CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Proof Checker for Extended Linear Time Temporal Logic Proofs About Small Concurrent Programs

Master's Thesis in Computer Science – Algorithms, Languages and Logic

JOHAN HÄGGSTRÖM

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2018

Master's thesis 2018

Proof Checker for Extended Linear Time Temporal Logic Proofs About Small Concurrent Programs

Johan Häggström

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2018

Supervisor: K. V. S. Prasad
Examiner: Thierry Coquand

Master's Thesis 2018
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2018

Abstract

Program verification is a time-consuming task and prone to errors when done manually. Verification tools are therefore essential when dealing with verification in larger scales. As of now, most verification tools use model checking when verifying program properties. Model checkers search for contradictions to properties regarding those programs, and if none are found then the property is considered valid. However, most model checkers are made for sequential programs, and with most modern environments using concurrency, the demands on the verification tools increase accordingly.

With the success of model checking, formal proofs regarding concurrent programs have gotten little attention the past years. Conducting formal proofs can be tedious and error prone when done manually, but can also be very useful in terms certainty and gaining a more intuitive understanding of the problems.

This thesis focuses on the development of a tool for formal proof checking of small concurrent programs. The tool was developed using the functional programming language Agda. A logic was implemented, along with a representation of a small programming language and a proof construction system.

The final result of the project is a proof checking tool able to verify liveness proofs regarding arbitrary programs of the specified language. The proofs are conducted using predefined logic rules. If the proof can be implemented in the proof checker, the proof is considered valid.

Agda turned out to be a useful tool for conducting formal proofs. The method of formal proofs for concurrent programs is still not preferable due to the complexity of the proofs, but with the development of more sophisticated automated theorem provers, the method may become increasingly viable in the future.

Acknowledgements

First, I would like to thank my supervisor at Chalmers University of Technology, K. V. S. Prasad, for for the continuous support and expertise during the entire project.

Next I want to thank all members of the bachelor's group in collaboration with this thesis, Andreas Standár, Erik Bergsten, Oskar Larsson, Oskar Rutqvist, Tobias Rastemo, and Carl Söderpalm. It was a pleasure working with them and the project would not have been the same without them.

I also want to thank Nachiappan "Nachi" Valliappan for his contributions and interesting discussions during the research stage of the project.

Finally, I would like to thank my examiner at Chalmers University of Technology, Thierry Coquand, for taking his time to provide help and feedback.

Johan Häggström, Gothenburg, 2018

Contents

1
Introduction

This thesis was performed at Chalmers University of technology, partly in collaboration with a Bachelor's thesis group. The purpose of this thesis is to develop a proof-checker tool regarding proofs for concurrent programs. The properties of these programs are represented using an extension of Linear Temporal Logic. The goal involves constructing such a tool, as well as evaluating the proof methods and potential future usage of similar systems.

1.1 Background

Verification of programs is a time-consuming task and prone to errors when done manually. Verification tools are therefore essential when dealing with verification in larger scales. This process is tedious enough when dealing with sequential programs, and having concurrency in programs further increases the complexity and the demands on such verification tools. The established verification tools are mostly limited to sequential programs [1, 2, 3], and most research on verification have been conducted in this area. Concurrency in programs introduces a new level of complexity, making previous verification tools rather obsolete in comparison to the demands of today. Therefore more sophisticated verification tools are required to handle this problem.
A concurrent environment differs in many aspects from sequential ones. A typical contemporary program communicates interactively with its environment, and has many different subprocesses working concurrently. This is especially true for embedded systems, where the concurrent behavior is essential; taking up a very large proportion of the current code base [4]. In this case, the importance of reliable concurrency is not just a matter of convenient interface, but rather the entire performance of the systems. It is therefore crucial to further develop proof-checking tools able to handle concurrent systems.
There are major problems that affect all system designs, first to express what the system is supposed to do, then what the system actually does, and finally to show that the two previous statements are satisfactorily related. Furthermore, it is important to show the completeness and consistency requirements of the system. We can and should also show the consistency and completeness of the requirements we make of the system. Operational semantics and linear temporal logic (LTL) [5] have shown themselves over several decades to be suitable analytic tools for concurrent systems [6, 7]. There exist established tools for model checking that can verify whether a property holds for system, or show an execution path leading to

failure for a property. These tools are very useful since they can show why a property fails. They cannot however show why a property holds, which usually is a far more complex task. Showing why properties hold is best done by formal proofs. The issue with formal proofs is that they often require long and detailed explanations, even when proving trivial properties. Proof checking this way is very tedious and time-consuming, and errors are likely to occur, even for smaller and simpler proofs. Automatic proof checkers are therefore needed in order to prove properties in a reasonable time span. Unfortunately, most proof checkers are built for sequential programs, since sequential programs makes the process of proof checking significantly simpler. The need for having reliable, concurrent programs is however increasing, which requires more complex proof checkers able to handle concurrent programs. There has been some research on proof checking in the past. A paper by Andersen, Harcourt and Prasad [8] proved a complete, data-dependent concurrent program correct. There has however been very little follow-up on this subject.
Concurrency in programming adds a level of complexity, by introducing nondeterminism. Because the program may take different computing paths depending on random events, the traditional tools are not sufficient for this task. Instead, a tool for evaluating all possible outcomes at once is needed. In order to verify program properties, a formal way of expressing properties is needed. LTL is suitable for program verification, since the time dimension can easily be related to program execution flow. The suggested tool must be able to verify LTL-formulae, extracted from the property specifications of concurrent programs.

1.2 Context

As mentioned earlier, there has not been an extensive amount of studies in the area of proof checking for concurrent programs. Previous studies of mechanical verification have however been conducted. Goldschlag [9] described a proof system for mechanically verifying concurrent programs, using the Boyer-Moore prover. This paper shows this is possible using Boyer-Moore logic, which is not quite the same as the logic framework that used for this thesis. However, it shows that extensive work has been done in the area of proof checking.
Most relevant papers are quite old, but a recent paper from Sandip Ray and Rob Sumners [10] shows promising results. They show that formal proofs can be handled mechanically, by showing that the executions of the programs can be viewed as executions of simpler systems, thus reducing the complexity of the proofs. This is quite close to the subject of this thesis, which is a solid indication of potential to the study.

Another method of reasoning about program execution is presented in the paper from Lamport [11]. The paper describes a formal specification language called TLA+ and was developed in order to model and verify behavior of concurrent systems. A proof checking tool called TLAPS [12] was developed for proofs written in TLA+. The proof-checking method involves translating program code into a purely logical language. The supplied proof is then proof checked mechanically using the resulting logic formulae of the translation. The advantages of this black-box style proof checker lie within the proof checker being rather independent from the program representation. However, it can also be considered a disadvantage due to the distancing from the original problem, reducing the intuitive understanding of the proofs.

Looking back in time, a paper from Owicki and Lamport [13] presents a method for proving properties of concurrent programs. The goal of the paper was to provide a method for programmers to formalize logical reasoning mechanically, this while still keeping references to program statements in the proofs. This method more accurately aligns with the goal of this thesis, and the provided semantics easier to implement. Large portions of the reasoning and methods presented in this thesis are therefore inspired by that paper.

1.3 Goals and Research Questions

The goal of the thesis is to develop and evaluate a tool able to prove properties of small concurrent programs. The final product should be a suitable complement to the existing model checkers. A model checker says a lot about why a program fails to uphold a relation or invariant. This tool however does the opposite, by instead commenting about why a statement succeeds. As of now, students have no way of checking the validity of their LTL-proofs, except for manually checking themselves. This process is tedious, time-consuming and error prone, even for smaller proofs, showing why such a proof checker is needed.
In order to construct such a tool, a logic has to be specified to represent properties of the concurrent programs. The logic has to be represented in a general logical framework in order to prove given properties. The framework chosen for this thesis is the dependently typed programming language Agda [14]. Agda is a strict language with extensive support for mathematical notations, making it suitable for this task. A large part of the goal is to simply explore whether this method of verification will be sufficient, and if it is worth further development.
Since dealing with concurrency is a complex matter, building a tool able to handle more complex concurrent programs can be challenging. Although the goal is to make the tool as sophisticated as possible, it is wise to limit the test cases to small concurrent programs. The final product can then hopefully be developed further to handle larger and more complex programs with relative ease. It would be preferred to eventually extend the tool, including user friendly features to make it into a proof assistant. This might not however be possible during the time span of this project and will only be included if the time allows it.

An interesting aspect to consider arises when formalizing proof systems, especially when based upon previous work. Even if a system appears to be rigid and bullet-proof, minor errors or assumptions can be easily overlooked. Some might be minor nuisances that can easily be adjusted, while others shake the foundation of the theory it is built upon. Assumptions made for easier and more fluent theoretical reasoning may prove too vague when implementing in a logical framework. The implication of these details becomes apparent when introduced to the strictness of Agda. It is therefore of importance to bridge the gap between paper and the digital world.

Research questions to be answered in this thesis:

- Can a tool suitable for students be developed to aid in conducting logical proofs about concurrent programs?
- Are previously established proof methods for concurrent programs complete and solid enough to formalize in Agda?
- Is proof checking a suitable addition or alternative to model checking?
- Is Agda expressive enough for representing the undecidability of concurrency, while still maintaining strictness for the proofs?

1.4 Methodology

The initial part of the project involved studies of previous work in order to determine the appropriate approach. The main goal of the initial study involved finding similar work related to this study. In order to eventually formalize the work of the studies, additional knowledge of the programming language Agda had to be acquired. A learning period of Agda was therefore initialized in collaboration with the previously mentioned bachelor's group.

Following the study, a small concurrent programming language (CPL) was specified. The CPL was used during the entire project and all the reasoning is built solely upon this language. In order to reason about this CPL, a logic was defined based upon Linear Temporal Logic (LTL) [5]. This extension of LTL is referred to as Extended LTL (ELTL), and is extended with operators regarding statements of the CPL.

Along with the CPL and the logic, a satisfaction relation had to be defined between the two. The satisfaction relation is meant to provide rules of inference for proofs of the desired type. Using these rules, proof trees regarding these programs can be constructed by manipulating the logic and its operators.

The programming language *Agda* was chosen as the general logical framework, where the definitions were formalized. Previous studies show formal methods for reasoning about concurrency in programs. These methods are however not completely formalized, which was a major challenge of the project. Using Agda, all the theory from the literature studies had to be formalized and implemented in this framework.

Finally, by combining all the implemented components, the final product results in a tool able to prove ELTL properties about programs of the defined CPL. During the final phase of the project, various proofs had to be conducted in this system, as well as showing properties that cannot be proved for certain programs.

1.5 Ethical Aspects

The ethical aspects of this project are not obvious and rather limited due to not involving any test subjects or any collected data that might be sensitive. Since the testing and verification will be done using local test programs, there is no conflict regarding ownership. The potential benefits from this study could not possibly be used to cause harm of any sort, since the results of the study can only be used to prove properties of programs.

The aim of this study is to further increase the selection of verification tools available to students and potentially developers. It might not be obvious why verification is so important and how it affects people's everyday lives. However, people today rely on automated systems every day, especially in traffic and production. Most of us live our everyday lives without reflecting too much on the possible implications should something go wrong in this regard. In most cases the effects of faulty systems might not cause much harm or even be noticed at all. Malfunctioning systems could however cause tremendous harm with the potential of lethal implications. A good example of this would be automated vehicles, now close to being introduced into the society. The passengers of such vehicles put their safety in the hands of the software developers to ensure the correct behavior of the vehicles. There exists therefore an underlying responsibility for all programmers dealing with these problems to ensure that their products ensures the safety of the passengers. Having solid and flexible verification tools is therefore not only a convince, but may also save lives in the future.

2
Concurrent Programming Language

In order to reason about or prove anything about the execution of a program, the programming language in which it is composed has to be clearly defined, both in regards to functionality and its limitations. Therefore, a basic CPL suitable for the goals of this thesis has been defined and is presented in this chapter. The proofs presented in this thesis are conducted based solely on this language.

2.1 Language Definition

The CPL defined in this thesis covers the most fundamental operations required to model simple concurrency. Despite being a rather minimalistic language, many programs can be translated into this language without losing functionality. This translation is made possible due to most advanced data structures and data types can be simplified into their basic subparts. Having more complex data types does not necessarily affect functionality, but are used mostly to enhance program structure, making it more accessible to programmers. It is beneficial to keep the set of available operators to a minimum when reasoning about potential outcome of program execution. An example of an abundant operator would be the addition of *goto* statements to the CPL. In addition to being abundant, if program control is allowed to be moved to an arbitrary statement, then a large portion of the inference rules could be affected and no longer valid.
The language resembles imperative languages and is presented in a pseudo-code style. Just like most imperative languages, data types are supported to be used both for variables and regular expressions. The two supported data types are natural numbers and booleans; increasing the complexity of the system by adding additional data types would not make the proofs any more rewarding. Let $\mathbb{N}^*$ denote the set of natural number expressions, and $\mathbb{B}^*$ the boolean expressions. The program structure consists of statements, each labeled to enable referencing of the statements when conducting proofs about the program.

The following list shows all allowed statements of the programming language:

1. $x := n$, x is assigned the value of n, x variable of $\mathbb{N}^*$, $i \in \mathbb{N}^*$.
2. $x := b \mid x$ is assigned the boolean value of b, x variable of $\mathbb{B}^*$, $b \in \mathbb{B}^*$.
3. s_1 ; s_2 ; ... ; s_i , a block of i sequential statements.
4. if x then s, where $x \in \mathbb{B}^*$, s a statement.
5. while x then s, where $x \in \mathbb{B}^*$, s a statement.
6. cobegin p_1 || p_2 coend, spawns two statements p_1 and p_2, representing two processes running in parallel.

2.2 Program Structure

A program in this programming language is defined by a single main process. This process decides the entry point of the program, where the first statement in the statement block will always be the first to execute. A process is allowed to spawn an arbitrarily amount of new processes. With regards to complexity reduction, each process is only allowed to be executed once, meaning if control would be assigned to the spawning statement more than once, control is simply moved to the following statement and no new processes are spawned.
Each statement in this language belongs to a parent process, and cannot exist independently. Just like imperative languages, after executing a statement, program control is moved to next statement. The program in figure 2.1 shows an example of a valid program in this language.

Code 2.1: Example of a program in the defined CPL.

```
s0: {
  s1: p := true
  s2: q := false
  s3: cobegin
    s4: {
      s5: p := false
      s6: while ~ q then
        s7: x := 5
    }
    ||
    s8:: {
      s9: while p then
        s10: if true then
          s11: x := 6
        s12: q := true
    }
  coend
  s7: x = 1
}
```

2.3 Limitations

Limitations have been introduced to the CPL in order to simplify the proof rules of the proof checker. Once again, these limitations do not prevent functionality noticeably.

All assignment statements of the CPL are considered atomic. The atomicity is introduced to prevent unnecessary complexity. Otherwise, assignments would be overly complicated for the cause of this project. If atomicity were not assumed, then in order to reason about the eventual value of x for a statement $x := v$, an additional proof of safety would be required. If any other statement could assign to x, then no conclusions could be drawn from this assignment, since the possibility of race conditions is introduced.

Another slight limitation of the system involves the spawning of processes. Consider statement s_3 in figure 2.1. Unlike most other languages with support for concurrency, the program control is not moved from a statement after the spawning of new processes. Instead, program control for s_3 waits until both s_4 and s_8 have terminated. Having this property of the language enables easier reasoning about program flow, without removing any significant functionality of the language.
As mentioned earlier, there is no way to "jump" in the program, meaning there are no *goto* statements allowed, and no function calls or similar methods can be invoked.

Some fairness is assumed for the CPL, meaning if a statement is waiting for execution priority, it will eventually have priority. If fairness were not assumed, control could get stuck at a statement forever, since priority would not be guaranteed at some point. Not having fairness to at least some degree would make proofs about termination impossible when dealing with concurrency.

3

Extended LTL

Temporal logic has been proven a great tool when reasoning about program execution. The time dimension provides both rules and a set of symbols for expressing program properties, as well as enabling reasoning about possible program execution. Linear temporal logic (LTL) is a temporal logic referring to time as future paths. Possible program execution can be seen as the set of all future paths of the program, making LTL suitable for dealing with formal proofs about concurrent programs. This chapter briefly presents the fundamentals of linear temporal logic, followed by the definition of the extension to the logic. The purpose of the extension is to enable more intuitive program referencing in the logic. Finally a satisfaction relation between the logic and the CPL defined in section 2 is presented.

3.1 Linear Temporal Logic

When reasoning about program execution, the liveness and safety properties become interesting. The liveness property of concurrent programs states that eventually progress is made, despite the possibility of having multiple processes executing simultaneously. This kind of property can be expressed using the "eventually"-operator of LTL, and can be used for conducting termination proofs.

The safety property states informally that "nothing bad will happen". More formally, a state where errors would occur cannot be reached from any legal program state. Safety can be expressed using the "always"-operator, stating that a property will always hold for each program state during an execution. The two operators $\Box$ and $\Diamond$ can then be defined as:

$$\Diamond\ \phi : \phi \text{ holds now or in the future} \tag{3.1}$$

$$\Box\ \phi : \phi \text{ always holds.} \tag{3.2}$$

In theory, all program properties can be represented using the propositions and operators from LTL. This would however lead to a very abstract and unintuitive reasoning about the programs. Therefore an extension of the logic is defined in order to make the reasoning about proofs more convenient.

3.2 Extended LTL

In order to express program execution for specific programs in an intuitive and easy-to-follow manner, LTL requires extension with additional operators. This extended logic is referred to as Extended Linear Temporal Logic (ELTL). LTL is a subset of ELTL, thus all operators of LTL are available to ELTL. This extended logic contains additional operators for expressing necessary properties about program execution. The logic is extended with operators for dealing with variables and their potential values, and program pointers referencing the execution state of the processes. The set of additional operators regarding expressions and program pointers is the following:

Let s be a program statement. Then

$$\begin{aligned}\text{at } s &: \text{ Process control at } s\\ \text{in } s &: \text{ Process control in } s\\ \text{after } s &: \text{ Process control after } s\\ b^* &: \text{ Any boolean expression } b^*,\ b^* \in \mathbb{B}^*\end{aligned}$$

ELTL manages program pointers using the operators *at*, *in*, and *after*. Since all program statements are labeled, these operators can be directly associated with program statements. It is however important to note that even though the logic uses program references as pointers, the programs are never executed when conducting the proofs. The pointers simply enables reasoning about the implication would a statement execute. This style of reasoning about program behavior is heavily inspired by the work of Owicki and Lamport [13].

Extending LTL with additional operators referencing program statements impacts the properties of LTL. Formulae of LTL are decidable [7], meaning for every LTL-formula, there exists an algorithm for determining its validity. With ELTL being more expressive, formulae can be expressed that are not decidable, an example being the halting problem. With decidability being present, one could definitely argue for the benefits of interactive theorem proving for ELTL.

3.3 Satisfaction Relation

The set of rules for ELTL plays an important role in the proof systems of this thesis, by introducing a satisfaction relation between programs and logic formulae. Let $P \vDash \phi$ denote that a program P satisfies the ELTL-formula ϕ. A rule in this system makes use of proved properties of P, but also references to statements of P. Each rule is defined as a set of preconditions, and a postcondition. If all preconditions are met, then the postcondition can be concluded. New properties about a program can be deduced by applying rules to proved properties of the program.

Note that $\models$ refers to entire programs and not subparts. Usually when dealing with temporal logic, a satisfaction relation is defined between a sequence of states and a property. If sequences were used to reason about a set of statements of a program, it would be very difficult to determine the implications for the entire program. Therefore the relation is defined over entire programs, enabling proving new properties by using previously proved ones.

3.3.1 Inference Rules

In this section, all inference rules, in the form of axioms and theorems, used in this thesis are presented, with the exception of the inherited rules from LTL. Note that for all inference rules, $P \vDash$ is omitted for convenience.

Program flow is defined by introducing a new operator, ~>. The formula $\phi \vDash \psi$ reads "ϕ leads to ψ". More formally, $\Box(\phi \implies \Diamond\psi)$, where ϕ and ψ are ELTL-formulae. Most of the following inference rules use this operator to reason about program execution flow.

$$\frac{\phi \sim> \psi \qquad \psi \sim> \chi}{\phi \sim> \chi} \sim>\text{-trans} \tag{3.3}$$

Transitivity property of ~> (3.3), if ϕ leads to ψ, and ψ leads to χ, then ϕ leads to χ.

$$\frac{l : x := v}{\mathit{at}\ l \sim> \mathit{after}\ l} :=\text{n-step} \tag{3.4}$$

Theorem 3.4, control will eventually proceed an assignment of type $\mathbb{N}^*$.

$$\frac{s : x := v}{\mathit{at}\ s \sim> \mathit{after}\ s \wedge s == v} \text{Natural Assignment} \tag{3.5}$$

3.5, if statement s is executed, control will be after s and the value v is assigned to the variable x.

$$\frac{l : \text{cobegin } p_1 \mid\mid p_2 \text{ coend}}{\mathit{at}\ l \sim> \mathit{at}\ p_1 \wedge \mathit{at}\ p_2} \text{Enter Cobegin} \tag{3.6}$$

Axiom 3.6, execution of cobegin-statement leads to control at both spawned processes.

$$\frac{s_i \ ; \ s_j}{\mathit{after}\ s_i \sim> \mathit{at}\ s_j} \text{Control Flow} \tag{3.7}$$

Axiom 3.7, control flow of a sequence of s_i and s_j in a block.

$$\frac{s : p_1 \mid\mid p_2 \qquad \textit{at } p_1 \sim> \textit{after } p_1 \qquad \textit{at } p_2 \sim> \textit{after } p_2}{\textit{at } s \sim> \textit{after } p_1 \wedge \textit{after } p_2} \text{Join} \tag{3.8}$$

Axiom 3.15, separate termination of two concurrent processes leads to both having terminated at some point.

$$\frac{s : p_1 \mid\mid p_2}{\textit{after } p_1 \wedge \textit{after } p_2 \sim> \textit{after } s} \text{Coend Inf} \tag{3.9}$$

Axiom 3.16, termination of two concurrent processes leads to termination of the cobegin statement that spawned them.

$$\frac{\phi \sim> \psi \qquad \Diamond\, \phi}{\Diamond\, \psi} \sim>\text{-e} \tag{3.10}$$

Axiom 3.10, if ϕ leads to ψ, and ϕ eventually holds, then ψ eventually holds.

$$\frac{\phi}{\Diamond\, \phi} \Diamond\text{-i} \tag{3.11}$$

Axiom 3.11, if the immediate assertion of ϕ holds, ϕ eventually holds.

$$\frac{\textit{Program} :\ \ s}{\Diamond\ \textit{at } s} \text{init} \tag{3.12}$$

Axiom 3.12, if s is the main statement of a program, control will eventually be *at* s.

$$\frac{s \text{ first in } b}{\textit{at } b \sim> \textit{at } s} \text{Enter Block} \tag{3.13}$$

3.13, if statement s is the first statement in the block b, then b leads to s.

$$\frac{s : \text{cobegin } p_1 \mid\mid p_2 \text{ coend}}{at\ s \leadsto at\ p_1 \wedge at\ p_2}\ \text{Enter Cobegin} \tag{3.14}$$

3.14, if control at cobegin statement, enter parallel processes.

$$\frac{s : \text{cobegin } p_1 \mid\mid p_2 \text{ coend} \qquad at\ p_1 \leadsto after\ p_1 \qquad at\ p_2 \leadsto after\ p_2}{at\ s \leadsto after\ p_1 \wedge after\ p_2}\ \text{Join} \tag{3.15}$$

3.15, if both parallel processes terminate, both processes will eventually be terminated at the same time.

$$\frac{s : p_1 \mid\mid p_2}{after\ p_1 \wedge after\ p_2 \leadsto after\ s}\ \text{Coend Inf} \tag{3.16}$$

3.16, if both parallel processes are terminated, exit statement that spawned them.

$$\frac{s : x := v}{at\ s \leadsto after\ s}\ \text{:=n-step} \tag{3.17}$$

3.17, control will always proceed past natural assignment.

$$\frac{s : \text{While } b\ s_0 \qquad at\ s \leadsto \Box\ \text{b} \qquad at\ s_0 \leadsto at\ s}{at\ s \leadsto after\ s}\ \text{Exit While} \tag{3.18}$$

3.18, if the condition of a while-loop is eventually never met, and control always returns from the inner statement, control will proceed past the loop.

$$\frac{s : \text{Block} \qquad s_0 : \text{Stm} \qquad s_0 \in s}{at\ s_0 \leadsto at\ s}\ \text{Infer Block} \tag{3.19}$$

3.19, if a statement of a block is executed, the head of its block was executed at some point.

$$\frac{s\ :\ s_1 \ || \ s_2}{\mathit{at}\ s_1 \ \text{~>}\ \mathit{at}\ s_2}\ \text{Infer Par2} \tag{3.20}$$

3.20, for any two parallel processes, if control is at the first process then the second one will eventually execute.

$$\frac{s\ :\ \text{While false}\ s_0}{\mathit{at}\ s \ \text{~>}\ \mathit{after}\ s}\ \text{Skip While} \tag{3.21}$$

3.21, control will always proceed past while-false statements.

$$\frac{s\ :\ b := \text{true}}{\mathit{at}\ s \ \text{~>}\ b}\ \text{Bool assign true} \tag{3.22}$$

3.22, if a boolean variable is assigned true, its immediate assertion is true.

4
Agda Introduction

The chosen logical framework of this project is the programming language Agda, which is a dependently typed functional programming language. Agda was developed with the intention of being a proof assistant in a functional-style environment. The language and the code style strongly resembles that of Haskell, using pattern matching and functions to construct expressions. Agda is however, compared to Haskell, a very strict language. The strictness implies that all cases need to be covered when pattern matching and no infinite patterns for functions are allowed. It is therefore not possible to prove arbitrary properties by letting functions call themselves, or excluding relevant cases during function construction. This makes a proof of an Agda property very trustworthy since it is not possible to trick the system; every possible outcome must be covered. Agda is therefore a very useful tool when dealing with formal proofs.
This section gives a brief introduction to the Agda concept, as well as showing how to conduct property proofs about simple data types in this system. It is meant as an introduction for those inexperienced in functional programming, but also as a way to introduce the implementation methods and motivate why Agda is used for this thesis. Without at least some knowledge of the basic concepts of Agda, the reasoning and the conclusions of the thesis will not be easy to follow. However basic understanding of programming will be assumed to keep this section at a reasonable level. The reader may skip this introduction if they are already familiar with similar programming languages.

4.1 Data Types

Data types are the most fundamental parts of Agda. A data type consists of constructors, each constructor representing a possible way to build an instance of that data type. Let us start by implementing one of the most basic data types in Agda, the boolean type. For the boolean type, the only two possible constructors are *true* and *false*. This is represented in Agda as:

Code 4.1: Representation of the boolean data type in Agda.

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

Data types can also be used in their own constructors. The data type for the natural numbers ℕ can be represented as:

Code 4.2: Representation of the natural number data type.

```
data ℕ : Set where
  zero : N
  suc  : (n : ℕ) -> ℕ
```

By definition, *zero* is a natural number, and every successor of n, *suc* n, is also a natural number.

Data types can also be indexed using other types. Consider the data type *Even* n, representing even numbers. The type is indexed over n, where n is an even number, meaning the type itself takes a natural number as a parameter. The successful construction of an instance of *Even* means the indexed number is in fact even. The type contains two constructors, or axioms for the type, where zero is even by definition, and for every $n \in \mathbb{N}$, $n + 2$ is also even. This can be represented as:

Code 4.3: Definition of Even

```
data Even : ℕ -> Set where
  evenZero  : Even zero
  evenStep  : {n : ℕ} -> Even n -> Even (suc (suc (n)))
```

The same approach can be applied for the odd numbers, figure 4.4

Code 4.4: Definition of Odd

```
data Odd : ℕ -> Set where
  oddOne  : Even (suc zero)
  oddStep : {n : ℕ} -> Odd n -> Odd (suc (suc (n)))
```

Note the brackets around the first argument to *evenDef*, 4.3, and *OddDef*. The use of brackets around n implies that the argument is implicit, meaning the parameter is not passed by itself to the constructor, but used in other parameters.

4.2 Functions

The functions in Agda behave more or less like functions of other functional languages. The difference lies in the strictness, since all possible patterns have to be considered when defining functions.

Simple functions can be defined over the set ℕ. For instance the *add* function can be defined as in 4.5.

Code 4.5: Definition of the add function

```
_+_ : ℕ -> ℕ -> ℕ
zero     + n = n
(suc m) + n = suc (m + n)
```

4.3 Basic Proofs

Constructors can be considered axioms of data types, thus not requiring proving. From these axioms, theorems in the form of functions can be derived to prove properties about these types.

Consider the property stating that the addition of two odd natural numbers always results an even sum. Let a function f represent this property. The definition of f is shown in figure 4.6.

Code 4.6: Definition of the function f

```
-- Proof definition of the sum of two odd numbers is even
f : {m n : ℕ} -> Odd m -> Odd n -> Even (m + n)
```

The function definition states that given two odd numbers, m and n, the sum $m+n$ sum must be an even number. A successful construction of this function proves the hypothesis correct. By using pattern matching and recursion, each possible input case can be covered. The implementation of f is presented in figure 4.7

Code 4.7: Implementation of f

```
-- Implementation of f using pattern matching
f : {m n : ℕ} -> Odd m -> Odd n -> Even (m + n)
f  oddOne       oddOne      = evenStep evenZero
f (oddStep x)   oddOne      = evenStep (o+o=>e x oddOne)
f  oddOne      (oddStep y)  = evenStep (o+o=>e oddOne y)
f (oddStep x)  (oddStep y)  = evenStep (o+o=>e x (oddStep y))
```

5
Proofs about the CPL

This chapter presents the theory behind conducting proofs about the CPL defined in section 2, and the limitations required to make those claims. Here, the method of constructing proofs is defined, as well as the types of properties that can be proved using this system. This chapter motivates the implementation methods presented in chapter 6.

5.1 Formalizing Informal Proofs

When dealing with formal proofs, it is important to keep hand-waving to a minimum. This is especially true if the proofs are to be formalized in a logical framework such as Agda. Even when a method is throughly formal and consistent, details considered trivial are often omitted due to convenience. However, the exclusion of details may cause issues regarding the implementation of the methods. When formalizing methods that has only been conducted on paper, the mistakes and missing parts are revealed in a way that is not so obvious when studying them. It can be quite a journey from handwritten proofs to formalization in logical frameworks. In the 19th century, Hilbert formalized Euclidean geometry into axioms in his paper on the fundamentals of geometry [15]. His work was later further formalized by the finish mathematician Von Plato [16] [17], where Plato worked towards an actual computer implementation of the logic. This thesis plays a similar role, although not on a similar scale, but is instead mainly built upon the work of Lamport [13].

5.2 Liveness Proofs

Liveness properties are essential when reasoning about program execution, especially when dealing with concurrent programs. A liveness property uses the $\Diamond$-operator to assert that a specified property eventually holds. When dealing with temporal logic, $\Diamond\ \phi$ means ϕ will be true for the immediate assertion or sometime in the future, regardless of chosen path. This means that the validity of $\Diamond\ \phi$ varies depending on the current state of the program. However, for the proof checker system presented in this thesis, reasoning about execution paths this way is not allowed, since the satisfaction relation refers to the entire program.

The presented method for proving liveness properties involves applying inference rules to previously proven properties of a program. The inference rules with references to program statements, use the operator ~>, defined in section 3.3.1. By composition of previously inferred properties, using the transitivity property of ~>, liveness proofs can be conducted. Depending on the goal of the proof and the type of statement is presented, different rules of inference will be used to advance towards the goal. A liveness proof consists of a tree structure, where each branch consists of the application of rules on previous proven properties. If a proof tree can constructed, where the final property is the goal, then the program satisfies the desired property.

5.3 Safety Proofs

Safety proofs about a concurrent program can be defined as proving properties that hold for the entire execution of the program. Informally, "nothing bad will ever happen". The formal approach would be to define a satisfaction relation between a program p and a safety property $\phi \Rightarrow \Box\ \psi$. The relation between the program and the property is defined as: $p \vDash \phi \Rightarrow \Box\ \psi$. This definition translates into: starting in a state where ϕ holds, ψ will hold for the entire execution. For simplicity, assume ϕ is always *at* s_0, where s_0 is the starting point of the program. Let P be a program, and S the set of all statements of P. Define a proposition $R(s)$, that holds iff a statement s is reachable, and V(s) that holds if s violates the safety property. In order to prove the safety property, the property in equation 5.1 is required.

$$\forall s \in S; \neg R(s) \vee \neg V(s); \tag{5.1}$$

The problematic part of 5.1 occurs when defining the proposition $R(s)$. The introduction of the set of reachable states is often considered trivial and therefore not well defined. When dealing with trivial properties, one might, for simplicity, look at the program definition and assume the set of reachable states. This method gets out of hand when programs become more complex, especially with concurrent processes, where each process relies heavily upon the other. Therefore, a formal method for defining $R(s)$ has to be developed in order to formally prove safety properties.

6 Implementation

In this chapter, a summary of the key components of the Agda implementation of the proof checker is presented. The repository for the project can be found at `https://github.com/JohanHagg/Proof-Checker`.

6.1 Approaches

During the planning phase, multiple approaches for building a proof checker were tested and evaluated. Eventually, two methods were decided upon to get further development. The first method is referred to as the proof by control flow. The method allows for conducting faulty proofs, and will instead give feedback depending on the success of the proof. The advantage of this method lies in the program representation, which is more intuitive and complete. However, proof construction is tedious and difficult to get right. The method was developed in cooperation with a bachelor's group and the implementation is described in their thesis [18] and will not be covered here.

The second method for proof checking is referred to as the proof by construction, and is considered the main result of the thesis. It relies heavily on the strictness of Agda, making for solid and quite intuitive proofs. The following sections describe the fundamental components of the proof checker.

6.2 Program Representation

The implementation of the CPL closely follows the definitions from chapter 2. The CPL module contains a program definition, where the program contains a reference to the entry point of the program. The entry point is defined by a single statement, working as the main process of the program.
The definition of statements is straightforward and presented in figure 6.1. The data type Stm^* is indexed over the label of a statement, thus representing a complete statement. Since statements are labeled, they can be used in inference rules for the programs.

Code 6.1: Statement definition in Agda

```

-- Block of statements, l the label of the statement.
data Block : Set where
  first : (l : String)          -> Block
  _::_  : Block -> (l : String) -> Block

-- Program statement representation.
data Stm : Set where
  _:=n_ : (x : String)  -> (n : ℕ*)      -> Stm
  _:=b_ : (x : String)  -> (b : Bool*)   -> Stm
  block : (b : Block)                    -> Stm
  _||_  : (s₁ : String) -> (s₂ : String) -> Stm
  if    : (b : Bool*)   -> (l : String)  -> Stm
  while : (b : Bool*)   -> (l : String)  -> Stm
  swapN : (x : String)  -> (y : String)  -> Stm
  swapB : (x : String)  -> (y : String)  -> Stm

-- Labeled statements
data Stm* : String -> Stm -> Set where
  stm  : (l : String) -> (st : Stm) -> Stm* l st
```

Constructing a program in this system requires defining all statements independently. The block structure enables the sequencing of statements. The statements are glued together using the initial statement of the program.

6.2.1 Satisfaction Relation

The satisfaction relation between the programs and the logic has already been mentioned earlier in section 3.3. The implementation strongly follows the previous definition of satisfaction, using the notation of $P \vDash \phi$, program P satisfies ϕ. A data type for this satisfaction relation, _⊨_, is implemented using a set of rules as constructors. The application of rules on known properties of a program helps advancing proofs by taking steps. In order to apply a rule to a satisfaction, all preconditions of the rule has to be met. The amount and type of preconditions varies among different rules. In addition to this, the set of rules are divided into two subcategories:

1. LTL rules, regular rules used to manipulate ELTL formulae.
2. Program rules, requires referencing the program as preconditions.

The LTL rules are used to modify ELTL formulae into the desired form. The data type $\vdash$ is defined as $\phi \vdash \psi$, meaning ψ is a logical consequence of ϕ, where ϕ and ψ are ELTL formulae. A relation between $\vdash$ and $\vDash$ is defined as: if for a program P, $P \vDash \phi$, and $\phi \vdash \psi$, then $P \vDash \psi$. Note that these rules require no program reference, other than the original program itself.

The program rules define the relation between the program statements and the logic. The rules follow a similar pattern as the ELTL-rules, but uses program references instead of previously defined properties of programs. Program rules require statements of the correct form in order to be legally applied to a satisfaction relation. The figure shows the program flow rule for assignment to integer variables, stating that given an assignment statement, $P \vDash$ *at l* ~> *after* l. This entailment holds since there is no condition preventing the program from moving forward when executing an assignment. Since the operator ~> is transitive, if $P \vDash \phi$ ~> ψ and $P \vDash \psi$ ~> χ, then $P \vDash \phi$ ~> χ. The transitivity property can therefore be used to construct a chain of proved properties, directly related to liveness proofs of these programs.

6.3 Absurdity and Termination of Proofs

A key component to the implementation of the proof system is the data type $\perp'$, which represents *absurdity*. Since non-terminating proofs is not allowed, and statements can be of arbitrary depth, the absurdity type lets you keep the flexibility of the program representation by only allowing the construction of terminating data structures. An example of a non-terminating structure could be a cyclic block of statements.

7
Results

The result of the project is a proof checker based upon the method of proof by construction. In this chapter, the method of proof construction is demonstrated, followed by example proofs. These proofs are presented using both mathematical notations and code segments implemented in Agda.

7.1 Proof Construction

Let P be a concurrent program, and ϕ an ELTL property. A proof definition of $P \vDash \phi$ in the proof checker is defined in figure 7.1.

Code 7.1: Agda proof definition of $P \vDash \phi$.

```
f : {P : Prog s0 0} -> P ⊨ ϕ
```

If the function f can be constructed, then a proof has been found and the specified property ϕ holds for the program P. Recall that the definition of $Prog$ takes two parameters, $Prog\ s_0\ n$, where s_0 is the statement representing the main process, and n the amount of assumptions made for the program. In order to chain lemmas of partial execution, all functions must be parameterized with the same program definition.

7.2 Liveness Proofs

In this section, two liveness proofs are presented. The first proof dealing with a very simple program of two rather independent processes. The second example is slightly more complex, where the execution of one process depends on the other.

7.2.1 Liveness 1

The first liveness example proves a liveness property regarding the eventual value of a variable. The desired liveness property is defined in 7.1, where P is the program in figure 7.2.

$$P \vDash \text{at } s_0 \mathrel{\sim>} x == 1 \tag{7.1}$$

Code 7.2: Simple program.

```
s₀: {
  s₁: cobegin
    s₂: x := 5
  ||
    s₃: x := 6
  coend
  s₄: x := 1
}
```

The figure shows a simple program including two concurrent processes. The Agda representation of the program is shown in figure 7.3.

Code 7.3: Agda representation of 7.2.

```
s0 = stm "s0" (block ((first "s1") :: "s4"))
s1 = stm "s1" ("s2" || "s3")
s2 = stm "s2" ("x" :=n (nat 5))
s3 = stm "s3" ("x" :=n (nat 6))
s4 = stm "s4" ("x" :=n (nat 1))
```

By inspecting the program, termination of both processes is trivial, since there is no conditional dependency between the two processes s_1 and s_2. Proof trees about liveness are constructed using the transitivity property of ~>, 7.2.

$$\frac{\phi \sim> \psi \qquad \psi \sim> \chi}{\phi \sim> \chi}\ \sim>\text{-transitivity} \tag{7.2}$$

The proof construction method involves chaining lemmas of partial execution of P, using 7.2. The following lemmas are used for proving the liveness property:

1. $at\ s_0 \sim> at\ s_1$
2. $at\ s_1 \sim> at\ s_2 \wedge at\ s_3$
3. $at\ s_2 \wedge at\ s_3 \sim> after\ s_1$
4. $after\ s_1 \sim> x == 1$

Lemma (1) is derived from the Enter Block rule, 7.3. If statement s is the first statement in the block of statements b, then b leads to s.

$$\frac{b : \text{Block statement} \qquad s \text{ head of } b}{at\ b \sim> at\ s}\ \text{Enter Block} \tag{7.3}$$

For lemma (2), the Cobegin Rule is applied to infer control at both concurrent statements, 7.4. If s is a cobegin statement of p_1 and p_2, then control at s leads to control at p_1 and p_2.

$$\frac{s : \text{cobegin } p_1 \mid\mid p_2 \text{ coend}}{\textit{at } s \sim> \textit{at } p_1 \wedge \textit{at } p_2}\text{ Enter Cobegin} \tag{7.4}$$

Lemma (3) is derived from the combination of the Join rule, 7.5, and the Coend-Inf rule, 7.6. Join states that, if the two concurrent statements terminates independently, then both will eventually terminate. Coend Inf requires that for a cobegin statement, if both concurrent statements reach termination, then the cobegin statement terminates as well.

$$\frac{s : \text{cobegin } p_1 \mid\mid p_2 \text{ coend} \qquad \textit{at } p_1 \sim> \textit{after } p_1 \qquad \textit{at } p_2 \sim> \textit{after } p_2}{\textit{at } s \sim> \textit{after } p_1 \wedge \textit{after } p_2}\text{ Join} \tag{7.5}$$

$$\frac{s : p_1 \mid\mid p_2}{\textit{after } p_1 \wedge \textit{after } p_2 \sim> \textit{after } s}\text{ Coend Inf} \tag{7.6}$$

In order to advance past s_1, both s_2 and s_3 are required to eventually terminate according to 7.5. Therefore, two separate lemmas regarding termination of s_2 and s_3 are required. In this case, the lemmas are trivial and takes only one inference rule to reach the goal. Using the theorem $\mathbb{N}^*$ Assignment Step, 7.7 , termination of both statements is derived.

$$\frac{s : x := v}{\textit{at } s \sim> \textit{after } s}\text{ :=n-step} \tag{7.7}$$

The final lemma (4) uses the assignment rule for natural numbers, 7.8, if statement s is executed, control will be after s and the value v is assigned to the variable x. The formula then requires modification to get the desired form, by eliminating *after* s.

$$\frac{s : x := v}{\textit{at } s \sim> \textit{after } s \wedge s == v}\text{ Natural Assignment} \tag{7.8}$$

Some additional rules are defined for dealing with program flow, 7.9, and managing the form of ELTL formulae, 7.10. Control Flow states that for a block of statements b, if t follows s, then *after* s ~> *at* t.

$$\frac{b:\ \text{Block statement} \qquad (s :: t) \in \text{b}}{\textit{after}\ s \sim\!>\ \textit{at}\ t}\ \text{Control Flow} \tag{7.9}$$

$$\frac{\phi \sim\!> \psi \wedge \chi}{\phi \sim\!> \chi}\ {\sim\!>}\text{-}\wedge\text{-}e_2 \tag{7.10}$$

Using all the defined lemmas, the respective proof trees can be constructed, 7.11, 7.12, 7.14, 7.15.

$$\frac{s_0}{\textit{at}\ s_0 \sim\!>\ \textit{at}\ s_1}\ 7.3 \tag{7.11}$$

$$\frac{s_1}{\textit{at}\ s_1 \sim\!>\ \textit{at}\ s_2 \wedge \textit{at}\ s_3}\ 7.4 \tag{7.12}$$

$$\frac{s_1 \qquad \dfrac{s_2}{\textit{at}\ s_2 \sim\!>\ \textit{after}\ s_2}\ 7.8 \qquad \dfrac{s_3}{\textit{at}\ s_3 \sim\!>\ \textit{after}\ s_3}\ 7.8}{\textit{at}\ s_2 \wedge \textit{at}\ s_3 \sim\!>\ \textit{after}\ s_2 \wedge \textit{after}\ s_3}\ 7.5 \tag{7.13}$$

$$\frac{7.13 \qquad \dfrac{s_1}{\textit{after}\ s_2 \wedge \textit{after}\ s_3 \sim\!>\ \textit{after}\ s_1}\ 7.6}{\textit{at}\ s_2 \wedge \textit{at}\ s_3 \sim\!>\ \textit{after}\ s_1}\ 7.2 \tag{7.14}$$

$$\frac{\dfrac{s_1}{\textit{after}\ s_1 \sim\!>\ \textit{at}\ s_4}\ 7.9 \qquad \dfrac{\dfrac{s_4}{\textit{at}\ s_4 \sim\!>\ \textit{after}\ s_4 \wedge \text{x==1}}}{\textit{at}\ s_4 \sim\!>\ \text{x==1}}\ 7.10}{\textit{after}\ s_1 \sim\!>\ x == 1}\ 7.2 \tag{7.15}$$

Now, all required lemmas have been proved. The last step simply involves chaining all these lemmas, using transitivity. The final proof of the liveness property is presented in 7.16.

$$\cfrac{7.11 \qquad \cfrac{\cfrac{7.12 \qquad 7.14}{at\ s_1 \sim> after\ s_1}\,7.2 \qquad 7.15}{at\ s_1 \sim> x == 1}\,7.2}{at\ s_0 \sim> x == 1}\,7.2 \tag{7.16}$$

The Agda code of the proof checker for this entire proof is presented in figure 7.4

Code 7.4: Agda proof of $P \vDash at\ s_0 \sim> \text{x equals 1}$.

```
s0s0~>s1 : {p : Prog s0 0} -> p ⊨ (at "s0" ~> at "s1")
s0~>s1 = enterBlock s0 "s1" (record {})

s1~>s2∧s3 : {p : Prog s0 0} -> p ⊨ (at "s1" (~> (at "s2" ∧ at "s3"))
s1~>s2∧s3 = enterPar s1

s2~>s2' : {p : Prog s0 0} -> p ⊨ (at "s2" (~> af "s2")
s2~>s2' = :=n-step s2

s3~>s3' : {p : Prog s0 0} -> p ⊨ (at "s3" (~> af "s3")
s3~>s3' = :=n-step s3

s2∧s3~>s1' : {p : Prog s0 0} -> p ⊨ ((at "s2" ∧ at "s3") ~> (af "s1"))
s2∧s3~>s1' = ~>-t (join s1 s2~>s2' s3~>s3') (exitPar s1)

s1'~>x==1 : {p : Prog s0 0} -> p ⊨ (af "s1" ~> (b* ("x" ==n (nat 1))))
s1'~>x==1 = ~>-t (flow s0 (record {})) (~>-∧-e2 (:=n-R s4))

s0~>x==1 : {p : Prog s0 0} -> p ⊨ (at "s0" ~> (b* ("x" ==n (nat 1))))
s0~>x==1 = ~>-t (~>-t (~>-t s0~>s1 s1~>s2∧s3) s2∧s3~>s1') s1'~>x==1
```

7.2.2 Liveness 2

The following example includes a slightly more complicated program in comparison to 7.2.1. The program, presented in figure 7.5, contains two concurrent statement as processes, s_4 and s_8, where the termination of each process depends on the other. The goal of this example is to show how and why some liveness proofs cannot be found for certain programs, and what modification of the programs are required to solve these issues.

Consider the program in figure 7.5, where there are two processes that depend on each other for termination.

Code 7.5: Program with co-dependent processes.

```
s0: {
  s1: p := true
  s2: q := false
  s3: cobegin
    s4: {
      s5: p := false
      s6: while ~ q
        s7: x := 5
    }
  ||
    s8: {
      s9: while p
        s10: while true
          s11: x := 6
      s12: q := true
    }
  coend
  s13: x := 1
}
```

The goal of the proof is to prove termination of the program P, in other words $P \models$ termination. For this example, trivial proof steps will be left out, since a similar method has been covered in 7.2.1.

The following list shows the major lemmas to prove for the final proof.

1. $P \models$ *at* s_0 ~> *at* s_3
2. $P \models$ *at* s_4 ~> *after* s_4
3. $P \models$ *at* s_8 ~> *after* s_8
4. $P \models$ *at* s_3 ~> *after* s_3
5. $P \models$ *after* s_3 ~> termination

The lemma (1) is trivial and therefore left out.

In order to prove termination of s_4 (2), the inner loop s_6 must eventually terminate. The chaining of *at* s_4 ~> *at* s_6 and *at* s_6 ~> *after* s_6 is sufficient to exit s_4, where the first lemma is trivial. One of the exit-while rules is used in order to exit s_6, 7.17.

$$\frac{s\ :\ \text{While}\ b\ s_0 \qquad \textit{at}\ s\ \texttt{\sim>}\ \Box\ \text{b} \qquad \textit{at}\ s_0\ \texttt{\sim>}\ \textit{at}\ s}{\textit{at}\ s\ \texttt{\sim>}\ \textit{after}\ s}\ \text{Exit While} \tag{7.17}$$

Apply 7.17 to s_6 and then prove 7.18 and 7.19, where the latter is trivial.

$$\text{at } s_6 \text{ ~> } \Box \sim q \tag{7.18}$$
$$\text{at } s_7 \text{ ~> at } s_6 \tag{7.19}$$

To prove 7.18, inference rules for accessing current block parallel processes are used, 7.20 and 7.21.

$$\frac{s : \text{Block} \qquad s_0 : \text{Stm} \qquad s_0 \in s}{at\ s_0 \text{ ~> } at\ s}\text{Infer Block} \tag{7.20}$$

$$\frac{s : s_1 \mid\mid s_2}{at\ s_1 \text{ ~> } at\ s_2}\text{Infer Par2} \tag{7.21}$$

Combining these, one can show that s_8 must eventually execute. The next step is therefore to prove 7.22.

$$\text{at } s_8 \text{ ~> } \Box\, q \tag{7.22}$$

From s_8, control will eventually be at s_9. To exit s_9 using 7.17, inner termination of the loop is required in addition to the condition $\sim p$. However, since the condition of s_{10} is always met, inner termination is impossible. By modifying the condition of s_{10} to *false*, 7.23 is used to exit the loop, yielding 7.24 using the flow control rule 7.9.

$$\frac{s : \text{While false } s_0}{at\ s \text{ ~> } after\ s}\text{Skip While} \tag{7.23}$$

$$\text{at } s_8 \text{ ~> at } s_{12} \tag{7.24}$$

$$\frac{s : b := \text{true}}{at\ s \text{ ~> } b}\text{Bool assign true} \tag{7.25}$$

Here it would be preferred to use the safety checker to verify $q \mathrel{{\sim}{>}} \Box\, q$, but for this example it will be assumed. Applying the bool-assignment rule 7.25 to s_{12}, and chaining with the safety property proves 7.22. This lemma is then used to prove

$$\frac{s_6 \qquad at\ s6 \mathrel{{\sim}{>}} \Box\ \mathrm{b} \qquad at\ s_7 \mathrel{{\sim}{>}} at\ s_6}{at\ s_6 \mathrel{{\sim}{>}} after\ s_6}\ 7.17 \tag{7.26}$$

The following steps simply involves chaining trivial lemmas of flow control 7.9, leading towards the termination of the program. Inspect 7.6 for the complete proof in Agda.

Code 7.6: Complete proof of termination for co-dependent processes.

```
s0~>ats2 : {p : Prog s0 0} -> p ⊨ (at (s 0) ~> at (s 2))
s0~>s2 = ~>-trans (enterBlock s0 "s1") (~>-trans ((:=b-T-step s1) (flow s0 (record {}))))

s2~>s3 : {p : Prog s0 0} -> p ⊨ (at "s2" ~> at "s3")
s2~>s3 = ~>-t (:=b-F-step s2) (flow s0 (record {}))

postulate q=>□q : {p : Prog s0 0} -> p ⊨ ((b* (var "q")) ~> □ (b* (var "q")))
postulate ~p=>□~p : {p : Prog s0 0} -> p ⊨ ((b* (~ (var "p"))) ~> □ (b * (~ (var "p"))))

s8~>~p : {p : Prog s0 0} -> p ⊨ (at "s8" ~> (b* (~ (var "p"))))
s8~>~p = ~>-t (infPar₁ s3) (~>-t (enterBlock s4 "s5" (record {})) (~>-∧-e₂ (:=b-F-R s5)))

s9~>s9' : {p : Prog s0 0} -> p ⊨ (at "s9" ~> af "s9")
s9~>s9' = exitWhileT s9 (~>-t (infBlock s8 "s9" (record {})) (~>-t s8~>~p ~p=>□~p)) (~>>-t (skipWhile s10) (retWhile s9))

s8~>s12 : {p : Prog s0 0} -> p ⊨ (at "s8" ~> at "s12")
s8~>s12 = ~>-t (~>-t (enterBlock s8 "s9" (record {})) s9~>s9') (flow s8 (record {}))

s8~>s8' : {p : Prog s0 0} ->p ⊨ (at "s8" ~> af "s8")
s8~>s8' = ~>t ~>t s8~>s12 (:=b-T-step s12)) (exitBlock s8 (record {}))

s8~>q : {p : Prog s0 0} -> p ⊨ (at "s8" ~> □ (b* (var "q")))
s8~>q = ~>t (~>-t s8~>s12 (~>-∧-e₂ (:=b-T-R s12))) q=>□q

s6~>s6' : {p : Prog s0 0} -> p ⊨ (at "s6" ~> af "s6")
s6~>s6' = exitWhileF s6 (~>-t (~>-t (infBlock s4 "s6" (record {})) (infPar₂ s3)) s8~>∧-e₂q) (~>-t (:=n-step s7) (retWhile s6))

s4~>s6' : {p : Prog s0 0} ~> p ⊨ (at "s4" ~> af "s6")
s4~>s6' = ~>-t (~>-t (~>-t (enterBlock s4 "s5" (record {})) (:=b-F-step s5)) (flow s4 (record {}))) s6~>s6'
```

```

s4~>s4' : {p : Prog s0 0} -> p ⊨ (at "s4" ~> af "s4")
s4~>s4' = ~>-t s4~>s6' (exitBlock s4 (record {}))

s3~>s3' : {p : Prog s0 0} -> p ⊨ (at "s3" ~> af "s3")
s3~>s3' = ~>-t (~>-t (enterPar s3) (join s3 s4~>s4' s8~>s8')) (exitPar
    s3)

s3'~>s13 : {p : Prog s0 0} -> p ⊨ (af "s3" ~> at "s13")
s3'~>s13 = flow s0 (record {})

s13'~>s0' : {p : Prog s0 0} -> p ⊨ (at "s13" ~> af "s0")
s13'~>s0' = ~>-t (:=n-step s13) (exitBlock s0 (record {}))

p⊨term : {p : Prog s0 0} -> p ⊨ term
p⊨term = term (~>-t (~>-t s0~>s2 (~>-t (~>-t s2~>s3 s3~>s3') s3'~>s13
    )) s13'~>s0')
```

7.3 Safety Proofs

A safety checker has not yet been completed in this system, although the foundation for development is in place. The dependencies between the liveness- and safety checkers is established and both use the same satisfaction relation type and can be used in the same proof; you don't need a separate module for the safety proofs.

7.3.1 Property violation in Unreachable Clauses

The problems encountered when developing the safety checker were related to the set of legal states of the programs. In order to establish safety properties, the proof checker must be able to find the set of reachable states, since the mere existence of a statement does not imply changes to the behavior of a program. In other words, let P be a program, s a statement violating ϕ, $P \vDash at\ s \sim> \sim \phi$. Going back to the context of the problem, this lemma only has meaning iff s is reachable. Therefore, in order to determine $P \vDash \Box\ \phi$, one must first prove that s is unreachable from the program starting point. In fact, this has to be made for every statement having the ability to violate ϕ. The proof checker would have to be significantly more sophisticated in order to handle these problems, but is definitely not impossible to achieve.

8 Discussion

In this chapter, the outcome of the project is discussed. The research questions are reflected upon and to what extent they were fulfilled. The design choices and the chosen development tools are evaluated, followed by a *future work* section, discussing the possible ways of extending the work in the future.

8.1 Was ELTL sufficient?

ELTL, as implemented in this thesis, revealed some issues regarding representing values of variables. The method is quite clear when dealing with trivial cases including assignment. For example, let $s_n : x := 5$. In this case, if control is at s_n at some point, x will eventually be assigned the value 5. However, as soon as assignment based on variable values are considered, complications arise due to the complex nature of concurrency. The implications of a statement incrementing a variable x may seem trivial, but can cause trouble when dealing with concurrent systems. This problem is present due to the programs are not actually being executed, but rather potential execution is being reasoned about. In the current implementation, there is no way of referring to previous values of variables, other than examining the values at some point in time. Knowing the value of a variable at some point in time does not help when incrementing based on immediate values, unless the variable holds a constant value after a certain point. The problem is prominent enough when dealing with a single statement, and gets increasingly complex when dealing with loops executing in parallel. With two loops running in parallel and both assigning to the same variable, reasoning about eventual values of the variable becomes tricky in the presented system. The possibility of adding states was discussed during the project, but since the goal included avoiding thinking about program execution, the concept was discarded. If the proof checker were more sophisticated in regards to safety verification, the handling of variables would be much more sufficient. The derivation of the $\forall$-property would be simplified with a safety checker, which would make dealing with variables much more manageable.

The strengths of the current implementation of ELTL lie in the liveness proofs. Linear logic proofs, while being somewhat tedious, proved rather intuitive and easy to follow when reasoning about program execution. For the examples presented in this thesis, the logic revealed no further complications in regards to representing program execution. With the method being tedious but straightforward, the method for proving Liveness strongly depends on the proof assistant tool, in this case Agda.

If the proof assistant is helpful and expressive, especially regarding automation of proof steps, then the process is clearly manageable even for larger programs.

8.2 Agda as Formal Proof Assistant

Agda has been a helpful tool due to the strictness of the language and the native support for mathematical notations. Agda helped simplify what would have been unintuitive and time consuming in traditional languages, both in terms of mathematical reasoning, but also the inference of statements. The implementation makes great use of the strictness when construction proofs; if a proof can be constructed then it is considered valid. A proof step is immediately rejected when the rule cannot be inferred from the established theorems and lemmas. This immediate feedback is of great assistance when constructing proofs and contributes to the intuitive feel of the method.

The automatic theorem proving tool of Agda proved quite useful in certain cases. While still being in its early stages of development, the theorem prover sometimes managed to prove non-trivial theorems in a short period of time. The consistency of the tool was however lacking since it often failed to prove seemingly trivial implications, thus not reliable in its current stage. It does however show the potential of getting rid of much of the tedious elements of the proof system, thus making a solid argument for its continued development.

8.3 Comparison to Model Checking

The dominance of model checking compared to formal proofs has been prominent in the industry, mainly due to the accessibility of the method. A minimum of overhead is required when using the tools, and results are often generated quickly, provided the program is not overly complex. Model checking has however mostly been developed for sequential environments, since the non-determinism introduced by concurrency cannot easily be modeled using a pure algorithmic approach. While a model checker is a useful tool, one must keep in mind is that the model checkers do not produce proofs. The tool might fail to disprove a property, but that does not imply validity. It simply means the checker could not find a counterexample. Finding and understanding proofs and their intermediate steps is a great way of getting an intuitive feel of the problems and their potential solutions. As a tool for students, a proof assistant tool is definitely preferred over simply plugging in a logical formula and hoping for positive results based on intuition. Therefore, even though model checking is currently preferred in the industry, proof checking can still hold significance simply for academic purposes alone.

The lack of support for concurrency is a major drawback of the model checking tools, since most programs of today use concurrency to at least some degree. The question therefore becomes whether proof assistants could be further developed with regards to user experience. Making the tools prove complex properties by themselves might be too much to ask at the moment, but using automated systems as proof assistants, where the user merely leads the proof checker towards the goal, could certainly be both possible and useful. The built-in automatic proof assistant in Agda has proved to be a useful feature when dealing with the tedious nature of formal proofs. The feature in its current state manages to prove simple properties, but struggles when dealing with even slightly complex proofs, and is mostly limited to the constructors of data types. Additional information, or hints, can be provided to the assistant to decrease the search span, leading to faster and more successful conclusions. However, the supplied hints do not seem to make a significant difference and are most often not worth the effort. Expanding and improving this feature could therefore make a significant contribution to the assisted proof verification.

Having completely automated proof systems is not necessarily always preferable. As mentioned earlier, the goal of the thesis is to develop a tool with students as the potential users. Students need to learn the methods of conducting proofs, but if all steps on the way become automated, the learning aspect is removed. A significant benefit from taking the time to deal with tedious proofs, is the learning outcome of the process. It is however necessary to remember that the proofs conducted in the presented systems largely consists of trivial modification of ELTL formulae. These trivial proof steps are tedious and do not provide any significant learning benefits after the basic knowledge has been established. Having automated proof assistants to help with the trivial proof steps is therefore necessary in order to make the proof checker worthwhile.

8.4 Tool for Students

One of the main objectives of the study involved researching the potential for this tool to be used by students. In order for this to be feasible, the tool would have to be sophisticated enough to handle common programming examples from student literature, but also accessible and well documented. Additionally, the proof checker would require thorough testing and verification. Since there is no graphical interface implemented, the proof checker is simply an Agda-framework for proof construction. The students would therefore be required to know Agda to some degree in order to use the tool. This requirement can be seen as both an advantage and a disadvantage depending on the educational background of the student. Having an interface between the user and the proof system could make it significantly more accessible for beginners, and also provide an overview of the proofs. The proof could be visualized using proof trees, giving an intuition of the progress and the current state of the proof. The interface could also be extended with the program supplying possible ways forward, in other words guiding the user towards the goal.
On the other hand, if the user is required to be familiar with Agda in order to use the tool, the user will gain a deeper understanding of the underlying logic, as well as gaining the ability to extend the library by contributing with new content. The transition from handwritten proofs to an Agda implementation is not necessarily difficult, since working with Agda is already being quite similar to writing proofs

by hand. As long as the user is familiar with functional programming, the language difficulties should be minimal.

It is however clear that the proof checker requires further development in order to be useful to students, mainly due to the underdeveloped safety checker.

8.5 Future Work

In order for the system to be considered complete, an implementation of the safety checker is required. If more interesting proofs are to be conducted, a combination of liveness and safety is needed. The foundation of combining the two methods is already established in the same proof system, without having extensive dependencies between the two. The safety checker can be implemented rather independently from the liveness component, enabling usage of completely different construction methods compared to the liveness module. The task involves coming up with a method to handle variable states without going too much in the direction of program execution.

Another aspect to take into consideration is the verification of the proof checker itself. As of now, the proof checker is based upon established LTL-rules and semantics. While having that stable foundation is reassuring, it is definitely not a guarantee for correct behavior or immunity to errors. Therefore, proving properties such as soundness and completeness of the tool would be an interesting extension to the project.

8.5.1 Automatic Proof Checking

As shown in the results of the thesis, even conducting proofs for small programs can be tricky and time consuming. The automatic prover of Agda showed potential, since rather trivial proof steps could be automated, thus saving time and effort. The function is however still quite underdeveloped and had a hard time proving even slightly complicated properties. Further development of the theorem prover could therefore have a large impact on the potential of formal proofs for concurrent programs.

9
Conclusion

In this thesis, the implementation of a proof checker for extended linear temporal logic has been presented. The results show how to conduct proofs about liveness in the system, as well as describing a general method for verifying safety properties. With further development of the proof checker automatic theorem prover of Agda, this method could be useful for students or other academic purposes.

9.1 Research Questions

The research questions involve the development of a tool aimed towards students. The results from chapter 7 are promising, but the proof checker tool is currently underdeveloped for the purpose of general usage. The defined CPL and the proof system are however fairly easy to use and are capable of representing a wide range of common problems.

The proof checker was mainly build upon the theory of Lamport. The established methods were useful and well defined, but a few issues arose during implementation. Trivial proof steps were often omitted and the state of the proofs were sometimes not clear. The safety requirements were often assumed by looking at the programs, which is not suitable for implementation in Agda, giving rise to new methods for representing program states.

The language Agda showed promising results as a logical framework, even when dealing with an imperative-style language. The strictness of Agda regarding termination caused some issues when dealing with arbitrary programs of the CPL, but these issues were resolved.

Bibliography

[1] J. Harrison, "Formal methods at intel–an overview," in *Second NASA Formal Methods Symposium*, vol. 8, 2010, pp. 179–195.

[2] E. Seligman, T. Schubert, and M. A. K. Kumar, *Formal verification: an essential toolkit for modern VLSI design.* Morgan Kaufmann, 2015.

[3] R. Kaivola, R. Ghughal, N. Narasimhan, A. Telfer, J. Whittemore, S. Pandav, A. Slobodová, C. Taylor, V. A. Frolov, E. Reeber *et al.*, "Replacing testing with formal verification in intel coretm i7 processor execution engine validation." in *CAV*, vol. 9. Springer, 2009, pp. 414–429.

[4] T. W. Barr and S. Rixner, "Medusa: Managing concurrency and communication in embedded systems." in *USENIX Annual Technical Conference*, 2014, pp. 439–450.

[5] A. Pnueli, "The temporal logic of programs," in *Foundations of Computer Science, 1977., 18th Annual Symposium on.* IEEE, 1977, pp. 46–57.

[6] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper, "Simple on-the-fly automatic verification of linear temporal logic," in *Protocol Specification, Testing and Verification XV.* Springer, 1995, pp. 3–18.

[7] M. Y. Vardi, "An automata-theoretic approach to linear temporal logic," in *Logics for concurrency.* Springer, 1996, pp. 238–266.

[8] J. H. Andersen, E. Harcourt, and K. Prasad, "A machine verified distributed sorting algorithm," *BRICS Report Series*, vol. 3, no. 4, 1996.

[9] D. M. Goldschlag, "Mechanically verifying concurrent programs with the boyer-moore prover," *IEEE Transactions on Software Engineering*, vol. 16, no. 9, pp. 1005–1023, 1990.

[10] S. Ray and R. Sumners, "Specification and verification of concurrent programs through refinements," *Journal of automated reasoning*, vol. 51, no. 3, pp. 241–280, 2013.

[11] L. Lamport, "Specifying concurrent systems with tla^+," *NATO ASI SERIES F COMPUTER AND SYSTEMS SCIENCES*, vol. 173, pp. 183–250, 1999.

[12] D. Cousineau, D. Doligez, L. Lamport, S. Merz, D. Ricketts, and H. Vanzetto, "Tla+ proofs." *FM*, vol. 7436, pp. 147–154, 2012.

[13] S. Owicki and L. Lamport, "Proving liveness properties of concurrent programs," *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 3, p. 155–495, July 1982.

[14] A. Wiki, "The agda wiki," http://wiki.portal.chalmers.se/agda/pmwiki.php, 2017.

[15] D. Hilbert, *Grundlagen der geometrie.* Springer-Verlag, 2013.

[16] J. von Plato, “The axioms of constructive geometry,” *Annals of pure and applied logic*, vol. 76, no. 2, pp. 169–200, 1995.

[17] J. V. Plato, “Formalization of hilbert’s geometry of incidence and parallelism,” *Synthese*, vol. 110, no. 1, pp. 127–141, 1997.

[18] E. Bergsten, O. Larsson, T. Rastemo, O. Rutqvist, and A. Standár, “Methods for using agda to prove safety and liveness for concurrent programs,” *Department of Computer Science, Chalmers University of Technology*, June 2017.