T	T	T	T	T	T	T	T	...
reset	F	F	F	F	T	F	F	F	F	F	...
start_stop	F	T	F	F	F	F	F	F	T	F	...
time	0	1	2	3	0	1	2	3	3	3	...
running	F	T	T	T	T	T	T	T	F	F	...

The behaviour of this program can be abstracted into LTL and verification of properties can be expressed in LTL (section 4.4). A specification (temporal property) to verify could for example be; 'Once stopped, `running` stays *false* until a restart'. This can be encoded into LTL, yielding the following specification: $(\circ$ `start_stop & running`$) \rightarrow \circ((\neg$`running`$) \cup \circ$ `start_stop`$)$.

'Once stopped' means that the stopwatch was running when it was stopped the next instant. A 'restart' happens only when the stopwatch was not running before and the stopwatch is started again in the next instant. Note that for an arbitrary input stream, it doesn't have to be the case that `start_stop` ever becomes true a second time. To avoid this, we can add an extra assumption or use the *weak until* operator ($\mathbf{W}$) for which $\psi \ \mathbf{W} \ \phi \equiv (\psi \ \cup \ \phi) \vee \Box \ \psi$ applies. Which means either $\psi \ \cup \ \phi$ eventually becomes true or ψ stays true forever.

To summarise, given an informal program P and a temporal property ϕ, the verification of ϕ can be done by defining a formal abstraction of P that conserves the property being verified (section 2.3.1), defining an abstraction of ϕ and using this abstraction to verify that the abstract program P' satisfies the abstract property ϕ'. From this it can be inferred that P must satisfy ϕ [DHJ+01].

In the following section, I will formalise the semantics of Lustre in Agda. Using this formal abstraction, temporal properties of Lustre programs can be verified with LTL.

5 Related work

In this section I will briefly review other relevant studies on program verification for Lustre programs. In "*A methodology for proving control systems with Lustre and PVS*", [BCPvD99] Bensalem et al. show how to use Lustre combined with the PVS proof system[5] to derive provably correct control programs.

In "*Synchronous program verification with Lustre/Lesar*" [Ray10], Raymond proposes general methods for formal verification of Lustre programs.

In "*Transformation von Scade-Modellen zur SMT-basierten Verifikation*" [Bas14] [BGHM14], Basold et al. procedure fully automatic verification of the security characteristics of *Scade* models [Ser11]. Scade is a graphical environment which semantics are based on Lustre. It is developed by the Esterel-Technologies company. To do this, Basold et al. model (abstract) the programs as quantifier-free first-order formulas. This model is transformed to an SMT instance and passed to a solver.

In "*Scaling up the formal verification of Lustre programs with SMT-based techniques*" [HT08], Hagen et al. present a general approach for verifying safety properties of Lustre programs automatically.

In "*Formal modelling and automatic verification of Lustre programs using NP-tools*" [Lju99], Ljung gives a procedure for translating Lustre programs to NP-Tools code (a propositional theorem solver).

In "*Towards Mutation Analysis for Lustre Programs*" [dD08], de Bousquet and Micheal Delauney use mutation analysis [DLS78] to produce a testing tool dedicated to synchronous programs which randomly and dynamically produces test sequences.

Lastly, in "*A formally verified compiler for Lustre*" [BBD+17], Bourke et al. describe the specification and verification of a compilation chain that treats the key aspects of Lustre: sampling, nodes and delays in the interactive theorem prover Coq. This is very similar to the approach I present in this paper, which I will discuss in the next sections. "*Towards a denotational semantics of streams for a verified Lustre compiler*" [BJP22] is a recent advancement of this appoach where Bourke et al. give a compilation correctness proof by modelling the input language with a relational-style semantics.

This collection of work shows that there are a lot of ways to tackle the verification of Lustre (or similar synchronous languages). The general approach is to model Lustre programs into simpler models for which proofs can be verified by theorem solvers.

[5] http://pvs.csl.sri.com/

6 Dependent type theory in Agda

In this section I will explain how Type theory is defined in Agda and how programs can be written by construction. In Agda, the data keyword creates a data type the constructors of the type are listed. $\bot$ (bot) can be defined as a data type without constructors. It is therefore empty and also sometimes called the *empty* type.

```
data ⊥ : Set where
```

Records are types for grouping values. They are a generalisation of the dependent product type Σ and provide named fields. $\top$ (top) is defined in Agda as a record type with a single constructor. $\top$ has no fields: it is trivial to make one, and contains no information.

```
record ⊤ : Set where
  constructor tt
```

The Boolean type has two constructors, it can either be true or false (one bit of information).

```
data Bool : Set where
  true : Bool
  false : Bool
```

Natural numbers are inductively defined as either zero, or the successor of another natural number.

```
data Nat : Set where
  zero : Nat
  suc : Nat → Nat
```

The product type is defined as a record with two fields, `fst` and `snd`:

```
record _×_ (A : Set) (B : Set) : Set where
  constructor _,_
  field
    fst : A
    snd : B
open _×_
```

Agda offers support for mixfix notation. Mixfix notation can be indicated by providing Agda with the places where the arguments must go by using underscores. The product type can be constructed with $A \times B$ where the type parameters are A and B.

The record constructor _, _ can be used to make something of type $A \times B$ using this notation. This constructor takes two parameters: the fields `fst` and `snd`, of type A and B, respectively. If we have x of type A and y of type B, then (x , y) is of type $A \times B$.

Opening _ × _ enables the two projection functions:

```
fst : A × B → A

snd : A × B → B
```

to be used by the programmer.

A simple proof of the commutativity of × can be written like this:

```
×-comm : A × B → B × A
×-comm (a , b) = b , a
```

The proof consists of constructing of something of type $B \times A$ which yields the type of the outcome, namely: b , a.

6.1 A programming example in Agda

Consider the definition of a list from the Agda standard library:

```
data List (A : Set) : Set where
  [] : List A
  _::_ : A → List A → List A
```

A list is another example of an inductive type. Every element in the list is of type A. A list can be constructed with [], which yields an empty list, or by a constructor which takes an element of A and a list of A, which produces a new list of A.

A specific list can then be constructed by declaring its type and definition. For example:

```
list : List Nat
list = zero :: (suc zero :: [])
```

Agda can help with program construction by validating just a part of a written program using so called *holes*. These holes can be filled with the correct definitions using *case-splitting* and *refining*.

When an Agda program is compiled, it checks the type definitions for any errors. Also, it checks whether the programmer has left any holes in his program, denoted by a question mark. To illustrate this, consider the following example of writing a program to append two lists together:

```
_++_ : List A → List A → List A
xs ++ ys = ?
```

When running this program, Agda will detect the hole and provides the type signature of the code that should be filled in:

```
?O : List A
```

Using Agda's case-split, we can pattern match on `xs`, which is either empty or a constructor of a list. After splitting the program looks like this:

```
_++_ : List A → List A → List A
[] ++ ys = { }
(x :: xs) ++ ys = { }
```

Now, there are two holes to fill. The first case, in which `xs` is the empty list, the resulting list is just `ys`. The second hole can be recursively defined. Here Agda's features can be used again to write only part of the program and leave the rest as a hole. The first item of the resulting list has to be the first item of `xs`, resulting in the following program:

```
_++_ : List A → List A → List A
[] ++ ys = ys
(x :: xs) ++ ys = x :: { }
```

Now the rest of the list can be defined in terms of recursively appending the tail of `xs` with `ys`, giving the final program:

```
_++_ : List A → List A → List A
[] ++ ys = ys
(x :: xs) ++ ys = x :: (xs ++ ys)
```

6.2 Vectors

Vec is an inductive dependent type parameterised by $(n : Nat)$. It has two constructors: [] constructs an empty vector and _::_, which takes an element of A and a Vec of A of length n and produces a new Vec of A of length $(suc\ n)$. Arguments between curly brackets are implicit, and don't have to be given and can be inferred with Agda.

```
data Vec (A : Set) : Nat → Set where
  [] : Vec A zero
  _::_ : {n : Nat} → A → Vec A n → Vec A (suc n)
```

For example, a vector of length n consisting of repeating copies some element of type A can be defined like this:

```
constr : (n : Nat) → (x : A) → Vec A n
constr zero x = []
constr (suc n) x = x :: constr n x
```

6.3 Streams, coinduction and bisimilarity

A stream can be defined with a record type with projections `hd` (head) of type A and `tl` (tail) of type Stream A. Intuitively this will generate an infinite structure of elements which can be explored with hd and tl. The coinductive keyword lets the programmer define a recursive record:

```
record Stream (A : Set) : Set where
  coinductive
  constructor _::_
  field
    hd : A
    tl : Stream A
open Stream
```

Structures like *List* and Vec are finite in the way that they are defined by their constructors. Formally they are called *initial algebras*. Streams on the other hand, are infinite structures (formally called *coalgebras*) with *destructor* operators `hd` and `tl` [JR99]. Note that this is reflected in the definition of a Stream in Agda, which takes no constructors.

Like in the previous example, a stream of repeating elements of some type A can be defined like this:

```
rep : (x : A) → Stream A
hd (rep x) = x
tl (rep x) = rep x
```

Two streams of the same type are the same if they have the same term when destructing the streams at the same point (pointwise equality). This is called *bisimilarity*. Bisimilarity on streams is defined as a coinductive record with equivalence and a *bisimulation*, a binary relation on streams:

```
record _≈_ {A : Set} (x : Stream A) (y : Stream A) : Set where
  coinductive
  field
    hd : x .hd ≡ y .hd
    tl : x .tl ≈ y .tl
```

7 Formalising the syntax and semantics of Lustre

I used the Lustre V6 Reference Manual [6] to model the syntax of a Lustre program into Agda. For the description of the Agda language, I used the Agda Language Reference documentation [7].

Agda can be run from the Emacs editor [8], which is also the basis for Agda's interactive development system for proof assistance, which I will talk about in the following section.

For the scope of this thesis, I only formalised a part of the semantics of Lustre. In this implementation of the formalisation of the semantics of Lustre, I have omitted clock calculus and systems of nodes. The goal of this program is to model a single node, which can use any point-wise operator, and the `pre` and `fby` temporal operators. Then, proofs can be written to prove the (temporal) properties of nodes hold. Parts of the code were written in collaboration with H. Basold. The full source code can be found on the LIACS GitLab server [9].

7.1 Formalised semantics

Using the Agda programming language, the syntax and semantics of Lustre can be formalised as Agda programs. To model the behaviour of Lustre nodes, streams of values, expressions, equations and their semantics have to be defined.

7.1.1 Values

First, I defined the Value type (value.agda) using the standard Agda library for the definitions of integers, vectors, Booleans and rational numbers. I have limited the possible types of a value to Integers, Booleans, Rationals and nil (empty) for practical purposes. Together with their definitions, I defined the semantics of both Boolean and numerical operators. I also defined some predicates and relations for values that can be used to reason about them.

```
data Value : Set where
  int   : ℤ → Value
  bool  : Bool → Value
  ratio : ℚ → Value
  nil   : Value
```

Predicates on values like isPos (is positive), are implemented as follows using the predicates on Integers and Rationals from the Agda standard library. Similar definitions are omitted for reading purposes:

```
data isPos : Value → Set where
  isPos-int   : ∀ x → Int.Positive x → isPos (int x)
  isPos-ratio : ∀ x → Ratio.Positive x → isPos (ratio x)
```

[6] https://www-verimag.imag.fr/DIST-TOOLS/SYNCHRONE/lustre-v6/doc/lv6-ref-man.pdf
[7] https://agda.readthedocs.io/en/v2.6.2.1
[8] https://www.gnu.org/software/emacs/
[9] https://git.liacs.nl/s1914839/lustre

```
data isNeg : Value → Set where
  isNeg-int : ∀ x → Int.Negative x → isNeg (int x)
  isNeg-ratio : ∀ x → Ratio.Negative x → isNeg (ratio x)

isV : Value → Value → Set
isV = flip (_≡_)

isFalse : Value → Set
isFalse = isV (bool false)

isTrue : Value → Set
isTrue = isV (bool true)
```

Relations on values also use the relations from the standard library:

```
data _≤_ : Value → Value → Set where
  ≤-int : ∀ x y → x Int.≤ y → int x ≤ int y
  ≤-ratio : ∀ x y → x Ratio.≤ y → ratio x ≤ ratio y


data _≥_ : Value → Value → Set where
  ≥-int : ∀ x y → x Int.≥ y → int x ≥ int y
  ≥-ratio : ∀ x y → x Ratio.≥ y → ratio x ≥ ratio y

...
```

Boolean logic is lifted from the Boolean logic from the standard library to Values:

```
if_then_else_ : Value → Value → Value → Value
if int x then _ else _ = nil
if bool x then y else z = Data.Bool.if_then_else_ x y z
if ratio x then _ else _ = nil
if nil then _ else _    = nil

...


liftBools : (Bool → Bool → Bool) →
            (Value → Value → Value)
liftBools _ (int _) _ = nil
liftBools f (bool x) (bool y) = bool (f x y)
liftBools _ (bool _) _ = nil
liftBools _ (ratio _) _ = nil
liftBools _ nil _ = nil

_xor_ : Value → Value → Value
_xor_ = liftBools Data.Bool._xor_

...
```

Arithmetic for values. Operators and relations from the standard library are lifted to Values:

```
liftArith : (ℤ → ℤ → ℤ) →
            (ℚ → ℚ → ℚ) →
            (Value → Value → Value)
liftArith f g (int x)   (int y) = int (f x y)
liftArith f g (int x)   _       = nil
liftArith f g (bool x)  _       = nil
liftArith f g (ratio x) (ratio y) = ratio (g x y)
liftArith f g (ratio x) _       = nil
liftArith f g nil       _       = nil

...

_+_ : Value → Value → Value
_+_ = liftArith Int._+_ Ratio._+_

_-_ : Value → Value → Value
_-_ = liftArith Int._-_ Ratio._-_

...
```

7.1.2 Expressions

The syntax of expressions defined like in section 4.1 is expressed below. Note that only non-temporal expressions are formalised like this. The formalisation of temporal operators will be discussed in section 7.1.4. Note that variables are defined as Fin Γ, a type that has Γ - 1 elements. This results in variables not being identified by a unique name but rather by a unique number, corresponding to a *De Bruijn index*. A De Bruijn index is a notation for representing terms in lambda calculus by their "distance" to the binding λ instead of using a variable name [Sel14]. Operator precedence follows the Lustre V6 manual. The declaration of the operator precedence is omitted for reading purposes.

```
data Expr (Γ : ℕ) : Set where
  const    : Value → Expr Γ
  var      : Fin Γ → Expr Γ
  not      : Expr Γ → Expr Γ
  -_       : Expr Γ → Expr Γ
  _and_    : Expr Γ → Expr Γ → Expr Γ
  _or_     : Expr Γ → Expr Γ → Expr Γ
  _xor_    : Expr Γ → Expr Γ → Expr Γ
  _-_      : Expr Γ → Expr Γ → Expr Γ
  _+_      : Expr Γ → Expr Γ → Expr Γ
  _*_      : Expr Γ → Expr Γ → Expr Γ
  if_then_else_ : Expr Γ → Expr Γ → Expr Γ → Expr Γ
  empty    : Expr Γ
```

7.1.3 Semantics of a Lustre expression

The semantics of an expression is implemented as a function which takes an expression and a vector of values, and returns a value. Operators preceded with 'v' are the operators that were defined in value.agda but renamed to avoid naming collisions. These operators return different types based on their inputs.

The outputs of a Lustre node are defined by their corresponding expressions. These expressions can use both input and output variables, which are identified by a de Bruijn index.

This function is then mapped over a stream of values by semExprV. The resulting stream is a stream containing the output values for a certain output variable. This mimics the semantics of point-wise operators as described in section 4.1.1. An empty expression will result in an undefined stream of `nil`s. This line can be removed to disallow empty expressions.

```
semExpr : {n : ℕ} (e : Expr n)
        → Vec Value n
        → Value
semExpr (const e) x    = e
semExpr (var v) x      = lookup x v
semExpr (not e) x      = vnot (semExpr e x)
semExpr (- e) x        = v-- (semExpr e x)
semExpr (e₁ and e₂) x  = (semExpr e₁ x) vand (semExpr e₂ x)
semExpr (e₁ or e₂) x   = (semExpr e₁ x) vor (semExpr e₂ x)
semExpr (e₁ xor e₂) x  = (semExpr e₁ x) vxor (semExpr e₂ x)
semExpr (e₁ - e₂) x    = (semExpr e₁ x) v-  (semExpr e₂ x)
semExpr (e₁ + e₂) x    = (semExpr e₁ x) v+ (semExpr e₂ x)
semExpr (e₁ * e₂) x    = (semExpr e₁ x) v*  (semExpr e₂ x)
semExpr (if e₁ then e₂ else e₃) x = vif (semExpr e₁ x) then
                                      (semExpr e₂ x) else (semExpr e₃ x)
semExpr empty v        = nil

semExprV : {n : ℕ} (e : Expr n)
         → Str (Vec Value n)
         → Str Value
semExprV e v = S.map (semExpr e) v
```

7.1.4 Formalisation of a Lustre node

In formalising the semantics of a Lustre node one has to keep the semantics of temporal operators in mind. A Lustre node is simulated by a new type of node with input, state and output variables. This node is parameterised by the number of inputs, state variables and outputs. A state assigns a value to each expression. A given expression can have different values in different states. Note that the $\bigoplus$ operator is just the '+' operator for natural numbers but renamed to avoid naming collisions.

```
record StateEq (I S : ℕ) : Set where
  constructor stateEq
  field
    eqInit : Expr (I ⊕ S)
    eqStep : Expr (I ⊕ S)
open StateEq public
```

The semantics can be described as a state machine in which the values of the outputs depend on the current value of the inputs and state variables. Temporal operations are simulated by the state equations. In the initial condition, the state variable is determined by eqInit. In the step condition, the state variable is determined by eqStep. Both expressions can use input and state variables.

In the case of simulating `pre`, the initial condition is nil, and the step condition is the original stream. In the case of simulating `a fby b`, the initial condition will be the head of `a` and the step condition the tail of `b`.

In general, expressions of type `pre b` are written as `s` where `s` refers to a state equation with both initial condition `empty` and step condition `a`.

Expressions of type `a fby b` are written as `if s then a else b` statements where `s` refers to a state equation with initial condition True and step condition False. An example this notation of this can be found in section 7.3.3.

A node needs to be constructed with the output and state equations (outEqs and stateEqs). Each output variable has one output expression and for each state variable there is one state equation.

```
record Node (I S O : ℕ) : Set where
  constructor node
  field
    outEqs   : Vec (Expr (I ⊕ S)) O
    stateEqs : Vec (StateEq I S) S
open Node public
```

The semantics of a system of equations in a Lustre node is illustrated in figure 8. Where the head of each transition and the output are respectively determined by:

Vec (Str Value) I × Vec Value S → Vec Value O

Vec (Str Value) I × Vec Value S → Vec (Str Value) I × Vec Value S.

The head and tail give a state machine with state type Vec (Str Value) I × Vec Value S and output type Vec Value O.

The state model S an infinite sequence of the form $(S : s_0, s_1, s_2, \ldots,)$ where s_0 is the initial state of the computation and each state $(s_i\ ,\ 0 \leq i)$ is the state of the system at time i.

Figure 8 shows the state diagram of the state machine. The output at $t = 0$ (Σ_0^{out}) is determined by the initial input Σ_0^{in} and the initial state vector (sv_0). The letter Σ is used to signify a collection of streams σ in a vector (which I will call *packed streams*). Each step, the state vector is updated according to the state equations and the computation continues with the tail of the input.

The linear computation of the outputs ensures that the computation of the output terminates when the state of the system at $t = x$ is requested.

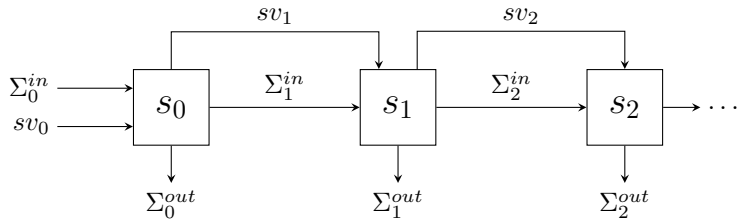


Figure 8: State model for the semantics of a Lustre node

The semantics of a node in Lustre is a function which takes the input streams returns the output streams. Note that eqSysSem is a forward reference. The order of the functions are reversed for reading purposes.

```
nodeSem : Node I S O → Vec (Str Value) I → Vec (Str Value) O
nodeSem n i = eqSysSem (outEqs n) (stateEqs n) i
```

The initial state vector is made by the mapping of the initial equations to the head of the input vector. Then, a system of equations is simulated by using this initial state vector together with the input streams as input.

Equations can use the output of other equations as terms because they were previously defined in terms of the input stream and the state vector. Functions that operate on streams that are packed in a packed stream like `hdvs` and `svtovs` are defined in `vecstr.agda`. `hdvs` returns a vector of values with the head of each stream. `svtovs` converts a stream of vectors to a packed stream.

```
eqSysSem : Vec (Expr (I ⊕ S)) O →
           Vec (StateEq I S) S →
           Vec (Str Value) I →
           Vec (Str Value) O
eqSysSem oe se i =
  let init = Vec.map (λ e → semExpr (eqInit e) (hdvs i ++ replicate nil)) se
  in eqSysSemCoiter oe se (i , init)
```

Every state has access to the output and state equations, which remain unchanged when changing from state to state.

```
eqSysSemCoiter : Vec (Expr (I ⊕ S)) O →
                 Vec (StateEq I S) S →
                 Vec (Str Value) I × Vec Value S →
                 Vec (Str Value) O
eqSysSemCoiter oe se = svtovs ∘ coiter (coiter-hd oe) (coiter-tl se)
```

In coiter-hd, the head of the output stream is determined by the head of the input streams and the current state variables. These are then mapped to the output vector.

```
coiter-hd : Vec (Expr (I ⊕ S)) O →
            Vec (Str Value) I × Vec Value S →
            Vec Value O
coiter-hd oe (i , s) = Vec.map (λ e → semExpr e (hdvs i ++ s)) oe
```

In coiter-tl the state vector is updated and the new input of the system is the tail of the previous step.

```
coiter-tl : Vec (StateEq I S) S →
            Vec (Str Value) I × Vec Value S →
            Vec (Str Value) I × Vec Value S
coiter-tl se (i , s) = (tlvs i , state-step se i s)
```

In state-step, the state vector is updated and the new input of the system is the tail of the previous step.

```
state-step : Vec (StateEq I S) S →
             Vec (Str Value) I → Vec Value S →
             Vec Value S
state-step se i s = Vec.map (λ e → semExpr (eqStep e) (hdvs i ++ s)) se
```

We now give a version of the semantics that als displays the state variables to the outside (nodeFullSem). The theorem nodeFullSem $\sim$ nodeSem shows that it restricts to the normal semantics, if we hide the state variables. The theorem was written by my supervisor and can be found in the source code on GitLab.

```
coiterFull-hd : Vec (Expr (I ⊕ S)) O →
                Vec (Str Value) I × Vec Value S →
                Vec Value (S ⊕ O)
coiterFull-hd oe (i , s) = (s ++ Vec.map (λ e → semExpr e (hdvs i ++ s)) oe)

eqSysFullSemCoiter : Vec (Expr (I ⊕ S)) O →
                     Vec (StateEq I S) S →
                     Vec (Str Value) I × Vec Value S →
                     Vec (Str Value) (S ⊕ O)
eqSysFullSemCoiter oe se = svtovs ∘ coiter (coiterFull-hd oe) (coiter-tl se)

eqSysFullSem : Vec (Expr (I ⊕ S)) O →
               Vec (StateEq I S) S →
               Vec (Str Value) I →
               Vec (Str Value) (S ⊕ O)
eqSysFullSem oe se i =
  let init = Vec.map (λ e → semExpr (eqInit e) (hdvs i ++ replicate nil)) se
  in eqSysFullSemCoiter oe se (i , init)

nodeFullSem : Node I S O → Vec (Str Value) I → Vec (Str Value) (S ⊕ O)
nodeFullSem n i = eqSysFullSem (outEqs n) (stateEqs n) i
```

7.2 A Lustre node in Agda

Using the formalisation of a Lustre node in Agda, we can model the 'plus' node from section 4.2.

```
module plusnode where
```

Initialisation of the number of input, output and state parameters:

```
I = 2
S = 0
O = 1

open import node I S O
```

The 'plus' node has two inputs; a and b. The state of the variables can be looked up by finding the index of their corresponding stream in the input vector.

```
a = # 0
b = # 1
```

```
aE bE : Ex
aE = var a
bE = var b
```

The output expression corresponds to $c = a + b$ and is defined like this:

```
c : Ex
c = aE + bE

oe : Vec Ex O
oe = c :: []
```

This node has no state expressions as there are no temporal operators.

```
se : Vec StateEq S
se = []
```

The node is constructed by giving the output and state equations as parameters.

```
n : Node
n = node oe se
```

The node takes two input streams and returns a single output stream which is given by the semantics formalised in section 7.1.4.

```
sem : Vec (Str Value) I → Vec (Str Value) O
sem = nodeSem n
```

By formalising the semantics of a Lustre node in Agda, proofs can be written about the output of the node, given some input. To write proofs about Lustre nodes, some kind of logic system has to be formalised so properties of vectors of streams can described. This logic can be used to write proofs about those properties.

7.3 LTL in Agda

The domain of a predicate on the state of a node is parameterised by the number of streams in the state object. A predicate over some type A (Pred A) is defined in the standard library as a unary relation that can be viewed as some property that elements of A might satisfy. P : Pred A can also be seen as a subset of A containing the elements that satisfy P.

```
StrPred : ℕ → Set₁
StrPred n = Pred (Vec (Str Value) n) _
```

The operators of propositional calculus are static. One can reason about the 'current' state of the system using these operators with the atHead function. The `hdvs` function returns a vector containing the heads of all streams at the current state.

```
atHead : ∀{n} → (Vec Value n → Set) → StrPred n
atHead P σ = P (hdvs σ)
```

The proof system to reason about the semantics of a Lustre node will be LTL. A constructive (Agda) version of LTL is defined below:

```
record □ {n : ℕ} (φ : StrPred n) (σ : Vec (Str Value) n) : Set where
  coinductive
  field
    hd : φ σ
    tl : □ φ (tlvs σ)
```

The always (□) operator is defined as a coinductive record over a set of streams (σ). φ holds for σ if it holds for the entire stream, so for the head and (coinductively) for the tail.

```
data ◇ {n : ℕ} (φ : StrPred n) (σ : Vec (Str Value) n) : Set where
  now : φ σ → ◇ φ σ
  later : ◇ φ (tlvs σ) → ◇ φ σ
```

The eventually operator (◇) is defined as a data type. It holds when either ◇ φ holds for σ at the 'current' state (now) or at some future state (later).

```
data ◯ {n : ℕ} (φ : StrPred n) (σ : Vec (Str Value) n) : Set where
  next : φ (tlvs σ) → ◯ φ σ
```

The next (◯) operator is defined as a data type with a single constructor. ◯ φ holds for σ if it holds at the next state (next).

```
data _𝒰_ {n : ℕ} (φ : StrPred n) (ψ : StrPred n) (σ : Vec (Str Value) n) : Set where
  here : ψ σ → (φ 𝒰 ψ) σ
  step : φ σ → (φ 𝒰 ψ) (tlvs σ) → (φ 𝒰 ψ) σ
```

The static operators can be used to reason about a state of the system. The `lookup` function returns the value from the vector of values at the current state from an index (Fin n). The operators use the relations defined earlier in 7.1.1:

```
_>_ : ∀ {n} → Fin n → Value → Vec Value n → Set
_>_ i n v = lookup v i V.> n

...

_=_ : ∀ {n} → Fin n → Value → Vec Value n → Set
_=_ i n v = lookup v i V.- n

_≡_ : ∀ {n} → Fin n → Fin n → Vec Value n → Set
(x ≡ y) v = lookup v x ≡ lookup v y

isNil : ∀{n} → Fin n → Vec Value n → Set
isNil i v = lookup v i ≡ nil

isFalse : ∀{n} → Fin n → Vec Value n → Set
```

```
isFalse i v = lookup v i ≡ bool false

isTrue : ∀{n} → Fin n → Vec Value n → Set
isTrue i v = lookup v i ≡ bool true

...
```

Propositional connectives use the connectives on sets from the standard library:

```
_∧_ : ∀ {n} → (φ : StrPred n) → (ψ : StrPred n) → StrPred n
_∧_ = λ φ ψ → φ ∩ ψ

_∨_ : ∀ {n} → (φ : StrPred n) → (ψ : StrPred n) → StrPred n
_∨_ = λ φ ψ → φ ∪ ψ

_⟶_ : ∀ {n} → (φ : StrPred n) → (ψ : StrPred n) → StrPred n
_⟶_ = λ φ ψ → φ ⇒ ψ

¬_ : ∀ {n} → (φ : StrPred n) → StrPred n
¬_ = λ φ → ∁ φ

neg : ∀ {n} → StrPred n → StrPred n
neg φ = φ ∘ vmap (map V.not)
```

7.3.1 The Agda LTL proof system

I will show that this proof system is consistent with the local LTL proof system presented by Cini and Francalanza [CF15] that allows reasoning about whether any finite stream σ at the current state of the system satisfies or violates some formula φ. The (extended) syntax can be found in Figure 1 of their paper. The proof rules can be found in Figure 2.

The axioms of the proof system presented by Clini and Francalanza can be related to the proof system introduced in 7.3 in which LTL is formalised in Agda. I will show that the axioms of their online monitoring proof system are theorems of the proof system presented in this thesis. For any theorem in their proof system that satisfies φ, a theorem can be constructed in Agda that satisfies φ. The same can be done for theorems that violate φ.

A satisfaction judgement in in the online monitoring proof system is denoted by $\sigma \vdash^{+} \varphi$ and a violation judgement is denoted by $\sigma \vdash^{-} \varphi$. This corresponds with φ σ in Agda. If a proof can be constructed that holds for σ, then φ is satisfied for σ. The inverse says that if φ doesn't hold for σ, a proof can be constructed that violates φ (φ σ → ⊥).

```
_⊢⁺_ _⊢⁻_ : {n : ℕ} (σ : Vec (Str Value) n) (φ : StrPred n) → Set
σ ⊢⁺ φ = φ σ
σ ⊢⁻ φ = φ σ → ⊥
```

The satisfaction and violation rules are derived below. Note that $\mathrm{Sum.inj}_1$ and $\mathrm{Sum.inj}_2$ are the constructors of the sum type (defined as a disjoint union). $\Sigma.\mathrm{proj}_1$ and $\Sigma.\mathrm{proj}_2$ are the constructors of a dependent product type.

$$\mathsf{pNeg} : \forall\ \{n\}\ \{\sigma\}\ \{\varphi : \mathsf{StrPred}\ n\} \to \sigma \vdash^{-} \varphi \to \sigma \vdash^{+} (\neg\ \varphi)$$
$$\mathsf{pNeg}\ p\ q = p\ q$$

$$\mathsf{pAnd} : \forall\ \{n\}\ \{\sigma\}\ \{\varphi\ \psi : \mathsf{StrPred}\ n\} \to \sigma \vdash^{+} \varphi \to \sigma \vdash^{+} \psi \to \sigma \vdash^{+} (\varphi \wedge \psi)$$
$$\mathsf{pAnd}\ p\ q = p\ ,\ q$$

$$\mathsf{pOr}_1 : \forall\ \{n\}\ \{\sigma\}\ \{\varphi\ \psi : \mathsf{StrPred}\ n\} \to \sigma \vdash^{+} \varphi \to \sigma \vdash^{+} (\varphi \vee \psi)$$
$$\mathsf{pOr}_1\ p = \mathsf{Sum.inj}_1\ p$$

$$\mathsf{pOr}_2 : \forall\ \{n\}\ \{\sigma\}\ \{\varphi\ \psi : \mathsf{StrPred}\ n\} \to \sigma \vdash^{+} \psi \to \sigma \vdash^{+} (\varphi \vee \psi)$$
$$\mathsf{pOr}_2\ q = \mathsf{Sum.inj}_2\ q$$

$$\mathsf{pNext} : \forall\ \{n\}\ \{\sigma\}\ \{\varphi : \mathsf{StrPred}\ n\} \to (\mathsf{tlvs}\ \sigma) \vdash^{+} \varphi \to \sigma \vdash^{+} \bigcirc\ \varphi$$
$$\mathsf{pNext} = \mathsf{next}$$

$$\mathsf{pUnt}_1 : \forall\ \{n\}\ \{\sigma\}\ \{\varphi\ \psi : \mathsf{StrPred}\ n\} \to \sigma \vdash^{+} \psi \to \sigma \vdash^{+} (\varphi\ \mathcal{U}\ \psi)$$
$$\mathsf{pUnt}_1\ p = \mathsf{here}\ p$$

$$\mathsf{pUnt}_2 : \forall\ \{n\}\ \{\sigma\}\ \{\varphi\ \psi : \mathsf{StrPred}\ n\} \to \sigma \vdash^{+} \varphi \to \mathsf{tlvs}\ \sigma \vdash^{+} (\varphi\ \mathcal{U}\ \psi) \to \sigma \vdash^{+} (\varphi\ \mathcal{U}\ \psi)$$
$$\mathsf{pUnt}_2\ p\ q = \mathsf{step}\ p\ q$$

$$\mathsf{nNeg} : \forall\ \{n\}\ \{\sigma\}\ \{\varphi : \mathsf{StrPred}\ n\} \to \sigma \vdash^{+} \varphi \to \sigma \vdash^{-} (\neg\ \varphi)$$
$$\mathsf{nNeg}\ p\ q = q\ p$$

$$\mathsf{nAnd}_1 : \forall\ \{n\}\ \{\sigma\}\ \{\varphi\ \psi : \mathsf{StrPred}\ n\} \to \sigma \vdash^{-} \varphi \to \sigma \vdash^{-} (\varphi \wedge \psi)$$
$$\mathsf{nAnd}_1\ p\ q = p\ (\Sigma.\mathsf{proj}_1\ q)$$

$$\mathsf{nAnd}_2 : \forall\ \{n\}\ \{\sigma\}\ \{\varphi\ \psi : \mathsf{StrPred}\ n\} \to \sigma \vdash^{-} \psi \to \sigma \vdash^{-} (\varphi \wedge \psi)$$
$$\mathsf{nAnd}_2\ p\ q = p\ (\Sigma.\mathsf{proj}_2\ q)$$

$$\mathsf{nOr} : \forall\ \{n\}\ \{\sigma\}\ \{\varphi\ \psi : \mathsf{StrPred}\ n\} \to \sigma \vdash^{-} \varphi \to \sigma \vdash^{-} \psi \to \sigma \vdash^{-} (\varphi \vee \psi)$$
$$\mathsf{nOr}\ p\ q = \mathsf{Sum.[}\ p\ ,\ q\ \mathsf{]}$$

$$\mathsf{nNext} : \forall\ \{n\}\ \{\sigma\}\ \{\varphi : \mathsf{StrPred}\ n\} \to (\mathsf{tlvs}\ \sigma) \vdash^{-} \varphi \to \sigma \vdash^{-} \bigcirc\ \varphi$$
$$\mathsf{nNext}\ p\ (\mathsf{next}\ q) = p\ q$$

$$\mathsf{nUnt}_1 : \forall\ \{n\}\ \{\sigma\}\ \{\varphi\ \psi : \mathsf{StrPred}\ n\} \to \sigma \vdash^{-} \varphi \to \sigma \vdash^{-} \psi \to \sigma \vdash^{-} (\varphi\ \mathcal{U}\ \psi)$$
$$\mathsf{nUnt}_1\ p\ q\ (\mathsf{here}\ \psi\sigma) = q\ \psi\sigma$$
$$\mathsf{nUnt}_1\ p\ q\ (\mathsf{step}\ \varphi\sigma\ u) = p\ \varphi\sigma$$

$$\mathsf{nUnt}_2 : \forall\ \{n\}\ \{\sigma\}\ \{\varphi\ \psi : \mathsf{StrPred}\ n\} \to \sigma \vdash^{-} \psi \to \mathsf{tlvs}\ \sigma \vdash^{-} (\varphi\ \mathcal{U}\ \psi) \to \sigma \vdash^{-} (\varphi\ \mathcal{U}\ \psi)$$
$$\mathsf{nUnt}_2\ p\ q\ (\mathsf{here}\ \psi\sigma) = p\ \psi\sigma$$
$$\mathsf{nUnt}_2\ p\ q\ (\mathsf{step}\ \varphi\sigma\ u) = \mathsf{nUnt}_2\ (\lambda\ \_ \to q\ u)\ (\lambda\ \_ \to q\ u)\ u$$

7.3.2 The stopwatch node

The following is a model of the stopwatch node introduced in chapter 4.5. The node uses 3 state variables to simulate `fby`, `pre(time)` and `pre(running)`.

```
module stopwatch where

  I = 2
  S = 3
  O = 2

  open import node

  ex = Expr (I Nat.+ S)

  False True Zero One : ex
```

Constants are declared beforehand and can be used inside expressions.

```
False = const (bool Bool.false)
True = const (bool Bool.true)
Zero = const (int (ℤ.pos 0))
One  = const (int (ℤ.pos 1))

reset      = # 0
start_stop = # 1

stateA     = # 2
stateB     = # 3
stateC     = # 4

r ss a b c : ex

r  = var reset
ss = var start_stop

a = var stateA
b = var stateB
c = var stateC
```

Note that the order of the equations for `running` and `time` are reversed. The output of the `running` equation is needed as input for the `time` equation.

The `fby` operator is formalised as an `if then else` statement which uses a state variable as it's conditional. For this node, the values of running and time at t = 0 will be False and Zero respectively. On the next steps (t > 0) the values the alternative applies. `pre`(time) and `pre`(running) are denoted by `b` and `c` respectively, signifying the value of `time` and `running` at the previous step.

```
running : ex
running = if a then False else (
                                if ss then (not c)
                                else c)

time : ex
time = if a then Zero else (
                            if r then Zero
                            else if running then (b +' One)
                            else b)

oe : Vec ex O
oe = time :: running :: []

se : Vec (StateEq I S) S
se = stateEq True False :: stateEq time time :: stateEq running running :: []

n : Node I S O
n = node oe se

sem : Vec (Str Value) I → Vec (Str Value) O
sem = nodeSem n
```

7.3.3 The trigger node

This node simulates a trigger which starts and stops a stopwatch. It converts a trigger signal defined as `True` when the trigger is pressed and `false` when the trigger is released, to a signal that becomes true at a rising edge. This output can be used as input for the stopwatch node to create a system of nodes. It also outputs separate `toggle_on` and `toggle_off` signals which take apart the `start_stop` signal into a `start` and a `stop` signal.

As the equations are somewhat simpler, it's a good place to start writing proofs about their behaviour. Like the stopwatch node, it uses 3 state variables for 1 `fby` and 2 `pre` operators.

```
module trigger where

I = 1
S = 3
O = 4

open import node

ex = Expr (I ⊕ S)

False True : ex

False = const (bool Bool.false)
True = const (bool Bool.true)

trigger = # 0
stateA = # 1
stateB = # 2
stateC = # 3

t a b c : ex

t = var trigger
a = var stateA
b = var stateB
c = var stateC
```

The `edge` equation detects a rising edge in the input. This is the case when a `False` is followed by a `True` at the next time step. This is formalised by the expression `(t and (not c))` in which `t` denotes the input and `c` denotes the state of t in the previous time step, which is analogous with `pre`(t). It can be used as the `start_stop` input for the stopwatch node.

```
edge : ex
edge = if a then False else (t and (not c))
```

The `mem` equation stores the information of the last rising edge. The `mem` signal starts at False and inverts when `edge` becomes true. When `edge` becomes true a second time, `mem` inverts again and so on. This information is necessary to decide whether `start_stop` or `reset` should become True.

```
mem : ex
mem = if a then False else (if edge then (not b) else b)
```

The `toggle_on` equation is defined by `edge` $\wedge$ `mem`. At the first rising edge, `toggle_on` becomes `True` for one time instant.

```
toggle_on : ex
toggle_on = edge and mem
```

At the next rising edge, `toggle_off` becomes `True`. At each rising edge either `toggle_off` or `toggle_off` becomes true in turn.

```
toggle_off : ex
toggle_off = edge and (not mem)

oe : Vec ex O
oe = edge :: mem :: toggle_on :: toggle_off :: []

se : Vec (StateEq I S) S
se = (stateEq True False) :: stateEq mem mem :: stateEq t t :: []

n : Node I S O
n = node oe se

sem : Vec (Str Value) I → Vec (Str Value) O
sem = nodeSem n
```

7.3.4 A proof on the trigger node

The following contains the proof of the LTL formula φ : `trig-ff` $\wedge \circ$ `trig-tt` $\rightarrow \circ$ `edge-tt`. Whenever the trigger signal is False, followed by a True at the next time instant, $\circ$ `edge-tt` should become True. To prove this, we use a Lemma that contains a proof for φ for the semantics of a node that displays the state variables to the outside and then use this Lemma to prove that φ holds for the original semantics. This method can also be used to prove other formulas about other nodes.

The lemma hideState allows us to prove that a weakened formula never uses the state variables. Hence, it can be proven purely on the input-output semantics.

```
hideState : ∀ {I S O} (n : Node I S O) (φ : StrPred (I ⊕ O)) → Resp~* φ →
            ∀ i → weaken I S φ (i ++ nodeFullSem n i) → φ (i ++ nodeSem n i)
hideState {I} {S} n φ φ-resp~* i = φ-resp~* (disect-lem n i)
```

From hideState, we obtain the following theorem that allows us to use a lemma that proves something involving the internal state variables of a node to prove a weakened formula. With this

theorem, we can "open up" a node, prove ψ on the state variables, then show that $\psi \to \varphi$ and then forget about the state variables to obtain φ.

```
fromStateLemma : ∀ {I S O} (n : Node I S O)
                 (ψ : StrPred (I ⊕ (S ⊕ O)))
                 (φ : StrPred (I ⊕ O)) →
                 (ψ ⊆ weaken I S φ) →
                 Resp~* φ →
                 (∀ i → ψ (i ++ nodeFullSem n i)) →
                 ∀ i → φ (i ++ nodeSem n i)
fromStateLemma n ψ φ ψ⊆φ φ-resp~* ∀ψ i =
  hideState n φ φ-resp~* i (ψ⊆φ (∀ψ i))
```

These lemmas can now be used to write a complete proof of φ.

```
t₀ : Fin I
t₀ = # 0

e₀ m₀ on₀ off₀ : Fin O
e₀ = # 0
m₀ = # 1
on₀ = # 2
off₀ = # 3


module Lemma where
```

First, the de Bruijn indices are raised to address both input *and* state variables.

```
iT' sA sB sC oE' oM' oOn' oOff' : Fin (I ⊕ (S ⊕ O))
iT' = inject+ (S ⊕ O) t₀
sA = # 1
sB = # 2
sC = # 3
oE' = raise (I ⊕ S) e₀
oM' = raise (I ⊕ S) m₀
oOn' = raise (I ⊕ S) on₀
oOff' = raise (I ⊕ S) off₀
```

We define the predicates over streams we need to prove φ. The testVar function checks whether the Value at a certain index satisfies some predicate P. In this case, `isFalse` and `isTrue` are used as predicates over values.

```
trig-ff trig-tt c-ff edge-tt : StrPred (I ⊕ (S ⊕ O))
trig-ff = testVar V.isFalse iT'
trig-tt = testVar V.isTrue iT'
c-ff = testVar V.isFalse sC
edge-tt = testVar V.isTrue oE'
```

The full theorems φ and ψ we want to prove. φ only considers in- and output variables whereas ψ also considers the state variables.

```
φ₀ φ ψ₀ ψ : StrPred (I ⊕ (S ⊕ O))
φ₀ = (trig-ff ∧ ∘ trig-tt) ⟶ ∘ edge-tt
φ = □ φ₀
ψ₀ = (trig-ff ∧ ∘ trig-tt) ⟶ (∘ c-ff ∧ ∘ edge-tt)
ψ = □ ψ₀
```

`e-up` states that for an input `x` with `x .hd = false` and `x .tl .hd = true` (so a False followed by a True at the next instant) give that ∘ `edge-tt` for the full semantics of n (nodeFullSem n).

```
e-up : ∀ {x s u v} →
       x .hd ≡ u → x .tl .hd ≡ v →
       u ≡ bool false → v ≡ bool true →
       ∘ edge-tt (nSem n Vec.[ x ] s)
e-up {x} p q refl refl = next r

  where
    open ≡-Reasoning

    r =
      begin
        (x .tl .hd) V.and V.not (x .hd)
      ≡⟨ cong (λ t → t V.and V.not (x .hd)) q ⟩
        (bool true) V.and V.not (x .hd)
      ≡⟨ cong (λ t → (bool true) V.and V.not t) p ⟩
        (bool true) V.and V.not (bool false)
      ≡⟨⟩
        bool true
      ■
```

`c-down` states that for an input x with `x .hd = false` and `x .tl .hd = true` (so a False followed by a True at the next instant) give that ∘ `c-ff` for nodeFullSem n.

```
c-down : ∀ {x s u v} →
         x .hd ≡ u → x .tl .hd ≡ v →
         u ≡ bool false → v ≡ bool true →
         ∘ c-ff (nSem n Vec.[ x ] s)
c-down p q refl refl = next p
```

The proof for ψ_0 applied to n with any input i is a pair of `c-down` and `e-up`, constructing ∘ `c-ff` ∧ ∘ `edge-tt`.

```
pψ₀ : ∀ {i} → (ψ₀ on n) i
pψ₀ {x :: []} (t-ff , next ∘t-tt) = c-down t-ff ∘t-tt refl refl , e-up t-ff ∘t-tt refl refl
```

If ψ_0 is satisfied for nodeFullSem n, then φ_0 will also be satisfied for nodeFullSem n.

```
ψ₀→φ₀ : stateOf n have ψ₀ ⇒₂ stateOf n have φ₀
ψ₀→φ₀ ψ₀x p = proj₂ (ψ₀x p)
```

ψ_0 is satisfied for nodeFullSem n.

```
ψ₀-safe : ∀ {i s} → (stateOf n have ψ₀) i s
ψ₀-safe {x :: []} {s} (t-ff , next ∘t-tt) = c-down t-ff ∘t-tt refl refl , e-up t-ff ∘t-tt refl refl
```

Proof that $\Box\varphi$ holds for nodeFullSem n (for any input). This proof requires that ψ is an *invariant*, meaning that if ψ holds for some state, it holds for every state. The proofs can be found in NodeReasoning.lagda.

```
pφ : ∀ {i} → φ (i ++ nodeFullSem n i)
pφ = □-safety-node n ψ₀ ψ₀-resp φ₀-resp ψ₀→φ₀ ψ₀-safe pψ₀
```

Now, we need to prove that φ holds for the version of the semantics that doesn't display the state variables to the outside (sem i).

```
iT oE oM oOn oOff : Fin (I ⊕ O)
iT = inject+ O t₀
oE = raise I e₀
oM = raise I m₀
oOn = raise I on₀
oOff = raise I off₀

trig-ff trig-tt : StrPred (I ⊕ O)
trig-ff = testVar V.isFalse iT
trig-tt = testVar V.isTrue iT

edge-tt : StrPred (I ⊕ O)
edge-tt = testVar V.isTrue oE
```

φ is now a StrPred over (I + O) instead of (I + S + O).

```
φ : StrPred (I ⊕ O)
φ = □ ((trig-ff ∧ ∘ trig-tt) ⟶ ∘ edge-tt)

w-trig-ff : ∀ x → (weaken I S trig-ff) x ≡ Lemma.trig-ff x
w-trig-ff x = weaken-proj₁ {atHead₁ V.isFalse} I S t₀ x

w-trig-tt : ∀ x → (weaken I S trig-tt) x ≡ Lemma.trig-tt x
w-trig-tt x = weaken-proj₁ {atHead₁ V.isTrue} I S t₀ x

w-edge : ∀ x → (weaken I S edge-tt) x ≡ p-assoc I {S} {O} Lemma.edge-tt x
w-edge x = weaken-proj₂ {atHead₁ V.isTrue} I S e₀ x

→assoc : Lemma.edge-tt ⊆ p-assoc I {S} Lemma.edge-tt
```

```
  →assoc {x} p = subst Lemma.edge-tt (sym (assoc-id x)) p
    where
      assoc-id : ∀ x → v-assoc {_} I S {O} x ≡ x
      assoc-id (_ :: _) = refl
```

Proof of φ_0 on the weakened state variables.

```
Lemma→φ₀ : Lemma.φ₀ ⊆ weaken I S ((trig-ff ∧ ∘ trig-tt) ⟶ ∘ edge-tt)
Lemma→φ₀ {x} p (t-ff , next ∘t-tt) =
  let r = tlvs-weaken I {S} trig-tt {x} ∘t-tt
      (next q) = p (subst id (w-trig-ff x) t-ff , next (subst id (w-trig-tt (tlvs x)) r))
      q' = subst id (sym (w-edge (tlvs x))) (→assoc {tlvs x} q)
  in next (weaken-tlvs I {S} edge-tt {x} q')
```

If $\psi \to \varphi$ we can forget about the state variables and obtain φ.

```
Lemma→φ : Lemma.φ ⊆ weaken I S φ
Lemma→φ p = weaken-□ I (□-mono Lemma→φ₀ p)

pφ : ∀ i → φ (i ++ sem i)
pφ = fromStateLemma n Lemma.φ φ Lemma→φ φ-resp (λ i → Lemma.pφ)
```

8 Conclusions and Further Research

By Lustre's design, Agda can be a useful tool for program verification. I have shown that a subset of Lustre's syntax and semantics (described in section 4) can be formalised in Agda by demonstration in section 4.3. This formalisation makes it possible to write proofs about properties of Lustre nodes. I've demonstrated that my encoding of LTL in Agda is useful for this purpose in section 7.3. In section 7.3.3 I've demonstrated how my formalisation of Lustre's syntax and semantics can be used to model the execution of a Lustre program and I've provided a proof about a temporal property of this node. My research question: *How can formalisation of the semantics of Lustre be achieved in Agda and how can Agda be used to prove properties of Lustre programs?* is therefore answered by demonstration.

As stated in section 7, clocks and clock calculus are omitted from my formalisation. The implementation of clocks, the temporal operators `when` and `current` together with their semantics on flows are therefore still subject for research. My implementation of the formalisation of the syntax and semantics can also be further improved by writing programs that make the writing of nodes easier, for example by automatically generating the in/output streams, equations and functions from a given node definition. The process of creating proofs for Lustre nodes can be simplified by writing functions that aid generalised proof writing. This thesis doesn't contain a proof that uses predicates on outputs as premises and is also still subject for research.

Additionally, a compiler could be written to directly translate a Lustre program to its formalisation in Agda. Much work is still needed to formalise the full Lustre syntax and semantics in Agda.

References

[AR02] Farhad Arbab and Jan Rutten. A coinductive calculus of component connectors. pages 34--55, 09 2002.

[Bas14] Henning Basold. Transformation von scade-modellen zur smt-basierten verifikation. *CoRR*, abs/1403.2752, 2014.

[BBD+17] Timothy Bourke, Lélio Brun, Pierre-Évariste Dagand, Xavier Leroy, Marc Pouzet, and Lionel Rieg. A formally verified compiler for lustre. *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2017.

[BCG88] Gérard Berry, Philippe Couronne, and Georges Gonthier. Synchronous programming of reactive systems: an introduction to esterel. 08 1988.

[BCPvD99] Saddek Bensalem, P. Caspi, Catherine Parent-vigouroux, and C. Dumas. A methodology for proving control systems with lustre and pvs. pages 89 -- 107, 12 1999.

[BD08] Ana Bove and Peter Dybjer. Dependent types at work. pages 57--99, 01 2008.

[BDN09] Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of agda – a functional language with dependent types. volume 5674, pages 73--78, 08 2009.

[Ber89] Gérard Berry. Real time programming: Special purpose or general purpose languages. *Information Processing*, 89, 01 1989.

[BG01] Henk (Hendrik) Barendregt and Herman Geuvers. *Proof-Assistants Using Dependent Type Systems*, volume 2, pages 1149--1238. 05 2001.

[BGHM14] Henning Basold, Henning Günther, Michaela Huhn, and Stefan Milius. An open alternative for smt-based verification of scade models. In *FMICS*, 2014.

[BJP22] Timothy Bourke, Paul Jeanmaire, and Marc Pouzet. Towards a denotational semantics of streams for a verified lustre compiler. 2022.

[Bun70] Alan Bundy. The automation of proof by mathematical induction. *Handbook of Automated Reasoning*, 1, 02 1970.

[C.12] Paulin-Mohring C. Introduction to the coq proof-assistant for practical software verification. 7682, 2012.

[CAB+96] Robert Constable, S. Allen, H. Bromley, W. Cleaveland, J. Cremer, Robert Harper, Douglas Howe, T. Knoblock, N. Mendler, Prakash Panangaden, J. Sasaki, and Scott Smith. Implementing mathematics with the nuprl proof development system. 01 1996.

[Cas92] Paul Caspi. Clocks in dataflow languages. *Theoretical Computer Science*, 94(1):125--140, 1992.

[Cas94] P. Caspi. Towards recursive block diagrams. *IFAC Proceedings Volumes*, 27:81--85, 06 1994.

[CF15] Clare Cini and Adrian Francalanza. An ltl proof system for runtime verification. volume 9035, pages 581--595, 04 2015.

[CGB+18] Darren Cofer, Andrew Gacek, John Backes, Michael Whalen, Lee Pike, Adam Foltzer, Michal Podhradsky, Gerwin Klein, Ihor Kuz, June Andronick, Gernot Heiser, and Douglas Stuart. A formal approach to constructing secure air vehicle software. *Computer*, 51:14--23, 11 2018.

[CMCHG96] Edmund Clarke, Kenneth McMillan, Sergio Campos, and Vassili Hartonas-Garmhausen. Symbolic model checking. pages 419--427, 01 1996.

[CPHP87] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John Plaice. Lustre: A declarative language for programming synchronous systems*. 1987.

[dD08] Lydie du Bousquet and Michel Delaunay. Towards mutation analysis for lustre programs. *Electronic Notes in Theoretical Computer Science*, 203(4):35--48, 2008. Proceedings of the International Workshop on Model-driven High-level Programming of Embedded Systems (SLA++P 2007).

[DHJ+01] Matthew B. Dwyer, John Hatcliff, Roby Joehanes, Shawn Laubach, Corina S. Pasareanu, Robby, Hongjun Zheng, and Willem Visser. Tool-supported program abstraction for finite-state verification. *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001*, pages 177--187, 2001.

[DLS78] Richard Demillo, R.J. Lipton, and F.G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11:34 -- 41, 05 1978.

[GBBG86] Paul Guernic, Albert Benveniste, Patricia Bournai, and Thierry Gautier. Signal : a data flow oriented language for signal processing. *Acoustics, Speech and Signal Processing, IEEE Transactions on*, ASSP-34:362 -- 374, 05 1986.

[Geu08] Herman Geuvers. Introduction to type theory. volume 5520, pages 1--56, 01 2008.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305--1320, 1991.

[HP14] Allen Hazen and Francis Pelletier. Gentzen and jaśkowski natural deduction: Fundamentally similar but importantly different. *Studia Logica*, 102:1103--1142, 12 2014.

[HT08] George Hagen and Cesare Tinelli. Scaling up the formal verification of lustre programs with smt-based techniques. In *2008 Formal Methods in Computer-Aided Design*, pages 1--9, 2008.

[JR99] Bart Jacobs and Jan Rutten. A tutorial on (co)algebras and (co)induction. *Bulletin of the EATCS*, 62, 10 1999.

[KAE+14] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an os microkernel. *ACM Transactions on Computer Systems (TOCS)*, 32, 02 2014.

[Klo00] jan willem Klop. Term rewriting systems. 08 2000.

[KS98] G. Kotonya and Ian Sommerville. Integrating safety analysis and requirements engineering. pages 259--271, 01 1998.

[Lju99] M. Ljung. Formal modelling and automatic verification of lustre programs using np-tools. 12 1999.

[LT93] Nancy Leveson and Clark Turner. An investigation of the therac-25 accidents. *IEEE Computer*, 26:18--41, 01 1993.

[Maz96] Leonardo Mazzini. Ariane 5 flight 501 failure report by the inquiry board, 07 1996.

[MBAK11] Olfa Mosbahi, Leila Ben Ayed, and Mohamed Khalgui. A formal approach for the development of reactive systems. *Information and Software Technology*, 53:14--33, 01 2011.

[ML82] Per Martin-Löf. Constructive mathematics and computer programming. *Logic, Methodology and Philosophy of Science*, 104, 12 1982.

[ML98] Per Martin-Löf. *An intuitionistic theory of types.* 10 1998.

[MP90] Zohar Manna and Amir Pnueli. A hierarchy of temporal properties (invited paper, 1989). In *PODC '90*, 1990.

[Orb99] Mars Orbiter. Mars climate orbiter mishap investigation report. 12 1999.

[OVW98] Michael O'Sullivan, Siegfried Voessner, and Joachim Wegener. Testing temporal correctness of real-time systems-a new approach using genetic algorithms and cluster analysis. 01 1998.

[PN93] Lena Pareto and Bengt Nordström. The alf proof editor and its proof engine. volume 806, pages 213--237, 05 1993.

[Poh89] Wolfram Pohlers. *Proof theory : an introduction / Wolfram Pohlers*, volume 1407. 01 1989.

[Pol96] Randy Pollack. The theory of lego - a proof checker for the extended calculus of constructions. 09 1996.

[Ray10] Pascal Raymond. *Synchronous Program Verification with Lustre/Lesar*, pages 171 -- 206. 01 2010.

[Rus03] Bertrand Russell. *The Principles of Mathematics.* 02 1903.

[Sch96] David A. Schmidt. Programming language semantics. *ACM Comput. Surv.*, 28:265--267, 1996.

[Sel14] Jonathan Seldin. de bruijn n. g.. lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. koninklyke nederlandse akademie van wetenschappen, proceedings, ser. a vol. 75 (1972), pp. 381–392; also indagationes mathematicae, vol. 34 (1972), pp. 381–392. *The Journal of Symbolic Logic*, 40:470, 09 2014.

[Ser11] Thierry Le Sergent. Scade: A comprehensive framework for critical system and software engineering. In *SDL Forum*, 2011.

[Tur49] A.M Turing. Checking a large routine. *Report of a Conference on High Speed Automatic Calculating Machines*, pages 70--72, 06 1949.