CHALMERS

Methods for using Agda to prove Safety and Liveness for Concurrent Programs

Bachelor of Science Thesis in Computer Science and Engineering

ERIK BERGSTEN
OSKAR LARSSON
TOBIAS RASTEMO
OSKAR RUTQVIST
ANDREAS STANDÁR

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, June 2017

Bachelor of Science Thesis

Methods for using Agda to prove Safety and Liveness for Concurrent Programs

ERIK BERGSTEN
OSKAR LARSSON
TOBIAS RASTEMO
OSKAR RUTQVIST
ANDREAS STANDÁR

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
University of Gothenburg

Göteborg, Sweden 2017

Methods for using Agda to prove Safety and Liveness for Concurrent Programs

ERIK BERGSTEN
OSKAR LARSSON
TOBIAS RASTEMO
OSKAR RUTQVIST
ANDREAS STANDÁR

Examiner: NIKLAS BROBERG

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
SE-412 96 Göteborg
Sweden
Telephone: +46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg 2016

Methods for using Agda to prove Safety and Liveness for Concurrent Programs

ERIK BERGSTEN
OSKAR LARSSON
TOBIAS RASTEMO
OSKAR RUTQVIST
ANDREAS STANDÁR
Department of Computer Science and Engineering,
Chalmers University of Technology
University of Gothenburg

Bachelor of Science Thesis

Abstract

It is hard and error-prone to prove that a given concurrent program has a given property. To help develop such proofs, a modern implementation of dependent type theory, the proof checking tool Agda [Nor09], is used. The methods are based on the semi-formal theory of the semantics of concurrent programming from [OL82] (Owicki and Lamport, 1981);. The formalisation and implementation in Agda is done with two different approaches, which we call Proof by Construction and Proof by Control Flow. The first uses the dependent data types of Agda to create proofs in the form of type elements. The second uses recursive functions to check proofs written down as a tree of steps. With each method, how to prove liveness properties for concurrent programs is shown. For safety properties, time constraints restricted the work to proofs for sequential programs. The Agda implementations of these methods work well, but like formal and manually constructed proofs our machine checked proofs are still long and tedious. The implementations have potential for further development.

Sammandrag

Det är svårt och invecklat att bevisa att ett visst parallellt program har en viss egenskap. För att underlätta skapandet av sådana bevis så har vi utvecklat en modern metod, som är byggd på typechecking, med hjälp av bevisassistenten Agda. Denna implementation bygger på en semi-formell logisk grund som lades av Owicki och Lamport 1982, [OL82]. Två olika metoder för att formalisera och implementera denna logiska grund programmerades i Agda. De kallas för ”Proof by Construction” respektive ”Proof by Control Flow”. Den senare använder sig av rekursiva funktioner för att verifiera ett bevis med hjälp av generella regler för ett kontrollflöde. Den första utnyttjar Agdas strikta typsystem och använder typkonstruktorer för att skapa hela bevis. Båda metoderna kan visa livlighetsegenskaper för parallella program, medan bevis för säkerhetsegenskaper endast har påbärjats och kan ännu bara bevisas för sekventiella program. Den utvecklade bevismetodeerna i Agda fungerar bra, men precis som alla formella och maskinverifierade bevis är det långa och krävande. Det kan fungera som en grund för vidare utbyggnad.

"There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult."

C.A.R. Hoare

Acknowledgement

Throughout this project we have been assisted by some people and we especially want to thank:

Johan Häggström for the cooperation throughout this project. He has been very helpful in all stages of the project, especially his great knowledge of Agda. He introduced us to Agda by several workshops and sessions, spending a lot of time on this, and needs a lot of credit in our making of the final products. We have had closely correlating work with him, especially in the development of the Proof by Control Flow method. The majority of the code for this proof method was developed in cooperation with him.

K. V. S. Prasad, our supervisor, for making this project possible. He has gone over and beyond the expectations for a supervisor and has been very supportive in all aspects of our work. During the writing Prasad helped with very detailed feedback, and with setting the context. He wrote Secs 1 and 2.1. But under time pressure he also sometimes wrote the actual final form rather than try remotely to get us to write it to his satisfaction.

Nachiappan Valliappan ("Nachi") for helping us during the earlier stages of the project with the connections between TLA [Lam94] and the approaches we use.

Carl Söderpalm for taking the initiative, being the first student saying that the proposal of this project was interesting, making this project possible.

Contents

1 Introduction

This report presents results on formalising methods to use Agda to prove safety and liveness properties of Concurrent Programs.

1.1 Background

Computers are now ubiquitous in many safety-critical applications, for example embedded in vehicular and medical systems. Other systems might not be safety-critical, but could still be mission-critical: errors in hardware design, for example, can result in ruinously expensive recalls (hardware cannot be patched like software). In all such systems, a whole range of formal verification (FV) tools and techniques are now used to ensure design correctness [Har10, SSK15, Sel16, KGN$^+$09].

Among such tools are model checkers and proof checkers. A model checker takes a program p and a property ϕ, and says whether p satisfies ϕ (written $p \vDash \phi$). Also, if p fails ϕ, a model checker will display a *run* (execution) of p where the failure shows up. Thus model checkers are very useful to catch bugs during development. But they do not tell you *why* $p \vDash \phi$ holds, if it does. Such information, a *proof*, is useful not only to improve confidence in the design of p, but also to give insight useful during further development or modification. Because such proofs are long, delicate, and tedious, and also demand modelling and mathematical skills, it is very useful to get them mechanically checked (or even partially mechanically developed). The idea of machine-checked proofs has been widely explored, and proof checking programs have been available since the 1970's [GMW79]. The *Agda* [Nor09] proof checker is the product of many years of development at Chalmers [Coq92], and very modern in approach. It is the tool used in this project, which is about *formal methods* for proving properties of concurrent programs.

1.2 Concurrent Programs

Concurrent systems too are now ubiquitous: hotel and airline reservation systems and banking systems both connect servers to large numbers of users acting independently and yet sharing a database. It is obviously harder to coordinate and synchronise independent activities, and to prevent interference between them, than to organise one sequential workflow. Among other problems, concurrent systems are inherently nondeterministic—over a communication medium shared between independent users, we cannot say who will speak first, and the answer may change each time the program runs. Concurrent programs cannot therefore be debugged in the traditional way, by re-running the program with breakpoints.

The way out is to look at all possible runs, either with a model checker or with a proof checker. Agda has been only little used for concurrent systems [O'C16, DSNB16]. A predecessor of Agda, ALF, was used in 1996 [AHP96] to not only prove a whole concurrent program correct, but also to prove its implementation correct. It was pioneering work in its time, but [AHP96] was not followed up because of illness of one of the authors. The present project is a fresh look at concurrent programs in Agda. But our scope is more narrow than that of [AHP96]; we look only at the properties of the concurrent program text, and do not consider data dependent behaviour or implementation. It is also different in that we characterise the behaviour of our programs logically, as in [Hoa69, OL82], while [AHP96] uses operational semantics as in [Mil89, Pra95].

1.2.1 Safety and Liveness Requirements

A classic example of a *safety* requirement is that a railway signalling system should prevent collisions, while the *liveness* requirement is that each train should be allowed, eventually, to proceed. Turning all signals red meets the safety requirement but breaks liveness.

We show how to prove liveness properties for concurrent programs. For safety properties, we only had the time to do proofs for sequential programs. We could only do a few toy proofs, but are able to say clearly what the next steps are, and how they can be carried out.

1.3 Problem Description and Purpose

As already mentioned, proving concurrent safety and liveness properties is nothing new, and machine checking these properties is also well established. The purpose of the thesis is to present methods of how to formally prove properties of concurrent programs using Agda, a less explored subject.

The main work has been to formalise existing methods and implement them in Agda. Our main source is [OL82], from whom we borrow CPL, the toy programming language we use, and ELTL, in which we formulate program properties. ELTL is an extension of Linear Temporal Logic (LTL), itself an extension of propositional calculus.

The results presented consist of several methods, all used for formal verification. The methods have been divided into two categories abbreviated as: Proof by Control Flow and Proof by Construction. The first is used to prove concurrent liveness properties and the latter to prove sequential safety as well as concurrent liveness.

2 Technical Background

This section covers the prerequisites for understanding the results presented. It explains the tools used and some of the most important theoretical concepts of historic work in the area. To fully comprehend the content the reader is expected to have quintessential knowledge of concurrent programs. The most essential being how to reason about termination and contention. This corresponds to an introductory course to Concurrent Program on university level. For an in depth explanation see for example [BA06]. In addition intermediate knowledge of propositional logic and proving by natural deduction is assumed, see for example [HR04].

2.1 Concurrenct Programs

Concurrent programs have been analysed since the 1960's—a very few example papers are: [Dij65, Hoa69, Lam80, OL82, Lam84, LS84, Mil89, Sti91, MPW92, BA06]. Below is a theoretical model that captures their essential aspects and permits reasoning about them.

2.1.1 Modelling Concurrent Programs

Concurrent programs consist of processes communicating by messages or via shared variables. Think of a *process* as an independent computing agent, which, once *spawned*, cycles between *running* and *waiting* states until it *terminates.*

A process waits, like a train waiting for a green signal, for a specific *event* or *condition*, which can only be triggered by another, running, process. So a system is *deadlocked* if all process are waiting; no one can release any of the waiting processes.

Processes run at an unknown and possibly varying speeds—it could happen that a "running" process is not scheduled onto a CPU. Such a process is not waiting, but is not progressing either.

What happens if a process is never scheduled when it can run? To rule out such silly cases, we usually impose a *fairness* requirement on the system scheduler, which says that a process that is continually ready to run will actually eventually run.

2.1.2 More Terminology

Safety of railway signalling can be ensured by *mutual exclusion* (*"mutex"*), which says, here, that at most one train may use any given section of track. The sections of program code which use the mutex are called *critical sections.*

2.1.3 Reasoning by Counterexamples

Because of the indeterminate speeds, and because contention for shared resources (variables or communication media) is resolved arbitrarily, concurrent programs are inherently nondeterministic, nothing can be said about which process will execute first. Hence, a concurrent program may show different behaviour when re-run, traditional debugging fails, and the general behaviour of all possible *runs* of a program must be considered.

To show that a safety condition has been failed, a counterexample is merely a single program state where the condition does not hold. But the counterexample that shows the failure of a liveness (or *progress*) requirement is an execution path, often a loop, where a particular process is never released from its waiting state.

2.2 Agda

In contrast to conventional programming languages Agda programs are seldom run. Instead it is often enough to make sure that a program *type checks.* This is done by *loading* a program, which means that Agda checks the type of all statements and declarations.

Presented in figure 1 is some Agda code. The data type Nat represents the set of natural numbers. The elements of this set are either zero or a successor of another natural number. This is encoded in Agda by the constructors `zero` and `suc`. By recursively calling the `suc` constructor

```
data Nat : Set where
  zero : Nat
  suc  : Nat -> Nat
```

Figure 1: Small Agda data type representing the natural numbers, here defined as either zero or a successor of another natural number. This is represented in Agda by the constructors `zero` and `suc`.

we can create new natural numbers, e.g. 1 is written as suc (zero). The value is therefore defined as the combination of constructors used to construct it.

Thus Nat is both a type and a set; we do not distinguish between the two in Agda. Further, we can only claim that an object is an element of Nat by producing it using the constructors `zero` and `suc`. Thus, claims (or propositions) too are identified with sets or types.

A clearer application of this idea of propositions as sets can be seen in figure 2. Here a new type is declared, the set `even`, which takes a natural number as a parameter and returns a proof that it is even. Its constructors `ZERO` and `STEP` say respectively that zero is even, and that if `n` is even, then so is `n+2`. So the proof that `zero even` is simply `ZERO`, but to show `4 even`, we need `STEP (2 even)`. That is, to prove that 4 is even, we need first to prove that 2 is even.

To clarify the notion of propositions as sets, note that the set `(3 even)` is empty (we cannot construct any elements of it), but the set `(4 even)` has the element `STEP (2 even)`. Thus a proposition is true if we can construct an element of its set representation.

`STEP` also provides our first example of a *dependent type*. The type returned by the constructor `STEP` depends on the actual type of the first parameter. Note also that {`n : Nat`} specifies that `n` must be a of the type `Nat`. This has to be explicitly declared as Agda would otherwise complain that the type of `n` is not known.

Finally, since types are seen as propositions, with proofs as elements, it follows that constructors in Agda can be treated as rules in a proof system.

```
1  data _even : Nat -> Set where
2    ZERO : zero even
3    STEP : {n : Nat} -> n even -> suc (suc n) even
```

Figure 2: A data type representing the set of even numbers. They are defined as the base case zero, `ZERO` in Agda, or another even number plus two, `STEP`. These constructors can be considered rules, compare to natural deduction. Note that on line three is an example of a dependent type.

As a last step we might want to use these types to prove that 4 is even. The proof for this can be seen implemented in figure 3. The proof is the application of the constructors `ZERO` and `STEP`, this can be seen on line two. We can express this in text as: 0 is even, $(0+2)$ is even, $((0+2)+2)$ is even, i.e 4 is even. Compare to natural deduction style of proving by applying rules.

```
1  foureven : suc (suc (suc (suc zero))) even
2  foureven = STEP (STEP ZERO)
```

Figure 3: A function in Agda that represents a theorem proving that four is even. This is done by calling the constructors of the `even` type. This is analogous to using rules.

2.2.1 Interactive Construction using Agda

In addition to using Agda [Wik17] as a type checking language there exists interactive ways of using it. This can at the moment be done using either Emacs[GNU] or Atom[Ato].

One example of this is using something called *holes*. If a `?` is typed as a placeholder for an expression, Agda will replace it with `{!  !}0`, when loading. This is called a hole. See figure 4 where the code in the editor can be seen before and after loading.

```
fun : (a : A) -> (b : B) -> C
fun a b = ?
```

```
fun : (a : A) -> (b : B) -> C
fun a b = {! !}0
```

Figure 4: Demonstration of how Agda treats ? when loading. As can be seen on the bottom two lines it is replaced by what is called a hole. Note that the actual text in the editor changes upon loading.

After generating a hole Agda will treat the hole as a *goal*, something to be filled in later with the data type that should go in the hole. It is then possible to place the cursor in the hole and execute a selection of commands. Two of them are *case split* and *auto fill*. With case split it is possible to enter one of the input variables in the hole, and then have Agda auto generate all cases for that type. With auto fill Agda will try to find constructors that can construct the desired type the hole should have.

Apart from using holes there exists a *normal form*. This is a command that when executed lets the user input commands in a dialog and then have Agda evaluate them. One can for example call functions specified and check the value of variables.

2.3 A Small Concurrent Programming Language (CPL)

We introduce CPL first through examples.

2.3.1 Examples of Nondeterminism

Say we have a process b and a process c, as in the program represented in figure 5. Both processes tries to assign a value to x and then terminates. If we run both processes, b and c, simultaneously, what will x:s value be after termination? The answer is 5 or 6 depending on whether b or c is executed last. Even though the program is rather simple it illustrates the nondeterministic properties of concurrent programs.

```
integer x;
a: cobegin
    b: x := 5;
   []
    c: x := 6;
   coend
```

Figure 5: A simple program consisting of two processes, b and c, both which assign a value to the global variable x. Depending on the execution order of the processes the final value of x can be either 5 or 6.

The next example program can be seen in figure 6. The program just consists of two processes who each tries to increment the value of x by 1. We may think that the final value of x would be 2, always. But if the processes executes in the order

$$b \to d \to c \to e \ldots$$

the final value of x is 1. If we think of b, c and d, e as being critical sections, we see that the program does not ensure *mutual exclusion*. Both processes can enter their critical sections at the same time. For further examples of principles and problems of concurrent programs see [BA06].

2.3.2 Remainder of CPL Syntax

Note that variable declarations are made at the beginning of the program, and that each statement of code is *labeled* to simplify referencing in proofs. See figure 7 for a larger example program in the CPL.

```
integer x, temp1, temp2;
a: cobegin
    b: temp1 := x;
    c: x := temp + 1;
   []
    d: temp2 := x;
    e: x := temp2 + 1;
    coend
```

Figure 6: A program consisting of two processes and a global variable x. At a first glance it seems like the program should increment the value of x twice resulting in 2 but depending on the execution order the final value of x can vary.

```
integer x, y;
a: < x := 0 >;
b: cobegin
    c: < y := 0 >;
    d: cobegin
        e: < y := 2*y >;  []  f: < y := y+3 >;
       coend
   []
    g: while < y = 0 > do  h: < x := x+1 >;
   coend;
i: < x := 2*y >;
```

Figure 7: A concurrent program demonstrating CPL.

As the smaller examples already showed, the parallel processes in a cobegin/coend statement are separated by a `[]`. Finally, note that `<` and `>` enclose *atomic actions*, which execute as if the whole enclosed sequence were a single action. All assignments throughout this report are considered atomic.

2.4 Linear Temporal Logic

Linear Temporal Logic (LTL) is an extension of propositional logic. It allows us to express logic regarding future events. This is done by introducing two new operators: $\Box$ and $\Diamond$, *always* and *eventually*. The definition for the two operators is as follows,

$$\Box\,\psi : \psi \text{ holds now and forever}$$
$$\Diamond\,\psi : \psi \text{ holds now or sometime in the future}$$

where ψ is a proposition in propositional logic or LTL, called an *immediate assertion*.

2.4.1 Execution Paths

To be able to interpret this correctly for a computer program we have to be specific about what we mean by *always* and *eventually*. For a concurrent program we can define an execution path to be a string of executions by different processes. Because of the nondeterminism of concurrent programs we may possibly have an infinite number of execution paths.

We can now define the operators $\Box$ and $\Diamond$ using paths instead as

$$\Box\,\psi : \psi \text{ holds at the current state and on all subsequent paths.}$$
$$\Diamond\,\psi : \psi \text{ holds at the current state or at some state along all subsequent paths.}$$

The two introduced operators are actually dual and can be related to each other by equation (1). The interpretation is straight forward: if ψ does not hold forever it must eventually be that ψ does

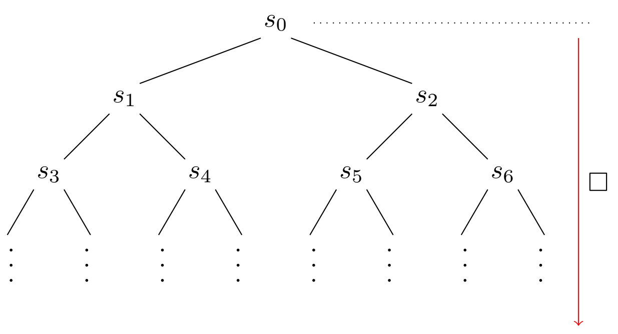

Figure 8: A tree representing possible execution paths of a concurrent program. The tree only shows some paths subsequent to s_0. ψ has to hold for all states in the tree for $\Box\psi$ to hold at s_0.

not hold.

$$\neg\Box\psi \equiv \Diamond\neg\psi \tag{1}$$

This can be compared with the duality of $\wedge$ and $\vee$, which is quite similar in nature.

2.4.2 Extended LTL

When proving statements regarding programs in the CPL, [OL82] use an extension of LTL (ELTL). This version adds the propositions

- *at a*
- *in a*
- *after a*

Here a is a program label. The proposition *at a* holds for all states where control is at the beginning of a. *in a* is true *at a* and at any statement in a block contained by a and *after a* is at the next sequential statement that is not *in a*. For more formal definitions of the extensions, see [OL82]. The point of ELTL is that it permits purely logical reasoning without referring to a computation stepping through the program.

Another common notation is $a \leadsto b$, read *leads to*, defined as $\Box(a \rightarrow \Diamond b)$. This means that any state of the computation that satisfies a will lead after zero or more steps to a state that satisfies b.

2.5 Reasoning about Concurrent Programs

```
integer x; boolean p;
a: cobegin
    b: p:=false;
   []
    c: while p do
        d: x:=x+1;
coend;
```

Figure 9: A terminating program with infinitely many possible execution paths.

The first example requires the use of many types of rules to prove termination. It is shown in 9. Proving termination for this program informally, yet thoroughly, is done in the *proof lattice* in figure 10, where the nodes are labelled by assertions, and the arrows are “leads to” connectives. Branching happens when one assertion *or* another is shown true. The proof can be explained with the following steps:

$$\begin{tikzcd}
 & & \mathit{at}\ b \land \mathit{at}\ c \arrow[d, squiggly] & \\
 & & \mathit{after}\ b \land \neg p \arrow[d, squiggly] & \\
 & & \Box(\mathit{after}\ b \land \neg p) \arrow[dl, squiggly] \arrow[ddddd, squiggly, bend left=60] & \Box(\mathit{after}\ b \land \neg p) \\
 & \mathit{in}\ c \arrow[dl, squiggly] \arrow[dd, squiggly, bend left=60] & & \\
 \mathit{at}\ d \arrow[dr, squiggly] & & & \\
 & \mathit{at}\ c \arrow[d, squiggly] & & \\
 & \mathit{after}\ c \arrow[dr, squiggly] & & \\
 & & \mathit{after}\ c \land \mathit{after}\ b &
\end{tikzcd}$$

Figure 10: A proof lattice for proving termination of the program in 9. Each node is the immediate assertion that is true after applying a hypothetical satisfaction rule from the former node. The arrows are the *leads to* operator, and branching is done when the rule proves one assertion *or* another being true.

1. The assertion *at b* will by the fairness assumption mean that `b` will eventually execute, proving that $\mathit{after}\ b \land\neg p$ will eventually be true.
2. Looking at the program reveals that there are no other assignments to p, which will therefore remain false.
3. We assume that expression evaluations terminate, e.g. p will evaluate to true or false in finite time. This assumption ensures `while` statements terminates if executed, and by fairness it will eventually execute. Executing `c` has two possible outcomes either it enters the loop or continues after it. $\mathit{at}\ c \leadsto \mathit{in}\ c\ \lor\ \mathit{after}\ c.$
4. *in c* means by definition either *at d* or *at c*
5. *at d* will eventually execute by fairness and proves *at c*
6. The step from *at c* to *after c* is true under the assertion of $\Box\neg p$. The $\Box$ is essential since it rules out interference. An assertion of a program eventually being at the boolean evaluation of the while loop with the boolean set to false does not imply anything, since another process might set it to true before the evaluation gets executed.

2.5.1 A Basis for Reasoning about Concurrent Liveness

From [OL82], here is a set of rules for reasoning about liveness properties. An example rule is presented below in equation 2.

Atomic Assignment Axiom

$$\frac{a : \langle x := f \rangle}{\mathit{at}\ a \leadsto \mathit{after}\ a}\ aaa \tag{2}$$

The rule states that any atomic assignment will progress. Fairness is incorporated into this rule (if the process that executes the assignment is never scheduled, progress cannot happen).

[OL82] does not deal with how to show safety properties, but these properties are sometimes needed to show liveness. These are instead informally motivated. In [Lam80], the notation for concurrent safety is used and looks like this:

$$\{P\}\,Q\,\{R\} \tag{3}$$

This notation is very similar to Hoare triples [Hoa69], but the meaning here is subtly different: given $\Box(\mathit{in}\ Q \supset P)$ we will have R, when Q terminates.

The graphical proof method using *proof lattices*, already introduced, is based upon splitting cases and a rule:

$$\frac{P \leadsto (R_1 \vee R_2), \quad R_1 \leadsto S, \quad R_2 \leadsto S}{P \leadsto S} \tag{4}$$

By writing graphs with lines that split at *nodes* we emulate the $\vee$-cases. By then showing that all paths lead to the same property S we conclude that what what was true at the node leads to S.

2.6 Hoare Logic

Hoare Logic, introduced in [Hoa69], is a logical framework for reasoning about sequential programs. It is used mostly to establish safety properties for sequential programs.

The basic idea is to have triples in the form (5).

$$P\{Q\}R \tag{5}$$

These are called *Hoare triples.* P and R are logical propositions and Q is a sequential program. P and R are called pre- and postconditions respectively. The meaning is that if we have P upon execution of Q we have R after. We can use these triples to reason logically about computer programs, using the rules below.

2.6.1 Hoare Triple Rules

There is one axiom D_0, and 3 basic rules D_1 - D_3. For a more in-depth explanation of the rule set, see [Hoa69]. But we show the basics here.

The axioms can be seen in (6) - (8).

D_0 Axiom of Assignment:

$$\frac{}{P(a)\,\{x := a\}\,P(x)}D_0 \tag{6}$$

The axiom says that any proposition P that holds for a will also hold for x after assigning $x := a$. Many find this rule strangely “backwards” at first sight. That is because our assignment notation is backward. The information flow of the assignment is from a to x; hence the rule.

Another rule is the rule of sequential composition, presented in (7).

D_2 Rule of Composition

$$\frac{P\,\{Q_1\}\,R_1,\ R_1\,\{Q_2\}\,R}{P\,\{Q_1;Q_2\}\,R}D_2 \tag{7}$$

The rule of composition states that given a program Q_1 whose postcondition is the precondition of Q_2, the two programs can be sequentially composed. The conclusion of the rule makes no mention of the intermediate condition R_1.

D_3 Rule of Iteration

$$\frac{(P \wedge B)\,\{S\}\,P}{P\,\{\mathit{while}\ B\ \mathit{do}\ S\}\,\neg B \wedge P}D_3 \tag{8}$$

Here, the proposition P is called the *invariant* of the loop. As the premise of the rule says, if P holds before S runs, it also holds afterwards. So the iteration rule seems to say that nothing much happens with a loop, except that the condition B will not hold after the loop. An example will dispel this misconception, and illustrate the power of invariants.

2.6.2 Application of Hoare Logic

As an application of Hoare Logic consider the program in figure 11. This program is taken from [Hoa69] and finds the quotient and remainder of x/y.

```
integer x, y, r, q;
a: r := x;
b: q := 0;
c: while  y <= r  do
    d:  r := r-y;
    e:  q := q+1
```

Figure 11: Simple sequential program that finds the quotient and remainder of $\frac{x}{y}$.

Intuitively it is easily verified that after termination $r < y$ and that we can recover x by the formula $x = r + y \times q$. Indeed, the latter is an invariant for the loop. It holds at c; and every iteration of the loop preserves it. Simply putting $r < y$ into the invariant now gives the result.

Following the proof in [Hoa69], we prove this formally using the Hoare rules, as shown in table 1. The programs in the Hoare Triples refer to the labels in figure 11. Lines 4 and 10 of the proof

	Proof Steps	Rule
1	true $\supset x = x + y \times 0$	Basic arithmetic
2	$(x = x + y \times 0)\,\{a\}\,(x = x + y \times 0)$	D_0
3	$(x = x + y \times 0)\,\{b\}\,(x = x + y \times q)$	D_0
4	true $\{a\}\,(x = r + y \times 0)$	$D_1(1, 2)$
5	true $\{a; b\}\,(x = r + y \times q)$	$D_2(4, 3)$
6	$(x = r + y \times q) \wedge (y \leq r) \supset x = (r - y) + y \times (1 + q)$	Basic arithmetic
7	$x = (r - y) + y \times (1 + q)\,\{d\}\,x = r + y \times (1 + q)$	D_0
8	$x = r + y \times (1 + q)\,\{e\}\,x = r + y \times q$	D_0
9	$x = (r - y) + y \times (1 + q)\,\{d; e\}\,x = r + y \times q$	$D_2(7, 8)$
10	$(x = r + y \times q) \wedge (y \leq r)\,\{d; e\}\,x = r + y \times q$	$D_1(6, 9)$
11	$x = r + y \times q\,\{c\}\,\neg(y \leq r) \wedge (x = r + y \times q)$	$D_3(10)$
12	true $\{a; b; c\}\,\neg(y \leq r) \wedge (x = r + y \times q)$	$D_2(5, 11)$

Table 1: A formal proof that the sequential program in figure 11 satisfies the properties $r < y$ and $x = r + y \times q$ after termination.

also used the formal logical rules in equation 9

D_1 Rules of Consequence:

$$\frac{P\,\{Q\}\,R, \quad R \supset S}{P\,\{Q\}\,S} D_1 \qquad \frac{P\,\{Q\}\,R, \quad S \supset P}{S\,\{Q\}\,R} D_1 \tag{9}$$

3 Formalising Informal Arguments

The Owicki-Lamport formalism [OL82], OL, is the basis for our Agda proof implementations. Though very convenient for human reasoning, several applications in OL are not formal enough for type checking. Much of our work therefore consisted of formalising the OL proof methods.

3.1 Example of Excluded Formality

To illustrate the problem of formality consider the program in figure 12. Say that for the program the safety property $\Box\neg p$ holds. We want to apply the formal While Exit Rule (WER), see equation (10), in order to prove the liveness property

$$at\ b \rightsquigarrow after\ b.$$

The problem is that given $(at\ b \wedge \Box\neg p)$, direct application of WER is not allowed. The rule clearly states that $\Box(at\ b \supset \neg p)$ is needed. In a completely formal system we need several rules to first show that $(at\ w \wedge \Box(at\ w \supset \neg B))$ also holds. Then we can apply (10).

While Exit Rule

$$(at\ w \wedge \Box(at\ w \supset \neg B)) \rightsquigarrow after\ w \tag{10}$$

Intuitively easy steps like these are often left out. Note that in [OL82] only deals with liveness. All safety properties used are assumed or proven by inspection of program code, i.e informally.

In contrast to the semi formal proofs of [OL82], rules of Natural Deduction and Hoare Logic are easily used in completely formal proofs. Because of this they are more easily implemented in Agda.

```
boolean p;
a: cobegin
     b: while p do c: ...
   []
     d: ...
```

Figure 12: An incomplete program symbolising something arbitrary running concurrently with a while loop.

4 Result

This section presents the methods developed for proving safety and liveness properties of concurrent programs. The actual result lies in the methods themselves and not the exact implementations presented here. Two conceptually and practically different methods are described, and are abbreviated by Proof by Construction and Proof by Control Flow. Both methods use implementations of CPL and ELTL in Agda though slightly different. These are presented in section 4.1. In addition to this a direct implementation of Hoare Logic in Agda is presented in section 4.2, which has an approach very similar to Proof by Construction. In section 4.3 Proof by Construction is presented and in section 4.4 Proof by Control Flow.

4.1 Program and Logic Representation

This section presents two implementations of CPL and ELTL in Agda. Both represent the same program language and logic but differ slightly practically.

4.1.1 Proof by Construction

This the representaton used in the Proof by Construction method.

Program Representation A CPL program is represented in Agda by the type `Statement`. Conceptually a statement is a piece of code in CPL that does something, for example a while statement or assigning a value. In figure 13 this data type and its constructors are presented.

```
data Statement : Set where
  assignN      : Label -> NVar -> NExpr -> Statement
  assignB      : Label -> BVar -> BExpr -> Statement
  composition  : Statement -> Statement -> Statement
  while        : Label -> BExpr -> Statement -> Statement
  cobegin      : Label -> Statement -> Statement -> Statement
```

Figure 13: The type `Statement` which represents a CPL program.

A small declaration of a computer program using the `Statement` type can be seen in figure 14. This is the same program as presented in 9.

```
a = l zero
b = l (suc zero)
c = l (suc (suc zero))
d = l (suc (suc (suc zero)))
e = l (suc (suc (suc (suc zero))))

p = bvar zero
x = nvar zero

prog = cobegin a
         (assignB b p (constB false))
         (while c (rvarB p)
           (assignN d x ((rvarN x) N+ (constN (suc zero)))))
```

Figure 14: The type `Statement` which represents a CPL program.

Among the arguments to the constructor are also expressions and variables, denoted by `NExpr`, `BExpr`, `BVar` and `Nvar`. Part of the `NExpr` type can be seen in figure 15. The constructors represent different expressions that evaluate to a `NExpr`; the same goes for the boolean expressions. Note that the constructors represent natural expressions such as plus and minus as well as evaluating a constant and a variable, `constN` and `rvarN`. The variables are merely a wrapper type for explicit use of variables.

```
data NExpr : Set where
  _N+_   : NExpr -> NExpr -> NExpr
  _N-_   : NExpr -> NExpr -> NExpr
  _N*_   : NExpr -> NExpr -> NExpr
  constN : N -> NExpr
  rvarN  : NVar -> NExpr
```

Figure 15: The type representing expressions evaluating to natural numbers.

Logic Representation The data type `Props` in figure 16 is a representation of the ELTL propositions in Agda. The constructors are mostly direct implementations of ELTL syntax with some exceptions. `var` for example allows reasoning about boolean variables as `Props`.

```
data Props : Set where
  patom            : N -> Props
  var              : BVar -> Props
  _<=>_          : Props -> Props -> Props
  at inside after : Label -> Props
  _~>_             : Props -> Props -> Props
```

Figure 16: A type in Agda representing ELTL formulae

As an example we can express the formula $p \rightsquigarrow q \wedge r$, where p, q and r are propositional atoms. This can be seen in figure 17.

```
p = patom zero
r = patom (suc zero)
q = patom (suc (suc zero))
formula = p ~> (q ^ r)
```

Figure 17: An example of how to declare ELTL formulae using the `Props` type.

4.1.2 Proof by Control Flow

The Proof by Control Flow uses a representation of ELTL and CPL that differs from the one used in Proof by Construction. It is briefly presented here.

Program Representation The representation for the Proof by Control Flow method is very similar to the Proof by Construction representation. The difference is that the data type `Statement` has been divided into two data types, one also called `Stm` (short for statement) and one called `Seg` (short for segment). These data types are defined in figure 18. Note that `Stm` constructs the actual assignments and `Seg` includes the constructors for building a program with labels. We separate statements from segments in order to avoid having to reason about actual assignment when talking about control being at a certain location in the program. The reasoning about whole blocks of code, i.e concatenated segments, are called blocks.

```
data Stm : Set where
  <_:=n_> : NVar -> N -> Stm
  <_:=b_> : BVar -> ExpB -> Stm

data Seg : Set where
  seg : Label -> Stm -> Seg
  block : Label -> List Seg -> Seg
  par : Label -> List Seg -> Seg
  while if : Label -> ExpB -> Seg -> Seg
```

Figure 18: The data types for statements and segments

As an example, the program in figure 9 using this representation can be seen in figure 19.

```
program = prog
(block (s 0)
  (
    seg (s 1) < vB ”x” :=b bool true >::
    par (s 2)
    (
      seg (s 3) < vB ”x” :=b bool false >::
      while (s 4) (bVar (vB ”x”))
      (
        seg (s 3) < vN ”y” :=n nat 1 >::
      )::
      []
    )::
    []
  )
)
```

Figure 19: The representation of the program in figure 9 for the implementation in Proof by Control Flow.

Logic Representation Since there are differences in the implementations of CPL, naturally there will be differences in the implementation of ELTL as well. The data type for Proof by Control Flow's implementation of ELTL, simply called LTL, can be seen in figure 20.

```
data LTL : Set where
  _ˆ’_ _v’_    : LTL -> LTL -> LTL
  _=>_         : LTL -> LTL -> LTL
  _˜>_         : LTL -> LTL -> LTL
  at in after : Label -> LTL
  _==n_       : NVar -> Nat -> LTL
  _==b_       : BVar -> BVar -> LTL
  isTrue      : BVar -> LTL
```

Figure 20: The definition of the type LTL in the implementation Proof by Control Flow with all its constructors.

4.2 Sequential Safety

A convenient basis for reasoning about sequential safety properties is *Hoare Logic* as discussed in 2.6. In this section we show how this has been encoded in Agda and how it can be used to prove sequential safety properties. For simplicity's sake the `Label` type used in the program structure has been left out; it is not needed for the Hoare Logic rules used.

4.2.1 Implementation of Hoare Logic in Agda

To represent Hoare Logic in Agda the Hoare triple has been implemented as a data type. The constructors to this type are the Hoare Rules discussed in section 2.6. In figure 21 the declaration of the Hoare triple data type can be seen.

```
data _[_]_ : Props -> Statement -> Props -> Set where
```

Figure 21: The declaration of the data type representing a Hoare Triple in Agda.

An instance of this type has the same meaning as the Hoare Triple $P\,\{Q\}\,R$. The pre- and postconditions are LTL formulae and the program here is represented by the data type used for the program representations in the Proof by Construction.

To reason about Hoare triples, we need the associated rules. We saw earlier that constructors of data types in Agda can be seen as rules. When applying rules we create new Hoare triples; by chaining the rules together by calling the different constructors we create proofs that these triples hold. Some of the constructors are presented below.

The Rules The first rule of Hoare Logic, axiom of assignment D_0, can be seen as a constructor to a dependent type in figure 22. Note that the rule implemented here only deals with assignment of natural numbers, as denoted by the n in `D0-n`. The constructor takes three arguments: an LTL formula `p`, a variable `n` and a natural number expression `e`. The constructor then returns a type which states that: with the precondition `p`, the postcondition `p` with all occurrences of `e` replaced by `n` will hold after assignment of `e` to `n`. The only special remark of this data type is the function `replace p e n` used in the post condition on line 5. This function replaces all occurrences of `e` with `n` in `p`. A further explanation of this function is presented in section 4.2.1.

```
1   D0-n : (p : Props) ->
2                              (n : NVar) ->
3                              (e : NExpr) ->
4                              --------------
5                              p [ assignN n e ] (replacePN p e n)
```

Figure 22: A constructor to the Hoare Triple data type representing the Axiom of Assignment.

Another rule, the rule of composition D_2, is presented in figure 23. This implementation is very straightforward. The constructor takes two other elements of the Hoare Triple type where the post condition of the first matches the precondition of the second. In accordance with the associated Hoare Triple rule the constructor returns a new Hoare Triple where the program is the composition of q_1 and q_2.

```
D2   : {p r1 r : Props} ->  {q1 q2 : Statement} ->
                            p [ q1 ] r1 ->
                            r1 [ q2 ] r ->
                            --------------
                            p [ composition q1 q2 ] r
```

Figure 23: The Rule of Composition implemented as a constructor to the Hoare Triple type.

For a full set of the rules and the entire Hoare Triple data type and module see Appendix A.1

Replace Functions In order for the Axiom of Assignment to be encoded we have implemented functions that replace the occurrences of expressions in LTL formulae. These functions recursively replace these expressions. In figure 24 the definitions of the functions can be seen.

```
replaceEqualN : NExpr -> N -> N -> NVar -> NExpr
replaceNN : NExpr -> NExpr -> NVar -> NExpr
replaceBN : BExpr -> NExpr -> NVar -> BExpr
replacePN : Props -> NExpr -> NVar -> Props
```

Figure 24: These are the replace functions used in the Hoare Triple data type.

The different functions replace expressions in different types. Because these types are nested and some depend on other the different functions call each other. For example if a `NExpr` is found in a `Props` the `replaceBN` function will be called.

4.2.2 Proof for Simple Sequential Program

As an example of the application of Hoare Triples consider the small sequential program of figure 25 [Hoa69]. This program finds the quotient, q, and remainder, r, of x/y. Inspection of the code verifies that upon termination

$$r < y \tag{11}$$

holds. From arithmetics it is known that the nominator x of x/y can be recovered from the quotient and remainder by the formula

$$x = r + yq. \tag{12}$$

Expressed in Hoare Triples these safety properties are

$$\mathit{true}\ \{Q\}\ \texttt{y > r}$$

$$\mathit{true}\ \{Q\}\ \texttt{x = r + yq}$$

The formal proof of these properties presented by Hoare in [Hoa69], is done in Agda using the implemented rules. Arithmetic parts of the proof are considered as assumptions. Steps 1–5 in the formal proof can be seen in figure 26. Included are also variable declarations to simplify the arguments.

The first step is an arithmetic assumption, namely that $x = x + y \cdot 0$ On line 8 we can see that the constructor `D0-n` has been called. The arguments are the previous assumption, the expression r and variable x. On line 9 we see a similar procedure. Then rule `D1-b` is used to change the precondition so that we then can apply `D2` to step 4 and 3. So far we have created the Hoare Triple of step 5 in the formal proof.

The remainder of the proof is done in a similar manner to finally arrive at step 12 and our sought after Hoare Triple. Observe that the applications of the rules are identical to their formal textual counterpart.

```
integer x, y, r, q;
a: r := x;
b: q := 0;
c: while  y <= r  do
    d:  r := r-y;
    e:  q := q+1
```

Figure 25: An example program in CPL calculating the quotient and remainder of $\frac{x}{y}$.

```
 1  -- Variables and constants
 2  x = (nvar N0)
 3  X = rvarN (nvar N0)
 4  n0 = constN zero
 5  n1 = constN (s zero)
 6
 7  -- Proof
 8  L1  = axiom (true -> (beval (static X N= (X N+ (Y N* n0)))))
 9  L2  = D0-n (beval (static X N= (X N+ (Y N* n0)))) r X
10  L3  = D0-n (beval (static X N= (R N+ (Y N* n0)))) q n0
11  L4  = D1-b L1 L2
12  L5  = D2 L4 L3
```

Figure 26: A proof of the values of the variables in the program 25 using an implementation of Hoare Logic in Agda. Step 1-5 can be found on lines 8-12.

4.3 Proof by Construction

As an application and explanation of Proof by Construction we consider the previously discussed program in figure 9. A proof that this program terminates was presented in the form of a lattice in figure 10. In this section we will show how Proof by Construction can be used to prove the first step in that lattice; which is shown in equation (13).

$$at\ b \wedge at\ c \rightsquigarrow after\ b \wedge \neg p \tag{13}$$

To show this we formally express that the program satisfies the proposition

$$at\ b \rightsquigarrow (after\ b \wedge \neg p). \tag{14}$$

As discussed this follows from the Atomic Assignment Rule and we express this in Agda. We also show that

$$at\ b \rightsquigarrow (after\ b \wedge \neg p) \vdash (at\ b \wedge at\ c) \rightsquigarrow (after\ b \wedge \neg p)) \tag{15}$$

holds. Notice that the sequent is the same property our program satisfies by AAR. Furthermore the right hand side is the property in equation (13); the one we want to prove. If we prove that (15) holds we can conclude that since our program satisfies the sequent it must satisfy what the sequent entails. We start by showing the syntactic satisfaction relation of equation (15).

4.3.1 Syntactic Satisfaction Relation

The Syntactic Satisfaction Relation is implemented as a type in Agda and denoted by $\vdash$. In figure 27 the definition and a selection of the constructors are shown. The first step in proving equation (15) is to show that equation (16) holds.

$$at\ b \wedge at\ c \vdash at\ b. \tag{16}$$

This is done in Agda using the type $\vdash$; the proof can be seen in 28. It is the usual way of constructing proofs in agda.

```
data _|-_ : Props -> Props -> Set where
  refl :                              (p : Props) ->
                                      --------------
                                      p |- p

  ^-e1   : {p q r : Props} ->         p |- (q ^ r) ->
                                      p |- q

  nd     : {p q : Props} ->           p |- q ->
                                      --------
                                      T |- (p => q)

  hs : {p q r s : Props} ->           p |- (q => r) ->
                                      p |- (r => s) ->
                                      ---------------
                                      p |- (q => s)

  T-i : {q : Props} ->                T |- q ->
                                      (p : Props) ->
                                      --------------
                                      p |- q

  ~>-i   : {p q : Props} ->           p |- q ->
                                      --------------
                                      T |- (p ~> q)
```

Figure 27: The definition of our Syntactical Satisfaction Relation, denoted by the the single turnstile, taken from the LTL module.

First the rule `refl` is applied to $at\ b \wedge at\ c$, which returns the relation that this syntactically entails itself. The second step is applying `∧-e1`, and the proof is finished.

```
S1 : (at b ^ at c) |- at b
S1 = ^-e1 (refl (at b ^ at c))
```

Figure 28: Sample usage of and-elimination.

The rest of the steps can be seen i figure 29. The only unusual step is `S2` were the rule `⇝-i` is applied. The rule states that if we can show that a proposition implies something, it also leads to it. This is intuitively verified rather easily but a formal proof is shown in appendix A.2, but has been left out for the sake of simplicity. The other steps are regular propositional calculus rules and can all be seen implemented in figure 27.

```
S2 : T |- (at b ^ at c) ~> at b
S2 = ~>-i S1

S3 : at b ~> (after b ^ not p) |- at b ~> (after b ^ not p)
S3 = refl at b ~> (after b ^ not p)

S4 : at b ~> (after b ^ not p) |- (at b ^ at c) ~> at b
S4 = T-i S2 S3
```

Figure 29: The rest of the proof for (15).

We have now proven that equation (15) holds and the next step is proving that our program satisfies (14).

4.3.2 Semantic Satisfaction Relation

Showing that a program satisfies the sequent of a syntactic entailment is not done by construction, but with functions. The function that does this is presented in its entirety in 30. Its arguments are a program and a syntactical satisfaction relation. It checks if the program satisfies the sequent by calling a recursive search algorithm: `verify`. The function can be seen as the mapping of a program and a proposition to the set of all programs that satisfy this proposition.

```
|= : {p q : Props} -> Statement -> (p |- q) -> Bool
|= {p} {q} prog _ = verify p prog
```

Figure 30: The declaration of the semantic satisfaction relation function.

The function `verify` and all its cases are presented in figure 31. Note that the sequent in an element of the type ⊢ is not represented by a list of propositions, but conjugations of propositions. The verify function therefore pattern matches on ∧ statements, not on a list. To verify the individual statements we call the helper function `verify'`.

Some of the cases and the definition of this function can be seen in figure 32. These cases represent the rules introduced in OL regarding programs. For example we have the Atomic Assigment Axiom and Atomic Assigment Rule. Each is rule has another helper function which searches the program and makes sure that the statements match. In our program we have `assignB b p (constB false)`. This has the label `b`. If we in equation (14) would swap b with for example c the function `verify-aar` would return false. This would also be the case if we used another program where b was not the atomic assignment of false to `p`. Note that each rule in OL that explicitly refer to a program have their own help function and the the meaning of the rule is encoded in the pattern matching.

We have now presented a way of checking if the program in figure 9 satisfies the sequent in equation (15). The last step of the proof, using the function and the proof of (15), can be seen in figure 33. The proof is complete because the function returns true.

```
verify : Props -> Statement -> Bool
verify (p ^ q) prog = (verify p prog) and (verify q prog)
verify p prog = verify' p prog
```

Figure 31: The verify function splits a sequent into its composite propositions.

```
verify' : Props -> Statement -> Bool
verify' T _ = true
verify' ((at (l a)) ~> (after (l a'))) prog =
    if (a == a') then
      (verify-aaa (Label a) prog)
    else
      false
verify' ((at (l a)) ~> ((after (l a')) ^ (b = e)) prog =
    if (a == a') then
      (verify-aar (Label a) prog)
    else
      false
verify' _ _ = false
```

Figure 32: The helper function for `verify`. The different cases mainly represent different rules regarding programs, taken from OL.

```
proof-valid = (prog |= prog S4)
```

Figure 33: The final step in proving that the program in figure 9 satisfies the property in equation 15.

4.4 Proof by Control Flow

This section will explain how one would go about performing the method of Proof by Control Flow with the program in figure 19 in mind, i.e. the program that the proof lattice is about. All code can be found on our GitHub [Git]

```
integer x; boolean p;
a: cobegin
    b: p:=false;
   []
    c: while p do
        d: x:=x+1;
coend;
```

Figure 34: Figure repeated for ease of access. Originally figure 9.

4.4.1 Connecting the Representations - Actions and Triples

Since the representations of CPL and ELTL have been defined in section 4.1.2, a connection between them has to be established. The ELTL representation already has constructors which are tied to the control of the program, but an even stronger connection between the program and the logic is defined in the module "Translator". This module translates a program into a list of logical statements. These statements consists of `Triple`, which are based on `Action` and ELTL formulae. There is also a function, `translate`, that can transform whole programs into a list of such triples.

Action The definition of this type is a reference to what happened when the control of a program was moved. E.g. if control was moved from being at a segment that assigns a value to a variable, to being after the segment. Then the constructor of `Action` that would be used as the reference for this assignment would be `assign`. The rest of the constructors are defined in figure 35 and are used analogously.

```
data Action : Set where
  assign : Action
  seq    : Action
  par    : Action
  while  : Action
  if     : Action
  flowA  : Action
```

Figure 35: Data type for `Action`.

Triple With `Actions` defined, the data type `Triple` can also be defined. The constructor is built up as a Hoare Triple by having a pre- and postcondition as input in the form of ELTL formulae as well as an Action to tie them together:

$$< \mathsf{LTL}_1 > \mathsf{action} < \mathsf{LTL}_2 >$$

The definition of the triple is: let *pre* and *post* be ELTL formulae and *a* be an Action, then if *pre* is satisfied as being a true state of the program and control is about to move in a way defined by *a*, then *post* will be the resulting state.

E.g. if we look at the segment *s 1* in the program:

$$\texttt{seg (s 1) <(vB ''x'') :=b (bool true) >,} \tag{17}$$

i.e assigning *true* to the variable *x*, then the resulting `Triple` that the assignment can be transformed into is:

$$\texttt{<at (s 1) >assign <(after (s 1))} \wedge \texttt{(''x'' ==b true) >.} \tag{18}$$

```
< at (s 0) > seq < at (s 1) > ::
< at (s 2) > par < at (s 3) ˆ' at (s 4) > ::
< at (s 3) > assign < after (s 3) ˆ' ˜ (isTrue (vB "x")) > ::
< at (s 4) ˆ' [] (isTrue (vB "x")) > while < [] (in' (s 5)) > ::
< at (s 4) ˆ' [] (˜ (isTrue (vB "x"))) > while <
after (s 4) ˆ' [] (˜ (isTrue (vB "x"))) >
::
< at (s 4) > while <
(at (s 5) ˆ' isTrue (vB "x")) v'
(after (s 4) ˆ' ˜ (isTrue (vB "x")))
>
::
< at (s 5) > assign < after (s 5) ˆ' (vN "y" ==n 1) > ::
< at (s 1) > assign < after (s 1) ˆ' isTrue (vB "x") > ::
< after (s 1) > flowA < at (s 2) > ::
< after (s 2) > flowA < after (s 0) > ::
< after (s 3) ˆ' after (s 4) > flowA < after (s 2) > ::
< after (s 5) > flowA < at (s 4) > :: [ ]
```

Figure 36: A translation of the program in figure 19 into triples.

Note that the triple describes an assignment in segment 1 that moved control to after segment 1 and set the value of x to *true*.

4.4.2 Translating a Program Into Triples

The main idea is to use the function

$$\textbf{translate} : \textbf{Program} \rightarrow (\textbf{List} \times \textbf{triple}) \tag{19}$$

which is the function that takes a program and returns a list of triples. E.g. passing the program from figure 19 would translate the program into a the list of triples seen in figure 36.

For more technical details of how `translate` and its assisting functions work, see Appendix B.1.

4.4.3 Rule Set and Truths

Now that a program, logic and a connection between them, in the form of triples, has been defined. What is left to define is the actual construction of proofs before verifying them is possible. A proof constructed with this method does not have to be correct, since the verification will be made later by a function.

The Rules Set consists of three different rule types that will be defined here: *program rules* (`progR`), *LTL rules* (`LTLR`) and *custom rules* (`customR`).

Program rules (progR) are associated with an `Action`. These rules together with an ELTL formula as a precondition work as a key to unlock the postcondition, a new ELTL formula, of a triple.

E.g. observe the triple (first presented in (18)):

$$\langle \texttt{at (s 1)} \rangle \texttt{assign} \langle \texttt{(after (s 1))} \wedge \texttt{(''x'' ==b true)} \rangle.$$

Here, if we have `at (s 1)` and the rule `assRule` would be applied to it, then `(after (s 1))` $\wedge$ `(''x'' ==b true)` would be guaranteed. This postcondition can now be used as a precondition together with another rule, in order to unlock the next postcondition. This is how you chain the rules together to conduct a proof.

LTL rules (LTLR) are rules describing what modifications that can be made to ELTL formulae. All of these are variations of introduction and elimination rules.

```
data ProgRule : LTL -> Action -> Set where
  assRule   : (l : LTL) -> ProgRule l assign
```

Figure 37: The definition of the data type `ProgRule`, which relates how control moves through the program by using the type `Action`, together with the first rule, the atomic assignment rule.

Custom rules (customR) are used when the other rules are not enough and work as user supplied axioms. A situations where one would need such an axiom, would be when a safety property has to be proven in order to proceed. For example, the while loop in the program in figure 19 requires a safety property stating that at some point the boolean variable will always remain false in order to exit it.

True ELTL formulae are the postconditions that rules are applied to which are represented as a data type that contains **List** ×**ELTL**, called `Truths`. Since this data type only should contain ELTL formulae that are true in the current state, we have to make sure that no contradicting pairs of ELTL formulae can exist in `Truths`. This is managed by updating the list after every time a rule is applied, which causes the state to change. This update works according to:

1. Before adding the new truths, remove all previously true ELTL formulae except those the form "$\Box$ . . . ", "at. . . ", "in. . . " or "after. . . " since these are guaranteed to still be true after application of a rule.
2. Append the list of new truths to the updated list of previous truths.

Therefore, if an element in the truth is (20)

$$\texttt{vB ''x'' ==b true} \tag{20}$$

we can assure that (21) is not, through the rules above.

$$\texttt{vB ''x'' ==b false} \tag{21}$$

4.4.4 Constructing Proofs

All that is left in order to constuct proofs is to tie the above defined types together. The representation of an actual proof is very similar the proof latices mentioned in section 2.5.1. There are two data types used for construction of the proofs, `Proof` and `ProofStep`. `Proof` is built as **List**×**ProofStep**, and a `ProofStep` is either the application of a rule or the construction of *branches*. Branches are, in simple terms, sub-proofs, i.e. **List**×**ProofStep**, that represents the branching in the proof lattice [OL82]. In order for a branch to be correct, both sub-proofs have to contain the same truths in their final states.

In 38 we will show a small example of how to take a few steps towards constructing a proof of termination. This small example displays the first steps from the example program from figure 19. Each proof step, `pStep`, in the example takes a rule (LTL rule or program rule), which in turn is applied to an ELTL formula said to be in the `Truth`. Since we are trying to prove termination, i.e. `at (s 0)` $\Rightarrow \Diamond$ `after (s 0)`, the initial truth will be `at (s 0)`.

4.4.5 Verifying the Proofs

With the data type for proofs implemented, the only thing that remains is the actual verification, to find out whether the proof is valid or not. This is done by a proof checking function. The function is defined in figure 39 and might be quite hard to grasp at a first glance.
The function itself takes a program, list of custom transition relations, a proof, a goal (ELTL) and an initial truth as input. Running the function with a set of input then yields a result, called `ValidProof`, that is either `yes` or `no`, accompanied with a brief message telling the user where the proof was incorrect. For a more technical presentation of the proofcheck function see appendix B.2

```
termProof : Proof
termProof = proof prf
  where
    step1 = pStep (progR (seqRule (at (s 0))))
    step2 = pStep (progR (assRule (at (s 1))))
    step3 = pStep (ltlR (^-e1 ((after (s 1)) ^' (isTrue (vB "x")))))
    step4 = pStep (progR (atomLive (after (s 1))))
    step5 = pStep (progR (parRule (at (s 2))))
    step6 = pStep (ltlR (exp-^ ((at (s 3)) ^' (at (s 4)))))
    step7 = pStep (progR (assRule (at (s 3))))
    ...
```

Figure 38: The definition of the data type `ProgRule`, which relates to how control moves through the program by using the type `Action`, together with the first rule, the atomic assignment rule.

```
proofCheck :    Prog -> List TransRel -> Proof -> LTL -> Truths ->
                ValidProof
proofCheck pr rels prf g tr =   proofCheck' ((translate pr) List.++
                                rels) prf g tr
```

Figure 39: The definition of the proof checking function.

In our case, the program would be `program` from figure 19, the list of custom transition relations would be empty, the proof would be the entire proof of which a part can be seen in figure 38, the goal would be `at (s 0)` $\Rightarrow \Diamond$ `after (s 0)` and the initial truth would be an empty list (since the actual initial truth is given in the goal).

4.4.6 Verifying Safety Properties

Safety properties are used in order to, among other things, prove that it is possible to exit a while loop. It is necessary to know that a boolean expression will always remain false after some point in time in order to ensure exiting. As mentioned in the article by Owicki and Lamport [OL82], such properties are usually proven by observing the actual structure, or code, of the program and realising that it can easily be deducted by reasoning about the possible states of the program.

In the case of the program in figure 9 it is easy to see that once the boolean p has been set to false it will not be changed by any other subsequent assignment, which implies that it will always be false after that point. Since the implemented set of rules lack any introduction of the $\Box$ operator, these properties cannot be generated from the program by translation. Instead custom rules must be created for these properties and a function is used to verify that the program never breaks this property. To read more about what the safety verifier can do, and also see some more technical aspects of it, see appendix B.3.

Note that the safety verifier is not completely functional with the rest of the implementation. It is possible to abuse a combination of custom rules and the safety verifier in order to construct an incorrect proof that will still type check as correct.

5 Discussion

In this section the difference between the two implemented methods are discussed. Furthermore the chosen limitations of these methods are presented. Discussion about future work follows in section 10.

5.1 Proof by Construction vs Proof by Control Flow

The two methods that have been developed are quite different both in implementation and usage. This section discusses their differences and advantages.

5.1.1 Differences

Constructing a proof with the proof by construction method is quite similar to doing the proofs with pen and paper. However a strict formality and a lot of massaging of ELTL-formulae is needed to make the constructors match. This makes the proof by construction method very tedious and time consuming.

With the proof by control flow method the rules are used to follow the flow of control through the program. This leads to shorter more concise and often intuitive proofs. However, since the proof has a state and requires the repeated application of a step function to traverse the flow you need to keep track of this state mentally. Learning this new system can be a big step from just doing formal pen and paper logic.

5.1.2 Similarities

Both systems use a function to verify the satisfaction relation. The proof by control flow method uses it to verify the applications of rules to ensure that they coincide with the actual program. The construction method however only uses it as a look-up function to see that the statements referred to in the sequent are in the program.

5.1.3 External Verification

Constructive proving using Agda has the advantage that given correct rules the proofs presented are also correct. When using functions on the other hand it can be difficult to prove that the functions work as intended. The code presented has not been verified externally. Since all proof methods use some sort of functions it is not possible to guarantee their correctness. This is a major flaw and further development to create completely constructive proofs is a future prospect, see section 6 for further discussion.

5.2 Limitations

Here we present some of the limitations to the proof methods and the reason for this.

5.2.1 Program Representation and Reasoning

The CPL programming language from section 2.3 that we implemented in Agda is very small, and lacks some common programming constructs, notably:

- *if* statements
- *atomic* blocks

If statements are very convenient but are not needed for the programs we have reasoned about, using our presented methods. Thus they have not been implemented. They could possibly be implemented easily in CPL, and use special cases of the rules presented in [OL82]. We already reason about *while* statements, and *if* statements are very similar in structure.

Atomic blocks such as a swap would be quite interesting to reason about. They especially become handy when discussing liveness in conjunction with mutex. The main reason for not implementing either of these structures were time constraints.

5.2.2 Operational Semantics

The implementation of CPL or ELTL allows the use of operational semantics. Programs are never executed and logical statements are not actually evaluated. This does not imply that it is not possible to reason about expressions and logical formulae. For example the types `NExpr` and `BExpr` can be used to express and reason about properties such as $(y + 1) \times (z + 3) > 2$. For the implemented methods this is fine because safety properties are only postulated, not proven. The liveness properties discussed can still be proven without the operational semantics; and thus they were not needed.

5.3 Safety for Concurrent Programs

A constructive method for proofs of safety properties of concurrent programs was never implemented. Usually proving safety properties is done by finding an invariant property i which guarantees the desired safety property. If you prove that the (i ⊃ □ i) holds, i.e. that if it holds at some point it will always hold, you have proven the safety property.

5.3.1 Mutual Exclusion

Many important properties of concurrent programs are safety properties, most commonly mentioned is mutex. The inability to formally prove these limits the usefulness of our methods. Often proving mutex involves finding some invariant property i which guarantees mutex and then proving that our program will at some point satisfy i.

5.3.2 Visual Verification

The invariance claims in OL were "proven" by looking at the program and each statement individually. No formal rules were ever introduced. The claim were often the invariance of the value of a variable, which holds when the program never assigns the variable to another value. This is obviously informal and cannot easily be extended to a formal system or to more complex invariants.

6 Future Work

The time schedule limited the generality of our method. This section brings up areas were continued work is needed and what possible extensions could be implemented.

6.1 Safety

Safety is one of the most important extensions for both the methods, and is probably among the first that would be tested.

6.1.1 More Old Theory into Agda

Based on the systems we have constructed one natural extension is the implementation of some other logical systems in Agda. The papers [Lam80] and [LS84] respectively present such systems for proving safety properties. Formalising these could possible allow both methods to implement a way to prove safety properties into the libraries as an extension. This could lead to a more robust system able to prove both liveness and safety properties.

6.1.2 Verifier

The safety verifier is in its current implementation very specific. It proves the invariant needed for proof of termination for some very simple programs, e.g. 9. However some proofs need more complex invariant propositions, such as the mutex proof presented in [OL82] that includes more than a single variable. It isn't obvious how the current method could be extended to work in these cases so a different method, such as the formalisation of [Lam80] or [LS84], might be required.

6.2 Proof Construction Improvements

The proof construction is in its early stages still and there are still work to be done before it becomes efficient to use.

6.2.1 Library of Subproofs.

Constructing a proof for even a small thing, like our $\leadsto$-introduction rule from appendix A.2, requires the application of many constructors. Many patterns used in proof construction are frequently occurring and could be written as reusable proofs, like the $\leadsto$-introduction rule. A library of sub proofs like these would make the construction of proofs a lot easier.

6.2.2 Independent Development of the Rule Set

A large part of developing the proof by construction method was translating the ELTL rules and propositions of OL to the domain of propositional calculus. These rules are usable outside of Agda and it might be interesting to develop this rule set independently. The pen and paper proving method presented by OL is using lattices and making assumptions. With our rules you can do it in the same manner as a normal proof in propositional calculus.

6.3 Proof by Control Flow Improvements

While the implementation of the Proof by Control Flow method is in a usable state, improvements can still be made.

6.3.1 Proof by Control Flow as a Teaching Tool

Since the structure of the method Proof by Control Flow is not as formally tied to the actual logic as the Proof by Construction method, but tries to be more intuitive in its implementation, one possibility is to turn the method more into a tool that can be used to teach students how to construct formal proofs. The idea of how take the method in that direction is to implement a better way to manage truths by having to state what ELTL formulae that will be true after the application of a rule.

Another improvement that can be made is to make better error messages when an incorrect proof is passed to the `proofCheck` function. This would help in improving the logical reasoning of a user with lacking logical knowledge.

6.4 Ongoing Work Based on Proof by Control Flow

The upcoming master thesis [Hä] concerns the implementation of yet another method for proving concurrent liveness and safety properties using Agda. The author Johan Häggström is also partly responsible for the Proof by Control Flow method, presented in this thesis. At present this new method is intended be completely constructional, but is based on Proof by Control Flow.

The thesis also explores the possibility to automate proof construction, using the auto complete functionality in Agda. This already works for simple proofs, e.g that 4 is even, but for bigger types the complexity grows rapidly. This results in it taking too long time but it is theoretically possible.

7 Conclusion

This thesis shows that it is possible to implement methods in Agda to verify formal proofs of concurrent liveness properties as well as sequential safety properties. The proofs concern a small programming language, CPL. The proof mechanisms for the concurrent liveness properties are based on [OL82] and for sequential safety [Hoa69].

7.1 Formal proofs

The formal systems of Hoare Logic and propositional calculus were easily implemented in Agda. Manually constructed proofs can be directly translated using Agda types; with constructors completely analogous to the formal rules. The resulting proofs can be completely constructional.

However as soon as interference is taken into consideration the situation becomes complicated. The proof system presented in [OL82] are not formal enough to express in Agda. As a result, formalisation of this was done. See for example the formal proof of `~>-i` in Appendix A.2 and the discussion on the WER in section 3.1. Even with our modern tool set (Agda) the same difficulties with proving concurrent properties as back in the 80's remain. In [Mis17], for example, a discussion on the subject can be found. This thesis takes no claim in having solved any of the problems with interference and concurrency.

7.1.1 Formal proofs are long

The methods implemented in Agda does not permit formal proofs to become any less tedious. Formal proofs are even when machine constructed very long. This is by their nature, formality requires precision.

7.2 LTL as a Basis for Logic

Introduced to the subject of properties of concurrent programs, LTL is not trivially expressive enough. The work in this paper however reveals that a rule system can be built around the close relative ELTL, presented in section 2.4.2, to formally prove liveness.

Bibliographic Notes

[OL82] This paper presents a convinient logic for reasoning about concurrent programs. Many of the rules and concepts presented here have been utilised by us, which helped much in our work. [Hoa69] The logic in this famous paper by Tony Hoare has been implemented as types in Agda. These types have then been used to construct proofs for safety properties of sequential programs. [Cai15] In this bachelor thesis by Leran Cai we found a structure for proving things in Agda. The topic of this report is Propositional Calculus which means that the topic itself was not relevant to us but we found a lot of value in reading this report anyway.

Slightly Newer Works

The topic of this report is still a widely researched topic. Some recent works on the topics are e.g. [DSNB16] handling correctness of concurrent data structures, an overview of the formal methods at Intel [Har10]. We encourage any reader that have any doubt to look into any of these [Mis17, KGN$^+$09, RS13, NLWSD14, Neu14, Sel16].

Historical Importance

[AHP96] This paper is the reason to why this project came to be. Therefore, it is of great historical importance to the project.

Agda Theory

Here are papers for the interested, handling theory of - and Agda itself [Nor09, Nor07]. See also the Agda wiki [Wik17]. An undocumented implementation of Hoare Logic in Agda can be found here [Ike11], it was not used by us.

References

[AHP96] Jørgen H Andersen, Ed Harcourt, and KVS Prasad. A Machine Verified Distributed Sorting Algorithm. *BRICS Report Series*, 3(4), 1996.

[Ato] Atom. Atom v1.16.0. `https://atom.io/`, accessed; 11th of May 2017.

[BA06] Mordechai Ben-Ari. *Principles of Concurrent and Distributed Programming.* Addison-Wesley, 2nd edition, 2006.

[Cai15] Leran Cai. *Formalising the Completeness Theorem of Classical Propositional Logic in Agda.* Bsc (hons) thesis, The University of Nottingham, 2015.

[Coq92] Thierry Coquand. Pattern matching with dependent types. In *Informal proceedings of Logical Frameworks*, volume 92, pages 66–79. Citeseer, 1992.

[Dij65] Edsger Wybe Dijkstra. Solution of a Problem in Concurrent programming control. *Communications of the ACM*, 8(19):569, 1965.

[DSNB16] Germán Andrés Delbianco, Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. Concurrent data structures linked in time. *arXiv preprint arXiv:1604.08080*, 2016.

[Git] GitHub. GitHub repo for the Agda code. `https://github.com/alps-chalmers/agda101`, 12th of May 2017.

[GMW79] Michael Gordon, Robin Milner, and Christopher Wadsworth. Edinburgh lcf: a mechanized logic of computation, volume 78 of lecture notes in computer science, 1979.

[GNU] GNU. Emacs 25.2. `https://atom.io/`, accessed; 11th of May 2017.

[Har10] John Harrison. Formal Methods at Intel — An Overview. Second NASA Formal Methods Symposium NASA HQ, Washington DC, 2010.

[Hoa69] Charles Antony Richard Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–580, 1969.

[HR04] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning About Systems.* Cambridge University Press, New York, NY, USA, 2004.

[Hä] Johan Häggström. Proof Checker for Extended Linear Time Temporal Logic Proofs About Small Concurrent Programs. Master's Thesis, Department of Computer Science, Chalmers University of Technology.

[Ike11] Daisuke Ikegami. Implementation of Hoare Logic using Agda. `https://github.com/IKEGAMIDaisuke/HoareLogic`, 2011. Github Repo, accessed; May 9th-2017.

[KGN$^+$09] Roope Kaivola, Rajnish Ghughal, Naren Narasimhan, Amber Telfer, Jesse Whittemore, Sudhindra Pandav, Anna Slobodová, Christopher Taylor, Vladimir Frolov, Erik Reeber, et al. Replacing Testing with Formal Verification in Intel® CoreTM i7 Processor Execution Engine Validation. In *International Conference on Computer Aided Verification*, pages 414–429. Springer, 2009.

[Lam80] Leslie Lamport. The 'Hoare logic' of concurrent programs. *Acta Informatica*, 14(1):21–37, 1980.

[Lam84] Leslie Lamport. An axiomatic semantics of concurrent programming languages. *LogicsandModels ofConcurrentSystems, volume13ofNATOASISeriesF: ComputerandSystemSciences, blzn77–122. SpringerVerlag, Berlin*, 1984.

[Lam94] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):872–923, 1994.

[LS84] Leslie Lamport and Fred B Schneider. The"Hoare Logic"of CSP, and All That. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 6(2):281–296, 1984.

[Mil89] Robin Milner. Communication and concurrency. international series in computer science, 1989.

[Mis17] Jayadev Misra. Bilateral Proofs of Safety and Progress Properties of Concurrent Programs. *arXiv preprint arXiv:1704.01814*, 2017.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, i. *Information and computation*, 100(1):1–40, 1992.

[Neu14] René Neumann. Using Promela in a fully verified executable LTL model checker. In *Working Conference on Verified Software: Theories, Tools, and Experiments*, pages 105–114. Springer, 2014.

[NLWSD14] Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. Communicating state transition systems for fine-grained concurrent resources. In *European Symposium on Programming Languages and Systems*, pages 290–310. Springer, 2014.

[Nor07] Ulf Norell. *Towards a practical programming language based on dependent type theory*, volume 32. Citeseer, 2007.

[Nor09] Ulf Norell. Dependently typed programming in Agda. In *Advanced Functional Programming*, pages 230–266. Springer, 2009.

[O'C16] Liam O'Connor. Applications of applicative proof search. In *Proceedings of the 1st International Workshop on Type-Driven Development*, pages 43–55. ACM, 2016.

[OL82] Susan Owicki and Leslie Lamport. Proving liveness properties of concurrent programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):455–495, 1982.

[Pra95] K. V. S. Prasad. A Calculus of Broadcasting Systems. *Science of Computer Programming*, 25(2-3):285–327, 1995.

[RS13] Sandip Ray and Rob Sumners. Specification and verification of concurrent programs through refinements. *Journal of automated reasoning*, 51(3):241–280, 2013.

[Sel16] Erik Seligman. The Formal Verification book: past, present, and future. In *Formal Verification: an essential toolkit for modern VLSI design*, 2016. University Lecture.

[SSK15] Erik Seligman, Tom Schubert, and MV Achutha Kiran Kumar. *Formal verification: an essential toolkit for modern VLSI design.* Morgan Kaufmann, 2015.

[Sti91] Colin Stirling. An introduction to modal and temporal logics for ccs. *Concurrency: Theory, Language, and Architecture*, pages 1–20, 1991.

[Wik17] Agda Wiki. The agda wiki. `http://wiki.portal.chalmers.se/agda/pmwiki.php`, 2017.

A Agda Code

In this appendix selected bits of code can be found. The entire library can be found on the [Git].

A.1 Hoare Triple Type

```
data _[_]_ : Props -> Statement -> Props -> Set where
  D0-n : (p : Props) ->
                            (n : NVar) ->
                            (e : NExpr) ->
                            --------------
                            p [ assignN n e ] (replacePN p e n)

  D1-a : {p r s t : Props} -> {q : Statement} ->
                            t |- (r > s) ->
                            p [ q ] r ->
                            ------------
                            p [ q ] s

  D1-b : {p r s t : Props} -> {q : Statement} ->
                            t |- (s > p) ->
                            p [ q ] r ->
                            --------------
                            s [ q ] r

  D2   : {p r1 r : Props} ->  {q1 q2 : Statement} ->
                            p [ q1 ] r1 ->
                            r1 [ q2 ] r ->
                            --------------
                            p [ composition q1 q2 ] r

  D3   : {p : Props} ->       {b : BExpr} ->
                            {s : Statement} ->
                            ((beval b) ^ p) [ s ] p ->
                            --------------------
                            p [ while b s ] ((~ (beval b)) ^ p)
```

Figure 40: The data type representing the rules of Hoare logic.

A.2 Proof of ⇝-introduction

In section 7.1 we used that:

$$(\vdash p \supset q) \vdash (T \vdash p \rightsquigarrow q) \tag{22}$$

here we formally prove this in agda. The function takes a proof that p ⊢ q and use this to prove

```
~>-i-proof : {p q : Props} -> p |- q -> T |- [] (p -> (<> q))
~>-i-proof {p} {q} r = ([]-i (hs (nd r) (nd (<>-i (refl q)))))
```

Figure 41: Proof of ⇝-introduction

T ⊢ p ⇝ q. We start by weakening the statement

$$(q \vdash q) \supset (q \vdash \diamond q). \tag{23}$$

We then use the natural deduction rule to create two implications.

$$(\vdash p \supset q) \wedge (\vdash q \supset \diamond q)) \tag{24}$$

We chain them using the hypothetical syllogism (hs). We then have a proof that

$$\vdash p \supset \diamond q \tag{25}$$

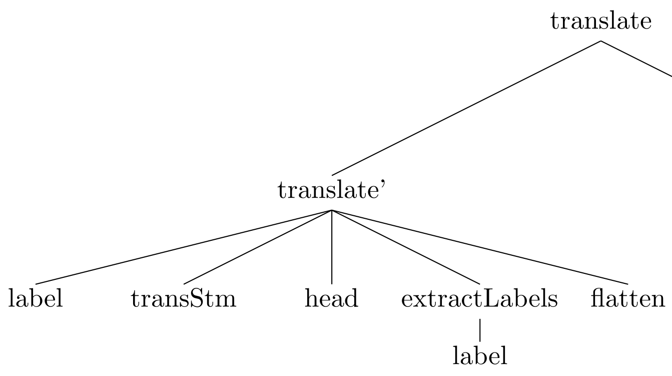

Figure 42: Function dependency for `translate`, note that label is located in the module Label.

at which point we can introduce a $\Box$, Since anything that is tautologically true is always true. At this point we have constructed

$$(\vdash \Box(p \supset \diamond q)) \equiv (\vdash p \rightsquigarrow q) \tag{26}$$

B Proof by Control Flow Definitions

B.1 Assisting functions for "translate"

The function `translate` calls upon other functions that assist it with the actual translation. A dependency graph for the `translate` function can be seen in figure 42. These functions will be explained in the order in which they show up layer by layer, also see Appendix **??** for code.

- **translate'** : **Seg** $\rightarrow$ (**List** $\times$ **triple**) - this function helps `translator` to transform segments into triples and pattern matches on the different constructors of `Seg` (see **??**) and transforms the input into their corresponding triples. If the segment given as input would contain more segments, recursive calls are made on each occurrence. E.g. if the passed segment is a `block` this would be the case.
- **transFlow** : **Seg** $\rightarrow$ (**List** $\times$ **Seg**) - this function works similarly to `translate'` except that depending on input it might call `parFlow` or `seqFlow`. Note that this function transforms the transitions between segments into triples. E.g. if a is a block and b the first segment in the block, then `transFlow` will would transform a as follows: `<at a> flowA <at b>`.
- **label** : **Label** $\rightarrow \mathbb{N}$ (see Label module) - takes a `Label` as input and returns the natural number used as the reference.
- **transStm** : **Label** $\times$ **Stm** $\rightarrow$ **triple** - transforms a statement into the corresponding triple (see (17) and (18)).
- **head** : (**List** $\times$ **A**) $\rightarrow$ (**Maybe** $\times$ **A**), where **A** is an arbitrary data type - returns the first element in a list as a `Maybe` type. I.e. if a list is passed with the element $a \in \mathbf{A}$ as the first element, `just` a is returned. If head is passed an empty list, it returns `nothing`.
- **extractLabels** : (**List** $\times$ **Seg**) $\rightarrow$ **LTL** - used by `translate'` in order to find all labels in a list of segments in case the segment to transform is a `par`. This way we can make sure that control ends up being `at` all the processes that are to be run concurrently.
- **flatten** : (**List** $\times$ (**List** $\times$ **A**)) $\rightarrow$ (**List** $\times$ **A**) - takes a list of lists and appends those lists into a single list.
- **seqFlow** : **Label** $\times$ (**List** $\times$ **Seg**) $\rightarrow$ (**List** $\times$ **triple**) - used to describe how control flows through a `block`. Let B be a `block`. If there are two or more segments left in the list of B at any call, let the first two elements be s_1 and s_2. Then if control is after s_1 and control

moves according to `flowA` it will end up at s_2. If there is only one element, s, left in B and control is after s and moves according to `flowA` it will end up after B.

- **parFlow** : (**List** $\times$ **Seg**) $\rightarrow$ **LTL** - works like `extractLabels`, except it describes how control moves out of a `par`, P. I.e. control has to be after all processes in P before it can move to after P.

B.2 Assisting Functions for "proofCheck"

- **proofCheck'** : (**List** $\times$ **TransRel**) $\times$ **Proof** $\times$ **LTL** $\times$ **Truths** $\rightarrow$ **ValidProof** ,
- **takeStep** : (**List** $\times$ **TransRel**) $\times$ **ProofStep** $\times$ **ValidProof** $\rightarrow$ **ValidProof**,
- **applyRule** : (**List** $\times$ **TransRel**) $\times$ **Truths** $\times$ **Rule** $\rightarrow$ **ValidProof**,
- **applyLTL-R** : **Truths** $\times$ **LTLRule** $\rightarrow$ **ValidProof**
- **legalApplication** : $\{\phi \in$ **LTL**, $a \in$ **Action**$\}$, (**List**$\times$**TransRel**)$\times$**Truths**$\times$**ProgRule**$_{\phi,a}$ $\rightarrow$ **ValidProof**

proofCheck' uses pattern matching on the goal that is passed to it form the original proof checking function. Depending on the goal it might send the information further or give an immediate return to `proofCheck`. If the goal would be true ($\top'$) nothing has to be done since every program given a proof satisfies true. On the contrary, if the passed goal would be false ($\bot$) there would be an immediate return of `no` since the goal is always false.

If ϕ is an LTL formula and the goal would happen to be of the form $\diamond\phi$ the return depends on whether or not the proof would reach the conclusion ϕ as one of the final truths after applying the `takeStep` function on each proof step. Similarly, if ϕ and ψ are LTL formulae and the goal is of the form $\psi \Rightarrow \phi$, it is equivalent to call `proofCheck'` with the goal $\diamond\phi$ and the initial truth ψ. That case is therefore handled as the $(\diamond \ldots)$ case.

The other cases are simply that the goal could happen to be satisfied in the initial state of the proof, else there is a catch all case that states that the goal is not true from the start. Since safety properties cannot be proved using this method, goals of the form $(\Box \ldots)$ are not handled.

takeStep mainly pattern matches on the two different types of proof steps that can be used. If the step would be a `pStep`, the fuction passes on to `applyRule` to check if the rule is applied in a legal way. If the passed step is a branch, then the two branches are checked individually by applying `takeStep` to each element in the branches and checks if the two reaches the same conclusions in their final steps. The other cases are error messages depending on what went wrong (see code in for details).

applyRule is the function that ties the application of rules into the proof checking. This function simply pattern matches on the different kinds of rules (`progR`, `LTLR` and `customR`) and calls the corresponding functions. E.g. if the rule would be a `progR`, then the function calls for `legalApplication`. If the rule would be an `LTLR`, then the function calls for `applyLTL-R`. On the other hand, if the rule is a `customR` all that has to be done is to check whether or not an element in the truths satisfies the pre condition of the custom rule. If it is satisfied, then the step is valid and the new truth will contain the post condition of the custom rule.

applyLTL-R handles the application of ELTL (elimination and insertion) rules. The function simply checks if the passed rule can be applied to the current truths. If the application is legal, the return is `yes` and the result of the applied rule is added to the new truth and if not an error message is returned with the `no`.

legalApplication handles program rules, i.e rules regarding control flow, and just like `applyLTL-r` is simple in its execution, so is `legalApplication`. The function simply checks if the given truths together with the action associated with the rule to be applied matches any of the transition relations given by the translation of the given program. If a match would be found, the step is valid and the post condition would be added to the new truths. If not, then `no` is returned with en error message.

B.3 Definitions of Safety Verifier

The implementation works by applying the strategy mentioned in section 4.4.6 by overlooking the translation of the program in order to find contradictions.

B.3.1 Evaluation of Safety Properties

In the module "Safety Verifier" is where the implementation is located. The main function of the module,

$$\_ \Rightarrow \_, \_ : \mathbf{LTL} \times \mathbf{LTL} \times \mathbf{Prog} \to \mathbf{Bool},$$

can be seen in detail in our GitHub [Git]. The idea of this function is that it will take a safety property of the form

$$\{l \in \mathbf{Label}\}, \quad (\text{after } l) \Rightarrow (\square \ldots),$$

i.e. that after a certain segment of the code, an LTL formula will always hold.

As mentioned, the method is to look for a single state that might be reached that contradicts the LTL formula. This check is different depending on the location of the segment l. If l is in a `par` segment, then all processes run in parallel with the process that l is located in have to be checked as well as all other subsequent segments. If l is in a while loop (`while` segment), then all segments appearing before l in the while loop have to be checked as well as all other subsequent segments. Also if l would be located in both a `par` and a `while`, then both methods are called. The simplest case is when l is not located in either a `par` or a `while`, then only the segments after l have to be checked.

The function $\_ \Rightarrow \_, \_$ that handles this simple concept unfortunately has to call many assisting functions that would take up to much space to mention here. All these functions are located in our GitHub [Git], where the comments explain them.