Eötvös Loránd University
Faculty of Informatics
Department of Programming Languages and Compilers

Formalizing a relational model of concurrent programs in a dependently typed environment

TDK dolgozat / Diplomamunka

Supervisor:
Ambrus Kaposi
Assistant professor

Author:
István Donkó
Computer Science MSc
2. year

Supervisor:
Melinda Tóth
Associate professor

Budapest, 2020

Abstract

Sequential programming languages have already been formalized in dependently typed programming languages, such as for example Agda or Coq, but the computer based formalization of concurrent programs is still in its early days. The goal of our research is to formalize a relational model that describes the behavior of distributed concurrent programs in a theorem prover system. Our long term goals include the verification of the material of the subject titled *"Specification and Implementation of Distributed Systems"*, which serves as a core part in the Computer Science education at Eötvös Loránd University in Budapest.

Contents

Chapter 1

Introduction

Software plays a critically important role in the life of modern societies. More or less everybody interacts with computer programs on countless occasions during our everyday lives, most of the time probably not even noticing. For example, just paying with a credit card while shopping, being able to call someone with our mobile phones, or even just as mundane tasks, as operating modern versions of basic home appliances, like washing machines or microwave ovens requires interaction with software.

During the COVID-19 pandemic in the spring of 2020 we also became more aware of how much we are relying on software for online communication to be able to do our work remotely and keep in touch with friends and family. Many of us have experienced during these times how frustrating it can be, when programs do not work as intended.

(Another clear indication of the importance and reliance on computers is the crash of the United States' unemployment system, which could not handle the huge influx of applicants due to the quarantine.)

Most of these examples are just about convenience factors, but if we take the amount of more critical scenarios into account, such as for example software running on the computer system of an airplane, or even keeping a nuclear power plant safe, we can see that programmers have an even bigger impact. (Self driving cars could also be mentioned, but machine learning is a whole another field with its own set of ethical and moral questions.) The economy of the world depends on software, the rulers of nations rely on computer systems at their disposals. To quote from a 2016 talk given by famous American software engineer, Robert C. Martin (one of the authors of the Agile Manifesto), titled "The Future of Programming", he claimed: *"Civilization depends on us."* and even went as far as to say that:

"We rule the world. The world doesn't know this yet. We don't quite know it yet. Other people believe that they rule the world, but they write the rules down and they hand them to us, and then we write the rules that go into the machines that execute everything that happens on this planet nowadays. No law can be enacted without software, no law can be enforced without software. No government can act without software, we rule the world."

I wouldn't make such an extreme claim, that developers rule the world, but I certainly agree that they play a significant role that comes with high responsibilities.

1.1 Motivation

Borrowing from the same talk, we can define the beginning of programming around the work of Alan Turing, since he was the first one, who wrote code for machines in the sense that we would recognize today. His work has undoubtedly played a crucial role in laying down the foundations of programming. Turing machines are a core part of computer science and are still taught in university courses today. In his time, he described the future possibilities of his vision with the following sentences in a lecture to the London Mathematical Society[1].

"In order to supply the machine with these problems we shall need a great number of mathematicians of ability. These mathematicians will be needed in order to do the preliminary research on the problems, putting them into a form for computation."

He stated the need for mathematicians for the precise formalization of problems. This need has since been abandoned and most of software products nowadays are just developed through trial and error processes, being patched until they pass all defined test cases, but nobody can really be assured of their correct behavior. For some types of applications, such as for example games this can be acceptable, because the worst outcome of bugs are frustrated end users, but for more critical systems, the correctness of which can decide between life and death, that is simply not enough. We need to have formal strategies to verify behaviors of programs under all circumstances instead of just observing them for the most likely situations.

There are lots of existing means for confirming the adherence of simple sequential programs to their specifications, ranging from formal verification procedures carried out on paper to contracts built into programming languages, that can be checked and enforced automatically, either via static code analysis, or during runtime by monitoring different values. Reasoning about parallel programs is a lot more complicated, but this complication also serves as an explanation for the need to do so, since concurrency is often a result of multiple systems working together, in which case it is a lot easier to make mistakes because of the unpredictable order of execution of instructions. Several different ways are known to approach formal proofs of correctness for concurrent programs. For example multiple specific methods can be seen in [2]. What we chose to base our research on is the material of the subject titled *"Specification and Implementation of Distributed Systems"* which is discussed in more detail in Section 3.1.

Our formalization does not follow the material exactly, but we tried to stay as close to the original notation as possible, as we also have intentions to later further expand this project to cater for educational usage for example as part of the practical courses. We consider this to be a valuable opportunity for creating a teaching tool that can greatly aid the understanding of the subject for students. The use of proof assistants has already been successfully introduced in several other classes[3], which helps making this idea seem quite feasible.

1.2 Outline

In chapter 2.5 we present several related works, with the most similarities and differences of the most akin article discussed in more detail. After that in chapter 3 the necessary background knowledge for the subject and the tool of our formalization is introduced. Following this, in chapter 4 our approach to the problem and some experiences encountered through our project are described. Our results are presented in chapter 5, separated into the formalization which laid down the foundation and the proofs that were built on top of it. Finally in chapter 6 we conclude our research until now and discuss the possible ways in which it could potentially be extended in the future.

1.3 Results

The main results of our work include the fully formalized version of a big core part from the original model we built our research around, which is precise enough for computer based type-checking. On top of this foundation we have developed proofs for several generic lemmas and theorems. Also included in the article is the formalization of a parallelized version of the bubble sort algorithm and the verification of some of its specification properties.

Chapter 2

Related Work

Since our goal is not the introduction of a new way to address the problem of creating correctness proofs for parallel programs, rather the adaptation of an existing system for formalized implementation, instead of discussing the theoretical methods, here we focus on a more practical approach and explore what others have achieved in the field of computer based verification.

2.1 Previous works in Agda

A very recent article [4] presents an approach similar to ours, by setting out to provide an alternative to paper based proofs by formalizing an existing semi-formal model of concurrent programming. Their results provide methods for the confirmation of certain safety and liveness properties. Since it was the most closely related to what we have done, we compare their work to ours and discuss the similarities as well as the differences. The Agda programming language served as the meta-theory, in which the formalization and proofs were constructed for both projects, but the object-theory was different. We built our system around a language similar to UNITY, a language specifically created for and entirely based on parallelism for a book titled *Parallel Program Design: A Foundation*[5] (not to be confused with the modern game engine), that has essentially no guaranteed sequentiality, while their subject was an extension of the CPL language borrowed from [6], upon which they based most of their work, which is to some extent a regular imperative sequential language that has added constructs which introduce the capability for concurrency. This difference is also made very visible by their *Proof by Control Flow* method, which implies that there is a distinct order present in the execution. Another big difference is the main focus of the projects. The aim for creating safety proofs was shared, but their other target, liveness proofs were not among our top priorities (although the fixed point property among our results can be related to it), rather we concentrated more on the formalization of general theorems as well as confirming adherence to specification, which is not included in their work. Further differences can be found by observing that the methods they utilize are in closer relation to operational semantics, while our formalization describes behavior in a manner more similar to denotational semantics. Also, some of the proofs presented in that paper are not completely constructive, some properties are only checked by functions essentially

returning Boolean results, thus no so-called witness can be used for explaining or reasoning about the results, while the results that are presented in this current paper aim to all be constructive. They also did not implement every little detail included in their proofs, some steps include the usage of postulated properties, which are either relying on paper-based proofs or intuition. Our system is fully self-contained and is built from the ground up in a way meant not to include any external dependency. Their work was related to and part of it was further expanded in [7].

2.2 Previous works in Isabelle/HOL

Another paper[8] discusses a similar topic, but uses the Isabelle/HOL proof assistant[9] built on the imperative paradigm instead of the type theory based alternatives. Their main subject is a version of the generic SIMPL programming language originally introduced in [10], that they extend in a way to support parallel composition and synchronization through shared-variable concurrency and called it COMPLEX. This still results in a system, which is sequential in its nature and contains concurrent sections. This approach is fitting for their goals, as one of their future aims is the translation of low level C code into COMPLEX for verification of for example concurrent operating system kernels. Their work is inspired by the Hoare-Parallel framework[11] which is a formalization built using the Owicki-Gries method[12] in Isabelle/HOL. The Owicki-Gries method extends the verification procedure of sequential programs by introducing the notion of *interference freedom*. It works by first proving each single thread correct on its own and then the atomic instructions of the threads are proven not to interfere with the correctness of each other.

2.3 Previous works in Coq

A case study[13] shows that there are many advantages to the library based approach, which eliminates some complicated modeling tasks by implementing a generalized version of parallelism instead of aiming straight for the proof of a concrete program. They investigate these properties through the verification procedure of a mail server originally written in Java by rewriting it in a modeling language provided by a library. [14] The development of the Coq library is discussed in [15]. It contains a modeling language based on π-calculus [16] [17] [18] (which has been modified to enable the usage of Coq datatypes and control structures), a specification language and a collection of reusable lemmas. In this sense, it is similar to our approach, as these mentioned components are also present among our results. Models formalized in their library are also suitable for extraction into executable OCaml code using the facilities provided by Coq, although it is not fully automatic and requires manual modifications.

Another paper based in Coq [19] is closely related to our work in the sense, that it takes the UNITY theoretical programming language as its subject of formalization, the same system on top of which the material of the course discussed later is based on. Their approach is much more abstract, as they are only constraining elements to functions between certain types, not limiting them to be created by an established set of constructors. Also, they do not discuss conditions at all. Compared to this, while being less general, our results are much more approachable, since they are built using inductive types that not only give constructing blocks for programs

and predicates, but also serve as stricter and stronger guiding outlines to aid the development of proofs about the system.

2.4 Our work

Our approach is different from these examples in that its main goal is to aid the development of solutions given through the formalized specification language, not necessarily the verification of already existing codebases. The control flow (or more like the lack thereof) is also a big differentiation factor of our model when comparing to other similar systems.

2.5 Proof assistant for parallel systems

In another article[20], that is more on the theoretical side, but we still considered relevant, the description of a proof assistant based on the algebra of communicating processes, specifically designed for proofs around parallel systems is given.

Chapter 3

Background

In the following chapter we introduce the material that we based our formalization on and the system we chose as our meta-theory. We do not cover the whole model in detail, but mostly focus on the concepts that are needed for understanding the results of our work. A short introduction to Agda, the language of our choice can also be found at the end of the chapter.

3.1 Subject: Specification and Implementation of Distributed Systems

The subject of our formalization is a part of the material from the *Specification and Implementation of Distributed System* course taught at ELTE. The main goal was following the official notes of the subject[21] as closely as possible, but since the code needed to be more precise than written definitions, some constructs were altered. The system extends a relational model defined in [22] which itself is built upon Hoare-logic [23][24], more specifically concurrent Hoare-logic[25]. Proofs for sequential programs are handled using pre- and postconditions with the devices provided by first order logic. This is expanded to parallel execution in a similar fashion to UNITY. Abstract programs are formalized through a set of conditional assignments the execution of which are represented as a list of state transitions that can be considered as trees, which results in a branching-time temporal logic [26] (more specifically computation tree logic) like system.

An abstract program as defined by this system can be thought of as an initialization and a set of further conditional instructions. These are then selected randomly and if their condition is satisfied in the current program state, they are executed. This can be done in any arbitrary order, their behavior stays the same if the scheduling is impartial. The execution units running such a program can be thought of as processing cores, or even several computers working together in a cluster.

Definition 3.1.1. Interleaving semantics In a model that conforms to interleaving semantics, for every valid parallel execution, there also exists a linear execution path of the same instructions, that achieves equivalent results.

Definition 3.1.2. Impartial Scheduling A scheduling is called impartial if it ensures that running the program indefinitely will result in the selection of each and every conditional instruction infinitely many times.

After laying down the foundation for constructing programs and reasoning about their behavior, the subject introduces a formal concept of task specification and then defines the correctness of a program by proving that it adheres to those constraints posed by the specification. We iterate on these constructions and implement them, so the verification of the adherence can be done on the computer, thus avoiding potential human errors that can occur while carrying out such a process on paper.

There are multiple ways, in which said correctness of a program can be shown. Simply giving a proof, that all the imposed criteria are held is always sufficient, but gets inconvenient for larger programs. To avoid the complication of proofs, one can utilize the method of reduction to an already known problem. If there is a recurring pattern, general statements about its behavior can be proven, so that these do not need to be done each and every time the pattern is used. Just showing how the currently examined part of the program corresponds to the pattern is enough instead. Another technique for avoiding repetition and introducing segmentation into proofs to keep their brevity is to construct programs from other smaller programs, by for example taking their union, essentially using them as so called *building blocks*.

3.1.1 Mathematical foundation[1]

The basics of the material are built on top of a relational model, the development of which originates from the works of Ákos Fóthi[27]. This model deals with programs, that operate over a so called **state space**, which is built as a Cartesian product of the individual sets of possible values for the variables of the program: $A = A_1 \times A_2 \times ... \times A_n$. These sets are individually called **type value sets**. The value of a single variable in a certain state can be accessed by the projection function $v_i : A \to A_i$.

Let A^* denote the finite and A^∞ the infinite sequences made of elements in A. The set of all sequences with elements from A, that can either be finite or infinite is defined as $A^{**} = A^* \cup A^\infty$.

We call a function a *logical function*, if it maps into the set of Boolean logical values defined as $\mathbb{B} = true, false$. A subset of these functions, that not only have $\mathbb{B}$ as their range, but also a state space A as their domain can be called conditions. These have the type $\pi : A \to \mathbb{B}$.

Definition 3.1.3. Truth set The subset of the state space in which these conditions are satisfied is often of interest, for this purpose the notion of the **truth set** of a function is defined as the subset of its domain that it maps to *true*. It is denoted as follows: $\lceil \pi \rceil = \{a \in A \mid \pi(a) = true\}$

Definition 3.1.4. Implication One condition is said to imply another ($P \Rightarrow Q$) if its truth set is subset of the other's ($\lceil P \rceil \subset \lceil Q \rceil$).

Another specific kind of function that operates on states is the so called **state transition function**. It has the type signature of $A \to A$ and is used to describe the behavior of an

[1]The original model is more general than the one described here, but since our formalization uses functions, that are deterministic, we constrained the ranges of some relations to single values instead of sets.

instruction. In this model, we consider only a single kind of instruction, namely a generic type of assignment, which is defined in the following way. Given a function for every variable index of the state space, that maps from the entire state space to the type value set of that given variable ($\forall i \in \{1..n\} : F_i : A \to A_i$), combining these into a single function $F = (F_1, ... F_n)$ we get a state transition function. Semantically speaking, it describes a parallel assignment, where each variable can have a formula, that may depend on all the other variables (or itself as well), these then are evaluated in the same state and get their new values assigned concurrently to produce a new, updated state.

Such an assignment can be made conditional by pairing it with a logical function. By doing so, we alter its state transition function in a way, that it only applies the changes to the state, if its condition is fulfilled (meaning, that the current state is in its truth set), otherwise it acts as an identity function. With mathematical notation, given $\pi : A \to \mathbb{B}$ as a condition and $F : A \to A$ as an assignment, we get the following conditional state transition function:

$$F'(a) = \begin{cases} F(a) & \text{if } a \in \lceil \pi \rceil \\ a & \text{otherwise} \end{cases} \tag{3.1}$$

A parallel program ($S = (s_0, \{s_1, ..., s_m\})$) defined using these constructs consists of a single initialization assignment and a set of conditional assignments to be repeatedly performed later in no particular order. Any requirements that would be posed for their sequentiality need to be encoded into their conditions, which in turn can ensure some sort of a control flow. A parallel program constructed in such a way can be semantically thought of as a relation that maps any state to a branching series of possible successive states (see Figure 3.1 as an illustration). If we remove the successively duplicated states from these series by replacing all consecutive repetitions of a single state with only one instance, we get their so-called *reduced* versions. This gives us the type signature $A \times A^{**}$, meaning that for any given state, the program determines the set of possible execution paths, which are either finite (if a program terminates by reaching a fixed point) or infinite (that might represent a continuously running server) series of states.

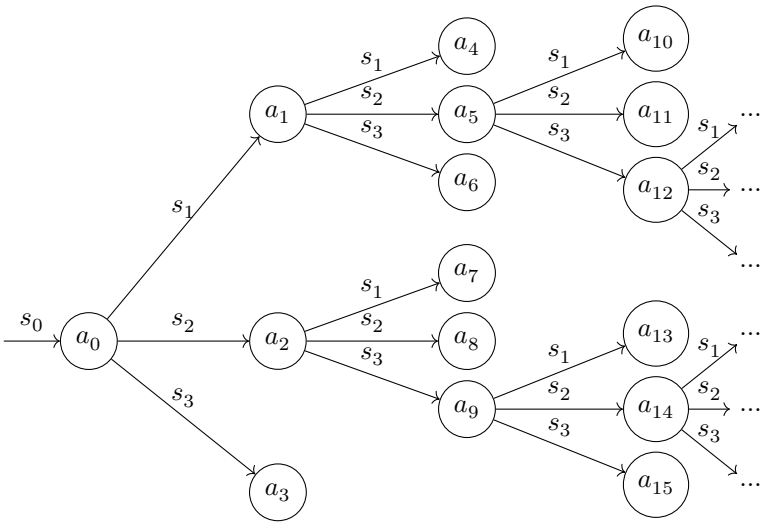

Figure 3.1: Branching execution with possible paths, e.g., (a_0, a_2, a_9, a_{15}), $(a_0, a_1, a_5, a_{12}, ...)$.

By constructing a type of specification that only places constraints on states, we can achieve independence from the actual structure of programs operating on these states. Problems consist of seven so called *Specification Properties*, which are themselves given as relations over the power set of the state space. These provide constraints for the behavior of the program, that will eventually have to satisfy all of them to be considered adherent to the specification and thus solve the problem.

Let $P, Q : A \to \mathbb{B}$, the infix notation $P \bigcirc Q$ mean $(\lceil P \rceil, \lceil Q \rceil) \in \bigcirc$, where $\bigcirc$ is any of the binary relations $\rhd, \mapsto, \hookrightarrow$ and for the unary relations $\bullet \in \{FP, INIT, inv, TERM\}$, let $P \in \bullet$ mean $\lceil P \rceil \in \bullet$

Let the following specification properties be proposed:

- $\rhd \in \mathcal{P}(\mathcal{P}(A) \times \mathcal{P}(A))$, where $P \rhd Q$ means, that leaving $\lceil P \rceil$ requires going through $\lceil Q \rceil$
- $\mapsto \in \mathcal{P}(\mathcal{P}(A) \times \mathcal{P}(A))$, where $P \mapsto Q$ extends upon $P \rhd Q$ by stating that $\lceil P \rceil$ can indeed be left through $\lceil Q \rceil$, and only through it
- $\hookrightarrow \in \mathcal{P}(\mathcal{P}(A) \times \mathcal{P}(A))$, where $P \hookrightarrow Q$ states that from every state in the truth set of P we must eventually get to a point in the truth set of Q (formally it is the transitive disjunctive closure (see Definition 3.1.5) of $\mapsto$)
- $FP \subseteq \mathcal{P}(A)$ where $P \in FP$ means that if the program has reached a fixed point (meaning, that it can make no further state transitions), then P must be satisfied
- $INIT \subseteq \mathcal{P}(A)$, where $P \in INIT$ means that the predicate P is satisfied at the beginning of the program
- $inv \subseteq \mathcal{P}(A)$, where $P \in inv$ means that P is satisfied at the beginning of the program and once its truth set has been entered, it cannot be left, thus it is an invariant for the entire program
- $TERM \subseteq \mathcal{P}(A)$, where $P \in TERM$ means that after the program has entered the truth set of P, it is guaranteed to eventually reach a fixed point

Definition 3.1.5. Transitive Disjunctive Closure The transitive disjunctive closure of a relation $R \subset (A \times A)$, where $\vee : A \to A \to A$ is given as the binary disjunction operator, is denoted by $R^{tdl} \subset (A \times A)$. It is defined as the minimal relation with the following properties.

- It is a superset of the original relation ($R \subset R^{tdl}$).
- Transitivity: For every $(a, b, c \in A)$ if $(a, b) \in R^{tdl}$ and $(b, c) \in R^{tdl}$, then $(a, c) \in R^{tdl}$ as well.
- Disjunctivity: For every $(a, b, c \in A)$ if $(a, c) \in R^{tdl}$ and $(b, c) \in R^{tdl}$, then $(a \vee b) \in R^{tdl}$ as well.

A tuple with seven items, each describing elements from one of the relations given above, can be considered as a specification for a program. If a parallel program satisfies all the constraints posed by a specification, it is considered to be a solution for the given problem.

3.2 Type Theory

In the following section we will discuss what Type Theory is, how it enables formalization of models with proofs, to be checked by computers and explain our choice of the Agda implementation.

3.2.1 General Principles Overview

Type theory is an alternative foundation for mathematics, which enables the formalization of constructive proofs through the connections to intuitionistic logic given by the Brouwer–Heyting–Kolmogorov interpretation. After formalizing a model by defining its types and their elements, one can express statements and theorems in forms of new types, the instances of which can be thought of as proofs for them. This is due to the so called *"propositions-as-types"* paradigm, formally known as the Curry–Howard isomorphism.

As an example, given two statements, A and B, the function with the type signature $A \to B$ represents the theorem which claims that every proof of A can be mapped to a proof for B, thus A ensures B. If an implementation for a function with said type is given, the theorem can be considered the be proven, since if we are in possession of a proof for A, we can execute it and obtain a proof for B. The fact that it does not only claim, that such an instance exists, but gives one right away instead is called a constructive proof.

3.2.2 Implementations

There are several existing programming languages that implement type theories, thus are capable of describing theorems and constructing proofs for them. They achieve this using dependent typesystems, in which types can not only be parametrized over other types, but also values. Out of these we tried the three most popular alternatives.

Idris

Idris[28] is a relatively new contender in the field of programming with dependent types. Its development is led by Edwin Brady with the aim of creating a general purpose language.

We first tried implementing our formalization in Idris, as it seemed to be the freshest language. It also has good support for every major operating system, wider editor compatibility than the alternatives and in some sense a more modern standard library, which is easier to discover due to the in our opinion better documentation.

Under certain circumstances the implicit parameter handling of Idris seemed to be better for us than that of Agda, and there are constructs, such as for example heterogeneous vectors that are easier to implement in Idris, because it uses cumulative universes, but after reaching a certain complexity, the type checker of Idris unfortunately turned out to be too slow, so we moved on to

other languages. It also lacks Unicode support, with perfectly valid and understandable reasons outlined by Edvin Brady, but it also makes the code a lot more verbose and harder to integrate with the original notation.

(Idris also has some support for writing proofs using tactics, but it has been deprecated in favor of a reflection based approach.)

Coq

The Coq proof assistant[29][30] is the earliest one among the three, even predating Agda by ten years. Its development began at the French Institute for Research in Computer Science and Automation, the initial version was released on the 1st of May, 1989. It is really well established and serves as a foundation for many formalization projects, some of which were already discussed in Chapter 2 (Related Works).

It uses the syntax of OCaml for its definition language *(Gallina)* and has a separate language for writing proofs using tactics instead of regular functional programming constructs. This separation might seem valuable in certain situations, but having to deal with two different types of code side by side did not seem ideal for this project. While support for Unicode characters is present, the conventions usually do not really utilize them.

Agda

Agda[31] is a dependently typed programming language that was originally described in the PhD thesis of Ulf Norell[32] and later completely rewritten for a second version. It is really similar to Idris (in fact they are both really similar to Haskell) syntactically, but is a lot more mature language with more tried and trusted principles, a broader range of available sources and a much larger user base.

The fact that it uses the same language for describing the components and behavior of the discussed model as for making statements about it and constructing proofs for them results in code, that is cleaner in our opinion.

Its conventions of highly relying on Unicode symbols for the naming of identifiers, such as functions and data types helped in staying similar to the original notation of the curriculum of the subject.

Another factor that was taken into consideration while making the choice was the fact that Agda has already been taught at our university and there are multiple people with high levels of knowledge of it, so in this environment it seemed optimal for collaborating with others on the project.

A brief introduction to Agda is included here, for a more thorough guide see [33] or [34].

Let us begin by introducing the inductive type of the natural numbers. Instead of the usual set-theory way of describing the constraints that define a certain set, in Agda we use a constructive

approach, by listing all the ways in which elements of a certain type can be created. These are called constructors, which often represent the underlying axioms of a model. The natural numbers have two of these rules, one states that zero is a number while the other claims that every natural number has a successor, which is a natural number as well.

```
data ℕ : Set where
  zero : ℕ
  suc : (n : ℕ) → ℕ
```

Now, that we have defined a datatype, the implementation of addition is shown as an example for a function. In a very similar way to what one might see for example in Haskell, inductive types can be processed using pattern matching.

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

For easier handling and the illustration of different ways to do so, we also define some constant numbers to be used later. The alternative definitions given in the comments would have the same results.

```
one two three four five six : ℕ
one = suc zero
two = suc (suc zero) -- or alternatively: (suc one)
three = suc two -- or alternatively: suc (suc (suc zero))
four = suc three
five = two + three -- or alternatively: (suc four)
six = suc (two + three)
```

These previous definitions together can be used to describe a property of natural numbers which states that a certain number is even. This property is given as a special function, that maps each natural number to a type representing the statement, for which giving an instance to that type serves as a proof. The first constructor simply states that zero is an even number, while the second one formalizes the fact that any number constructed by adding two to an already known even number becomes also even itself. This is the first definition where dependent typing comes into play, as the resulting type of the second constructor is not constant, but depends on the type of the parameters given to it. (The comment shows an equivalent alternative to the second constructor.)

```
data Even : ℕ → Set where
  even-0 : Even zero
  even-2+ : {n : ℕ} → Even n → Even (two + n)
  -- even-2+ : {n : ℕ} → Even n → Even (suc (suc n))
```

As a few examples on how to utilize these constructors, here we present the proofs for some known constant even numbers. Notice, that in the proof for six, we utilize the proof of two. This ability to reuse smaller parts greatly helps in the construction of bigger proofs.

```
even-zero : Even zero
even-zero = even-0

even-two : Even two
even-two = even-2+ even-0

even-six : Even six
even-six = even-2+ (even-2+ (even-two))
```

We also show the implementations with some examples for a few of the usual logical expressions, such as the disjunction. It has two constructors, since it can be proven by showing that either the left or the right side statement holds.

```
data _⊎_ (A B : Set) : Set where
  inj₁ : (x : A) → A ⊎ B
  inj₂ : (y : B) → A ⊎ B
```

To define negation, we first need to introduce the empty type, which essentially represents a false statement, by having no constructors, thus no means, by which an instance could be constructed for it. Afterwards we can represent negation as a function that maps to the empty type from a certain other type, since by constructing an instance for such a function we can show that the type given as parameter must be empty as well. This is due to the fact that if there existed an instance for that type, we could create an element for the empty type by applying this function to it, which we know is impossible, since by definition no instance exists for the empty type.

```
data ⊥ : Set where
-- No constructors given, this type is empty.

¬_ : Set → Set
¬ S = S → ⊥
```

As an example for the usage of negation we present a proof for the fact that five is not even. By unfolding the previous definition, we get that this can be done by implementing a function that maps to the empty type. Since an instance for that type cannot be returned, we instead need to show that the parameter we have (`even-five`) can not exist in the first place. To do this, we can rely on the pattern matching powers of Agda. By either interactively or manually pattern matching on `even-five`, we can see what constructors it could have possibly been created with. Since the definition of `five = suc (suc (suc (suc (suc zero))))` does not match the pattern `zero`, that would be required for it to be created by `even-0`, we can omit the inclusion of that case, we only have to go forward with the `even-2+` constructor, in which case it needed to be created from a proof claiming three is even. In a similar way we can take one more step, to arrive at a point where we supposedly have an instance (`even-one`) for the type `Even one`. At that point, none of the constructors for `Even` can be matched, since `one = suc zero` does not match any of the possible patterns (`zero` or `suc (suc _)`), thus it can be replaced by an absurd pattern, for which we do not need to provide a right hand side.

```
not-even-five : ¬ Even five
-- not-even-five even-five = ?
-- not-even-five (even-2+ even-three) = ?
-- not-even-five (even-2+ (even-2+ even-one)) = ?
not-even-five (even-2+ (even-2+ ()))
```

Lastly, after the previous illustrations dealing with constant numbers, we give an example for a more general proof. The statement claims that the sum of two even numbers is always even. This can be shown by pattern matching on the constructor of the first proof and recursively *"calling"* the theorem again, when needed. This is possible, where Agda can determine that the argument of the function is structurally decreasing, thus the recursion will eventually reach its base case and terminate.

```
even-plus-even : {n m : ℕ} → Even n → Even m → Even (n + m)
even-plus-even even-0 em = em
even-plus-even (even-2+ en) em = even-2+ (even-plus-even en em)
```

The remaining logical expressions, such as conjunction or existence are not presented here, but they can be implemented using simple pairs or dependent pairs respectively. They are provided for our formalization by the Agda Standard Library[35].

Chapter 4

Method

The development process was done in an iterative style, meaning that we had to backtrack multiple times and rewrite certain parts over and over, but this often helped in finding better solutions. We believe that experimenting with multiple variations instead of accepting the first one that works is vital for converging to a better result instead of achieving a certain goal earlier only to move on. This process can lead to code that is more expressive, concise, easier to understand and thus maintain and explain, reason with and about.

This approach also results in the later layers built on top of earlier parts having a better foundation. The complexity of the definitions of data types and the functions describing their behaviors directly impacts the complexity of the proofs written for statements about them. Finding a simpler, but equivalent way to formalize a definition, that still results in equivalent semantics can lead to greatly simpler proofs.

After formalizing certain constructs and writing some proofs in connection with them it is usually much easier to see what causes complications and how to modify them to make their handling easier. These discoveries have sometimes led to decisions that diverged from the original material, warranted by the differences in required precision between paper based proofs, where certain parts can be omitted and computer based proofs, that need every little detail to be verified.

Sometimes the formalization of certain parts led to deeper understanding of their behavior, because of which we were able to restructure the definitions and proofs in a way that helped incorporate more and more elements from the Agda Standard Library[35], thus reducing the amount of added code and representing parts of our formalization as specialized versions of more generic concepts.

The work was done in smaller segments, which made it feasible to build and extend the model step-by-step instead of having to write one huge monolithic codebase all at once without being able to test smaller parts of it. This approach sometimes had drawbacks, when adding a new element required deep modifications in the existing parts, but overall it helped in advancing the project at a steady pace.

We started with an implementation of a highly simplified model in Idris, that served as an initial proof-of-concept. The main outlines were laid down and valuable experience was gained from discovering different possible workflows for the transformation of paper based mathematical definitions into code. Some simple proofs were written using that, but after trying to expand to a more complex model, the type checking of Idris seemed to be too slow, which led to the decision of switching to Agda.

Luckily the similarities of Agda and Idris meant, that our previous work was not lost, because it was easily translatable into the new environment. Parallel to the development of the base model we started including some statements in our code as well, which we prove for the current state of the codebase, and seeing if they were still true after modifications to the underlying model helped us determine if the introduction of any given change would break the consistency of the model or not.

After proofs started getting longer and more complicated, we started looking into ways to simplify and shorten them. For example we were able to achieve cleaner types during the interactive hole based proof process after eliminating dependent branching from them by rewriting some semantic functions to use auxiliary functions instead of using the built-in with-abstraction provided by Agda.

We managed to further simplify proofs by moving the representation of arrays from lists to functions, which map from the set of natural numbers to the type of the contents of the array. By introducing this Agda could rely on the fact that functions always evaluate and simplify the proofs by computing the required types automatically, instead of us having to do pattern matches on the constructors of lists, thus several proofs became much shorter.

The iterative method also applied to our aim of increasing the precision of the formalization in terms of similarity to the original material where it was possible. Sometimes it was easier to experiment with simplified versions first, which then could be later adapted in their details to be more consistent with the subject. An example for such is the moving of variable indexes to finite sets instead of the initially introduced state space that was mapping from natural numbers. This was done as a step towards conformance with the statement defining state space as a Cartesian product of finitely many type value sets. We generally tried to stay as close as possible, but we encountered some parts, which required modifications, because they were designed to be convenient for human reasoning, but are not formal enough to be sufficient for type checking.

Another philosophy, that we tried to follow was the reduction of duplication by extracting and reusing code that appeared in multiple places. One example for this was the handling of Predicates, Conditions and Assertions. Originally, we had the **Predicate** inductive datatype, which described certain constraints that could either be checked if they evaluate to true in a certain state by converting them to a **Condition**, or a proof could be given that claims that they are satisfied by converting them to an **Assertion**. (More details about this will be covered in Chapter 5.) These conversions had common parts which we later extracted into an intermediary representation by introducing the **Decision** type, which unified the shared part of the semantics and could later be turned into either a **Condition** or an **Assertion** more easily.

A similar change was introduced, when we split the definition of **Ensures** ($\mapsto$) by introducing **Progress** ($\rightarrowtriangle$), which allowed us to construct partial proofs that applied to **Progress** on its own and later utilize them as parts of the proofs for **Ensures**. This resulted in a better structured and more readable code.

When introducing a new construct, it was often implemented in a very specific way at first, which aided the initial implementation, but after gaining a better understanding of the behavior and meaning of that certain part, by for example writing some proofs for it, they were often rewritten in a more generic fashion in order to make the model as extensible as possible, for example only introducing arrays of natural numbers first and later extending the array type to be parametrized with the specific set of types defined.

As another example for this, adding general equality of expressions as a predicate instead of only comparing natural numbers required the introduction of the notion of our own equality, since decidable equality for functions cannot be implemented, and that was needed due to the conversion of predicates to conditions. (This would have been necessary anyway when introducing data channels, since their equality is defined as a quotient with regards to their actual contents, their history does not matter during comparisons.) This generic decidable equality made the storage of array lengths necessary as well, since the comparison function needed to know which part of the function's domain it has to evaluate. When we replaced the built in equality with our own alternative, we used a function at first, that mapped to a different type for each datatype constructor (for example it simply mapped back to the built-in equality for natural numbers), but that turned many implicit variables that were previously automatically solved in our proofs unsolvable by Agda, thus we changed the representation to an inductive datatype with a single constructor instead of the function and successfully aided Agda in solving the metavariables this way.

If we had tried to do all of these changes at once, it would have been easy to get lost dealing with multiple design decisions at the same time and not really seeing a clear way forward, but dealing with them step by step allowed us to keep the tasks in a visible scope and not lose track of our goals.

After formalizing statements that describe some safety and progress properties of programs we noticed that there was a certain implicit variable that Agda had trouble figuring out on its own in multiple scenarios. This was the s_0 initialization instruction, the first part of the representation of a parallel program, which the original material defines as a pair made out of a single initialization instruction and a set of conditional instructions that can be executed afterwards in any order. By observing the definitions we realized that it is indeed essentially irrelevant at the later stages of the program and only plays a role in constructions dealing with the beginning, such as **INIT** and **inv**. By splitting up the definition into the separate notions of **ParallelProgram** and **InitializedParallelProgram**, we were able to avoid the extraneous inclusion in the properties for which it was unnecessary and eliminate the ambiguous implicit variables.

As the credibility of the results that are proven in such a system depend greatly on the soundness of the underlying implementation, we have also added some safeguards in forms of assertions that describe the desired behavior of the implemented semantic functions. While this does not fully cover all the possible mistakes, by constructing instances for these types, we confirmed that some important properties hold for the foundation upon which our further work is based.

Finally, we have prepared the system for the inclusion of data channels, by implementing a type called **QueueWithHistory** and the functions that are necessary for handling the manipulation of its representation. The particularity of channels in the sense, that they act more like objects with methods makes the description of their behavior with just simple assignments complicated, which is undesirable, since it would greatly increase the complexity of not only their definitions which in turn also makes it more prone to mistakes, but also undermine the clarity of proofs based around them. Because of this, we had to change the representation of instructions from a condition paired with a list of variables with value expressions to be assigned. Instead we now have a condition paired with a list of instructions, that will include the extension (enqueue) and removal (dequeue) operations for channels. The problem is that channels would need to be excluded from assignments, so their values cannot be arbitrarily overwritten. Due to time constraints we were not able to devise a clean method that achieves the correct behavior, thus proofs about channels are outside of the scope of this current paper.

Chapter 5

Results

In this chapter we discuss the different kind of outcomes we have currently arrived at, explain how they are built and illustrate them with samples from the source code.

5.1 Formalization

In this section we present our formalization of the notion of parallel programs, predicates and their semantics introduced by the model we build upon. The formalization of a parallel bubble sort and the concepts that are required for writing specification statements are also included.

5.1.1 Parallel Programs and Predicates

Since our formalization does not follow the book of the subject exactly, we present the following basic notions that we introduced in our version and built upon:

- `Type` - We defined the set of possible types of variables in the programs as an inductive datatype, which enabled the use of parametric types as well.

```
data Types where
  Nat : Types
  Array : Types → Types
  DataChannel : Types → Types
```

- `Environment` - An environment has a defined number of variables and a type assigned to each of them. It essentially describes a state space. (The type `Fin n` denotes the finite set with n elements, see Appendix A for its definition.)

```
module Environment (varCount : ℕ) (varTypes : Fin varCount → Types) where
```

- `Variable` - A variable in this model can be thought of as just an index, that can be used to reference a certain part of the state.

```
Vars : Set
Vars = Fin varCount
```

- `State` - We modeled a state as a function that maps the previously introduced variables to the respective value sets of their types. (The `evaluateType` function assigns the previously defined data types to their Agda representations. It has multiple possible definitions which can all be equivalent in terms of the expressive power of the resulting model, what it affects is the structure of the proofs. A possible definition we used can be found in appendix A.)

```
State : Set
State = (i : Vars) → evaluateType (varTypes i)
```

- `Expression` - The building blocks that can be used for constructing an expression that can either be on the right hand side of an assignment or serve as a part of a predicate. They are annotated with the types that are the results of their evaluation, which increases the potential for code reuse by eliminating the need for separate constructors for separate types and prevents their usage in the wrong contexts.

```
data Expression : Types → Set where
```

  - The `Const` constructor creates a constant expression by directly giving the Agda representation of its value.

```
Const : {A : Types} → evaluateType A → Expression A
```

  - The `GetArray` constructor creates an expression that accesses a value at a specific index from an array.

```
GetArray : {A : Types} →
  Expression Nat → Expression (Array A) → Expression A
```

  - The `SetArray` constructor creates an expression that creates a modified version of a given array, that contains an updated value at the given index.

```
SetArray : {A : Types} →
  Expression Nat → Expression A → Expression (Array A) →
  Expression (Array A)
```

  - The `Var` constructor creates an expression, the result of which is the value of a given variable in the state it is evaluated in.

```
Var : (x : Vars) → Expression (varTypes x)
```

  - The expressions created by the `Plus` constructor evaluate to the sum of what its subexpressions evaluate to.

```
Plus : Expression Nat → Expression Nat → Expression Nat
```

– The `Hiext` constructor allows for the creation of expressions that evaluate to an extended version of a given channel that has a new value added to it. This is more commonly called the *enqueue* operation.

```
Hiext : {A : Types} →
  Expression A → Expression (DataChannel A) → Expression (DataChannel A)
```

– The `Lov` constructor creates an expression that accesses the first value in a given channel. This is more commonly called the *peek* operation.

```
Lov : {A : Types} → Expression (DataChannel A) → Expression A
```

– The `Lorem` constructor creates an expression that evaluates to an updated version of a given channel, that has its first element removed. This is more commonly called the *dequeue* operation.

```
Lorem : {A : Types} →
  Expression (DataChannel A) → Expression (DataChannel A)
```

– The `History` constructor creates an expression that evaluates to a channel that has all the elements in it that were ever added to the channel given as parameter.

```
History : {A : Types} →
  Expression (DataChannel A) → Expression (DataChannel A)
```

– Expressions created with the `Len` constructor evaluate to the length of a given channel.

```
Len : {A : Types} → Expression (DataChannel A) → Expression (Nat)
```

+ Also, the following syntactical notations were defined for easier usage when constructing programs.

```
v[_] : (x : Vars) → Expression (varTypes x)
v[ x ] = Var x

infix 3 _g[_]
_g[_] : {A : Types} →
  Expression (Array A) → Expression Nat → Expression A
el g[ ei ] = GetArray ei el

infixl 3 _s[_]=_
_s[_]=_ : {A : Types} →
  Expression (Array A) → Expression Nat → Expression A →
  Expression (Array A)
el s[ ei ]= ev = SetArray ei ev el
```

◦ `Predicate` - The constructors of this inductively defined type describe the possible circumstances, to which we can later constrain the execution of conditional instructions. Their short descriptions are given in comments. By defining this in this manner, instead of simply relying on general logical functions that map from states to Boolean values, we can achieve

much cleaner proofs, by actually getting the information we have in a way that can be used for reasoning instead of having to extract it from the parameters every time individually.

This inductive definition was also crucial to enable the decidability of assertions, which were created from predicates (see `assertionDecidability` in Appendix A), by enabling pattern matching on the predicates instead of representing them as generic logical functions, which in turn made the formalization of De Morgan's law of union possible.

```
data Predicate : Set where
  TRUE : Predicate -- Predicate that is true in every state
  FALSE : Predicate -- Predicate that is false in every state
  NOT : Predicate → Predicate -- Logical negation
  AND : Predicate → Predicate → Predicate -- Logical conjunction
  OR : Predicate → Predicate → Predicate -- Logical disjunction

  EQ : {A : Types} → Expression A → Expression A → Predicate -- Equality
  LTE : Expression Nat → Expression Nat → Predicate -- Less Than or Equal
  GTE : Expression Nat → Expression Nat → Predicate -- Greater Than or Equal
  LT : Expression Nat → Expression Nat → Predicate -- Less Than
  GT : Expression Nat → Expression Nat → Predicate -- Greater Than
```

Similarly to the expressions, predicates have their own syntactical operators that make writing and reading them easier in more complex formulae.

```
¬_ : Predicate → Predicate
¬_ = NOT

infixr 6 _△_
_△_ : Predicate → Predicate → Predicate
_△_ = AND

infixr 5 _▽_
_▽_ : Predicate → Predicate → Predicate
_▽_ = OR
```

- `Instruction` - Currently there is only one type of instruction, a basic assignment that is made up from a variable and an expression of the same type to be evaluated and assigned to it. This will later need to be extended to add support for the proper behavior of certain more complicated data structures, like channels.

```
data Instruction : Set where
  -- SKIP : Instruction
  Assignment : (var : Vars) → Expression (varTypes var) → Instruction
```

(Initially a special instruction, `SKIP` was included to be used when no operation should be done, but in the current system that can be achieved by using an empty list of instructions.)

The following list contains the semantic operators that give the previously defined Predicates meanings. Their explanations are followed by code examples that give more precise typing information on them.

- ⟦_⟧a - Makes an assertion from a predicate by turning it into a function from state to type. By creating a type, an instance can later be given for it, proving that the assertion and thus the predicate holds in the given state.

```
Assertion : Set₁
Assertion = State → Set
⟦_⟧a : Predicate → Assertion
```

- ⟦_⟧d - Turns predicates into decisions, a special type that either carries a proof that a certain predicate holds in a given state, or a refutation claiming the opposite.

```
Decision : Predicate → Set
Decision p = (st : State) → Dec (⟦ p ⟧a st)
⟦_⟧d : (p : Predicate) → Decision p
```

- ⟦_⟧c - Turns predicates into conditions that can be evaluated for a given state. Essentially turns a predicate into a function from states to Boolean logical values.

```
Condition : Set
Condition = State → Bool
⟦_⟧c : Predicate → Condition
```

Similarly, there are also operators for handling instructions.

- ⟦_⟧pi - This operator describes the semantics of a parallel instruction, meaning an instruction that is meant to be executed concurrently with other instructions. It has two parameters, the state, based on which the expressions occurring in the instruction should be evaluated, and the state function that should be updated to reflect the effects of the instruction. This separation is necessary to properly represent the semantics of concurrency, as the instructions are supposed to be treated as if they were executed at the same time, thus the results of their expressions need to be based on a common original state.

```
⟦_⟧pi : Instruction → State → State → State
```

- ⟦_⟧pil - By utilizing the function defined above for individual parallel instructions, this semantic operator executes a list of instructions in a parallel-like fashion by starting them from the same state and accumulating their modifications.

```
⟦_⟧pil : List Instruction → State → State → State
```

- ⟦_⟧i and ⟦_⟧il - The usage of the previous operators is made easier by defining ⟦_⟧i and ⟦_⟧il, each having one less parameter. They call their respective more general versions by taking the argument they received and passing if forward twice. (The ⟦_⟧il operator essentially turns a list of instructions into a single deterministic state transition function, that is analogue to a simplification of what the original model describes.)

```
⟦_⟧i : Instruction → State → State
⟦_⟧il : List Instruction → State → State
```

While these would already be sufficient, a few more type aliases were also created for the purpose of increasing clarity and brevity, thus making the code easier to read and understand:

- `Batch` - A list of instructions that should all be executed at once. Their expressions are all evaluated in the same state, thus their results turn out as if they were done simultaneously.

```
Batch : Set
Batch = List Instruction
⟦_⟧b : Batch → State → State
⟦_⟧b = ⟦_⟧il
```

- `ConditionalBatch` - A `Batch` that is paired by a `Predicate` that represents the preceding conditions under which it should be executed.

```
ConditionalBatch : Set
ConditionalBatch = (Predicate × Batch)
```

- `ParallelProgram` - A list of `ConditionalBatch`-es, that represents the set of operations which are executed in an impartially scheduled (see Definition 3.1.2) way during runtime.

```
ParallelProgram : Set
ParallelProgram = List ConditionalBatch
```

- `InitializedProgram` - A `ParallelProgram` extended with a special batch of instructions to be performed at the beginning of the execution.

```
InitializedProgram : Set
InitializedProgram = (Batch × ParallelProgram)
```

With all of these in place we can finally introduce the operator that determines the meaning of the main building block of our model, and by doing so essentially connects the previous definitions to the specification statements discussed later.

- $[\![\_]\!]cb$ - The conditional batch effect function checks if the given condition is fulfilled, in which case it produces a state to be returned, that is updated by the effects of the instructions in the batch, otherwise it returns the state untouched. Its behavior can be mathematically described using the following definition:

$$[\![(P, b)]\!]cb(st) = \begin{cases} [\![b]\!]b(st) & \text{if } [\![P]\!]c(st) = true \\ st & \text{otherwise} \end{cases} \tag{5.1}$$

  This translates to the following Agda representation using dependent pattern matching:

```
⟦_⟧cb : ConditionalBatch → State → State
⟦ (P , b) ⟧cb st with ⟦ P ⟧c st
... | true = ⟦ b ⟧b st
... | false = st
```

Even though the definition given above is equal to the one below in terms of their meaning, separating the decision by introducing a helper function resulted in cleaner proofs, so in the current codebase the following version is in use:

```
⟦⟧cbHelper : Bool → Batch → State → State
⟦⟧cbHelper true b st = ⟦ b ⟧b st
⟦⟧cbHelper false b st = st

⟦_⟧cb : ConditionalBatch → State → State
⟦ (P , b) ⟧cb st = ⟦⟧cbHelper (⟦ P ⟧c st) b st
```

5.1.2 Example: Parallel Bubble Sort

As an example for how parallel programs can be represented in the system described above, we formalized the following concurrent bubble sort algorithm as it is defined in [21] and [36].

$$\begin{aligned} BubbleSort_n = (& \\ & SKIP, \\ & \{\, (\,(a[i] > a[i+1])\,,\ (a[i], a[i+1] := a[i+1], a[i])\,) \mid i \in \{1, \ldots, n-1\}\,\} \\ &) \end{aligned} \tag{5.2}$$

First we created an environment that has a single variable of the type that represents an array of natural numbers.

```
varTypes : Fin 1 → Base.Types
varTypes 0F = Base.Array Base.Nat

open module BubbleSortEnvironment = Base.Environment 1 varTypes
```

The resulting code has two functions for creating the predicates and the instructions and a main definition that acts as the set-builder notation by constructing the list using the functions. Notice that the program receives a natural number as parameter, it is used to determine the length of the array to be sorted.

```
makePredicate : ℕ → Predicate
makePredicate n = GT (v[ 0F ] g[ Const n ]) (v[ 0F ] g[ Const (suc n) ])

makeInstruction : ℕ → Instruction
makeInstruction n =
  Assignment 0F
    (
      v[ 0F ]
      s[ Const n ]=(v[ 0F ] g[ Const (suc n) ])
      s[ Const (suc n) ]=(v[ 0F ] g[ Const n ])
    )
```

```
bubbleSort : ℕ → ParallelProgram
bubbleSort count =
  Data.List.map
    (λ x → (makePredicate x , [ makeInstruction x ]))
    (downFrom count)
```

5.1.3 Specification Statements

Now, that we have the foundations laid down, we can start defining the statements that will be available for building specifications. To make the code cleaner the following operators are introduced for denoting the interaction between states and the various forms of predicates.

- Mapping a state and a predicate or assertion to a set creates a type that claims that the predicate is satisfied in the given state. An instance of the resulting type can serve as a proof for such a claim.

```
_⊢_ : State → Predicate → Set
st ⊢ p = ⟦ p ⟧a st

_⊩⊢_ : State → Assertion → Set
st ⊩⊢ a = a st
```

- Since all the predicates and conditions are decidable, we can evaluate them to Boolean truth values in any given state.

```
_⊨_ : State → Predicate → Bool
st ⊨ p = ⟦ p ⟧c st

_⊩⊨_ : State → Condition → Bool
st ⊩⊨ c = c st
```

We define a few so called *variables*, which in Agda essentially means identifiers with defined types that can be used as implicit parameters without having to mention them at every function that uses them.

```
variable
  P P₁ Q Q₁ R V : Predicate
  A B : Assertion
  C D : Condition
  W X Y Z : Set
  st : State
```

```
i : Instruction
s₀ b : Batch
cb : ConditionalBatch
cbs : List ConditionalBatch
S : ParallelProgram
I : InitializedProgram
```

Also, a new type synonym is introduced to make the code more expressive.

```
Statement : Set₁
Statement = Set
```

To express relationships between predicates we create an operator that describes implication (see Definition 3.1.4), meaning that for $P \Rightarrow Q$ to hold, every state, that satisfies P must satisfy Q as well. Extra operators are also defined for the inverse of the relation and a symmetric version.

```
infix 4 _⇒_
_⇒_ : Predicate → Predicate → Statement
P ⇒ Q = ∀{st : State} → st ⊢ P → st ⊢ Q

infix 4 _⇐_
_⇐_ : Predicate → Predicate → Statement
P ⇐ Q = Q ⇒ P

infix 4 _⇐⇒_
_⇐⇒_ : Predicate → Predicate → Statement
P ⇐⇒ Q = (P ⇒ Q) × (P ⇐ Q)
```

We can define the same relation for assertions in a similar way. Notice, that these definitions are essentially equivalent, the proof of their equivalence can be constructed trivially using the identity function, since they normalize to the same expressions, but some conversions between predicates and assertions can be avoided by introducing them separately.

```
infix 4 _⇛_
_⇛_ : Assertion → Assertion → Statement
A ⇛ B = ∀{st : State} → st ⊩ A → st ⊩ B

infix 4 _⇚_
_⇚_ : Assertion → Assertion → Statement
A ⇚ B = B ⇛ A

infix 4 _⇚⇛_
_⇚⇛_ : Assertion → Assertion → Statement
A ⇚⇛ B = (A ⇛ B) × (A ⇚ B)
```

Now, we can define the two core assertions, the *Weakest Precondition* and the *Strongest Postcondition*, that serve as the basis of the others. They describe the connection between predicates and instructions. We define them in an iterative manner, by first expressing their meaning using mathematical notation for simple instructions, then giving the formalized code segments for single instructions, batches of instructions, conditional batches and finally lists of conditional batches, which are essentially equivalent to parallel programs. Mathematical notation is also given for the last definitions. Since these cannot appear as conditions for instructions, nor can they always be decidable (since the decidability of a strongest postcondition could require the enumeration of the entire state space, which can potentially be infinite), they are not given as **Predicates** or **Conditions**, only as **Assertions**.

- Weakest Precondition - The weakest precondition for a given predicate with respect to an instruction represents the function that only evaluates to a non-empty type for a certain state, if after executing that instruction the given predicate holds in the resulting state. When applied to an entire parallel program, it represents the broadest possible subset of the state space, from which executing a single batch of the program definitely takes us into the truth set of the given predicate, no matter which one we chose.

$$WP(i, P) = [\![P]\!]a \circ [\![i]\!]i \tag{5.3}$$

```
IWP : (Instruction × Predicate) → Assertion
IWP (i , P) = λ st → ([[ i ]]i st) ⊢ P

BWP : (Batch × Predicate) → Assertion
BWP (b , P) = λ st → ([[ b ]]b st) ⊢ P

CBWP : (ConditionalBatch × Predicate) → Assertion
CBWP (cb , P) = λ st → ([[ cb ]]cb st) ⊢ P

CBLWP : (ParallelProgram × Predicate) → Assertion
CBLWP (S , P) = λ st → All (λ cb → st ⊩ (CBWP (cb , P))) S
```

$$CBLWP(S, P)(st) = \bigwedge_{s_i \in S} CBWP(s_i, P)(st) \tag{5.4}$$

- Strongest Postcondition - For a given condition and instruction the strongest postcondition gives the function that returns a type for a given state, which represents the existence of a starting state, in which the given condition is satisfied, and the execution of the given instruction results in the given state. In its last variation it represents the smallest possible subset of the state space, to which executing any single batch of the program can take it from the given starting state.

$$SP(i, Q)(st) = \exists st_0 \in State : [\![Q]\!]a(st_0) \wedge [\![i]\!]i(st_0) = st \tag{5.5}$$

```
ISP : (Instruction × Predicate) → Assertion
ISP (i , P) = λ st → Σ State (λ st₀ → st₀ ⊢ P → [[ i ]]i st₀ ≡ st)

BSP : (Batch × Predicate) → Assertion
BSP (b , P) = λ st → Σ State (λ st₀ → st₀ ⊢ P → [[ b ]]b st₀ ≡ st)

CBSP : (ConditionalBatch × Predicate) → Assertion
CBSP (cb , P) = λ st → Σ State (λ st₀ → st₀ ⊢ P → [[ cb ]]cb st₀ ≡ st)

CBLSP : (ParallelProgram × Predicate) → Assertion
CBLSP (S , P) = λ st → All (λ cb → st ⊩ (CBSP (cb , P))) S
```

$$CBLSP(S, Q)(st) = \bigwedge_{s_i \in S} CBSP(s_i, Q)(st) \tag{5.6}$$

Using these constructions, the specification statements from the original material can be created. These are given with a short verbal explanation first, then they are given with mathematical notation, which is followed by their Agda representation in our formalization.

Unless: $P \rhd_S Q$ represents a safety condition with the semantic meaning that if the predicate P is true in any given state of the program, then it either stays satisfied, or Q becomes true. In other words, the truth set of P can only be left through the truth set of Q.

$$\frac{P \wedge \neg Q \Rightarrow CBLWP(S, P \vee Q)}{P \rhd_S Q} \tag{5.7}$$

```
Unless : ParallelProgram → Predicate → Predicate → Statement
Unless S P Q = ⟦ (P △ (¬ Q)) ⟧a ⇛ (CBLWP (S , P ▽ Q))

infix 4 _▷[_]_
_▷[_]_ : Predicate → ParallelProgram → Predicate → Statement
P ▷[ S ] Q = Unless S P Q
```

Progress: $P \rightarrowtail_S Q$ means that there is at least one conditional batch in the program through which we can move from a state which satisfies P but not Q to a state that satisfies Q.

$$\frac{\exists s_i \in S : P \wedge \neg Q \Rightarrow CBLWP(s_i, Q)}{P \rightarrowtail_S Q} \tag{5.8}$$

```
Progress : ParallelProgram → Predicate → Predicate → Statement
Progress S P Q = Any (λ cb → ⟦ P △ ¬ Q ⟧a ⇛ (CBWP (cb , Q))) S

infix 4 _↣[_]_
_↣[_]_ : Predicate → ParallelProgram → Predicate → Statement
P ↣[ S ] Q = Progress S P Q
```

Ensures: $P \mapsto_S Q$ means that we can only leave the truth set of P through Q and there exists a conditional batch through which this can actually happen. (And due to the impartial scheduling, see Definition 3.1.2, it will eventually be executed, so being in the truth set of P essentially ensures that we will directly step into the truth set of Q at some point.)

$$\frac{P \rhd_S Q \quad P \rightarrowtail_S Q}{P \mapsto_S Q} \tag{5.9}$$

```
Ensures : ParallelProgram → Predicate → Predicate → Statement
Ensures S P Q = (Unless S P Q × Progress S P Q)

infix 4 _↦[_]_
_↦[_]_ : Predicate → ParallelProgram → Predicate → Statement
P ↦[ S ] Q = Ensures S P Q
```

Inevitable: $P \hookrightarrow_S Q$ is the transitive disjunctive closure (see Definition 3.1.5) of $P \mapsto_S Q$. This definition results in a statement which states that if we ever step into the truth set of P, we will inevitably have to reach a state in the truth set of Q, even if through numerous intermediate steps. In this sense, it is analogue to the $\Diamond$ *(future)* operator of Linear Temporal Logic.

This statement is special in the sense, that it is defined with its own inductive datatype instead of evaluating to lower level constructs. This is because the elements of a transitive disjunctive closure of a relation can be created in three different ways.

- They can be lifted from the base set. This rule makes sure that the resulting relation contains all the pairs from the initial set. It is so to speak, the base case, since all the other constructors have elements of the relation as parameters as well, but in this one it only appears in the result.

$$\frac{P \mapsto_S Q}{P \hookrightarrow_S Q} \tag{5.10}$$

- They can be the result of a transition through another state.

$$\frac{P \hookrightarrow_S Q \quad Q \hookrightarrow_S R}{P \hookrightarrow_S R} \tag{5.11}$$

- Or they can be the result of a conjunction.

$$\frac{P \hookrightarrow_S R \quad Q \hookrightarrow_S R}{P \vee Q \hookrightarrow_S R} \tag{5.12}$$

```
data Inevitable : ParallelProgram → Predicate → Predicate → Statement where
  FromEnsures : Ensures S P Q → Inevitable S P Q
  Transitivity : ((Inevitable S P Q) × (Inevitable S Q R)) → Inevitable S P R
  Disjunctivity :
    ((Inevitable S P R) × (Inevitable S Q R)) → Inevitable S (P ▽ Q) R

infix 4 _↪[_]_
_↪[_]_ : Predicate → ParallelProgram → Predicate → Statement
P ↪[ S ] Q = Inevitable S P Q
```

Invariant: A P predicate is invariant in an $I = (s_0, S)$ initialized program with the Q initial predicate if Q implies P, so it must be true at the beginning, and being in the truth set of P guarantees that we will forever stay there.

$$Invariant_I(Q, P) = (BSP(s_0, Q) \Rrightarrow [\![P]\!]a \wedge [\![P]\!]a \Rightarrow CBLWP(S, P)) \tag{5.13}$$

```
Invariant : InitializedProgram → Predicate → Predicate → Statement
Invariant I@(s₀ , S) Q P = ((BSP (s₀ , Q)) ⇛ ⟦ P ⟧a) × (⟦ P ⟧a ⇛ CBLWP (S , P))

infix 4 _∈inv[_/_]
_∈inv[_/_] : Predicate → InitializedProgram → Predicate → Statement
P ∈inv[ I / Q ] = Invariant I Q P
```

Here we also introduce the so-called *"Strongest Invariant"* assertion, which represents the smallest subset of the state space and is defined as the conjunction of all other invariants.

```
INVARIANT : InitializedProgram → Predicate → Assertion
INVARIANT I Q = λ st → ∀{P} → Invariant I Q P → st ⊢ P

infix 4 INV[_/_]
INV[_/_] : InitializedProgram → Predicate → Assertion
INV[ I / Q ] = INVARIANT I Q
```

Always True: A predicate is called always true, if it is implied by the strongest invariant. This means that we can rely on never leaving the truth set of these type of predicates, but they are not necessarily invariants themselves, since they can be weaker (meaning that they have a bigger truth set) and thus contain states, which are essentially inaccessible, but would lead to the chance of leaving the truth set of the predicate.

```
True : InitializedProgram → Predicate → Predicate → Statement
True I P Q = INVARIANT I Q ⇛ ⟦ P ⟧a

infix 4 _∈true[_/_]
_∈true[_/_] : Predicate → InitializedProgram → Predicate → Statement
P ∈true[ I / Q ] = True I P Q
```

Fixed Point: We call a certain state a fixed point (or fixpoint) of a program, if no execution of a conditional batch can further change the values of variables, because either their conditions are not satisfied, or their assignments all result in equal values being reassigned, which means that execution can be halted. Since the set of fixpoints is a subset of the state space, it is given here as an assertion. (Note: φ_S can also be used as a shorter notation of $Fixpoint_S$.)

$$Fixpoint_S(st) = \forall s_i \in S : [\![s_i]\!]cb(st) = st \tag{5.14}$$

```
Fixpoint : ParallelProgram → Assertion
Fixpoint S = λ st → All (λ cb → st ≡ ⟦ cb ⟧cb st) S

infix 4 φ[_]
φ[_] : ParallelProgram → Assertion
φ[ S ] = Fixpoint S
```

The specification statement based on this assertion is capable of describing constraints about the state in which a program can end by giving predicates which must be fulfilled by every state in the set of fixed points.

$$FIXPOINT_S(P) = Fixpoint_S \Rrightarrow [\![P]\!]a \tag{5.15}$$

```
FIXPOINT : ParallelProgram → Predicate → Statement
FIXPOINT S P = Fixpoint S ⇛ ⟦ P ⟧a

infix 4 _∈FP[_]
_∈FP[_] : Predicate → ParallelProgram → Statement
P ∈FP[ S ] = FIXPOINT S P
```

Guaranteed Termination: A predicate guarantees the termination of a program (by which we mean reaching a fixed point), if there exists another predicate, to which it inevitably leads and the truth set of that predicate is a subset of the set of fixed points. (This could later be simplified by adding a variation of the Inevitable statement that accepts assertions as its parameters.)

$$Termination_S(P) = \exists Q \in Predicate : P \hookrightarrow_S Q \wedge [\![Q]\!]a \Rrightarrow Fixpoint(S) \tag{5.16}$$

```
Termination : ParallelProgram → Predicate → Statement
Termination S P = Σ Predicate (λ Q → (P ↪[ S ] Q × ⟦ Q ⟧a ⇛ Fixpoint S))

infix 4 _∈TERM[_]
_∈TERM[_] : Predicate → ParallelProgram → Statement
P ∈TERM[ S ] = Termination S P
```

5.1.4 Constructions

Among the possible program constructions we have so far only formalized the simplest one, the union, that is also most likely to be useful, since it describes multiple programs working on a common, shared state space, which is a common occurrence in larger software systems.

Union

In a mathematical sense, the union of two parallel programs can simply be constructed by taking the union of the sets of conditional batches they are represented by. (Their initialized counterparts are more complicated.)

$$S = S_1 \cup S_2 \tag{5.17}$$

Because of the modification of the original model, that resulted in removing s_0 from the core **ParallelProgram** definition, we can simply just define the formal union of the parallel programs as their concatenation, since they are essentially just lists of conditional batches. This might result in the duplication of certain batches, but it does not affect the overall semantics, since impartial scheduling (see Definition 3.1.2) still applies.

```
Union : (S₁ S₂ : ParallelProgram) → ParallelProgram
Union S₁ S₂ = (S₁ ++ S₂)

infixr 4 _∪_
_∪_ : (S₁ S₂ : ParallelProgram) → ParallelProgram
S₁ ∪ S₂ = Union S₁ S₂
```

5.1.5 The notion of conformance to a specification

Basically everything that would be necessary for proving conformance to a specification is in place, but due to time constraints, it has not been discussed yet. The constructions that would support such a procedure are divided into two parts. The first one is the concept of a **Specification** record, which contains all the constraints and the known initialization properties under which those would need to be verified.

```
record Specification : Set where
  constructor mkSpecification
  field
    unless : List (Predicate × Predicate)
    ensures : List (Predicate × Predicate)
    inevitable : List (Predicate × Predicate)
    fixpoint : List Predicate
    termination : List Predicate
    initial : List Predicate
    invariant : List Predicate
```

The second part is the **Conforms** function, which accepts a **Specification** and a **ParallelProgram** as arguments and combines them into a type, that claims that the given program conforms to the specification. If an instance for the generated type can be constructed, the conformance can be considered to be verified.

```
Conforms : Specification → InitializedProgram → Set
Conforms specification I@(s₀ , S) =
  let
    INIT : Predicate
    INIT = Data.List.foldr (λ acc curr → curr △ acc) TRUE initial
  in
    (
      All (λ { (P , Q) → P ▷[ S ] Q }) unless
      ×
      All (λ { (P , Q) → P ↦[ S ] Q }) ensures
      ×
      All (λ { (P , Q) → P ↪[ S ] Q }) inevitable
      ×
      All (λ { P → φ[ S ] ⇛ ⟦ P ⟧a }) fixpoint
      ×
      All (λ { P → P ∈TERM[ S ] }) termination
      ×
      All (λ { P → P ∈inv[ I / INIT ] }) invariant
    )
      where open Specification specification
```

5.2 Proofs

In the second part of this chapter we will present the proofs we built on top of the previously introduced model. There are two kinds of statements that we could prove in our system. First, there are general theorems, which are true independently and then there are the correctness properties of certain specific programs.

5.2.1 Theorems

We show two proofs, one for a relatively simple lemma, for which simple straightforward reasoning is sufficient, and another one for a more complex theorem, that includes induction.

Reflexivity of Inevitability ($\hookrightarrow$)

The lemma states that every predicate inevitably leads to itself. This can be trivially seen intuitively by looking at the definitions and understanding the semantics behind them, but still requires quite a bit of work to write a fully formal proof for.

First we need to introduce the *NonEmpty* property for programs, since this lemma only holds, if there is at least one conditional batch in a program.

```
NonEmpty : ParallelProgram → Set
NonEmpty S = ¬ (S ≡ [])
```

Notice, that the reflexiveness does not only hold for the **Inevitable** ($\hookrightarrow$) property, but also for the **Ensures** ($\mapsto$), **Progress** ($\rightarrowtail$) and **Unless** ($\triangleright$). Since this list is laid out in such a way, that the former elements can always be composed from the latter, we should be able to work our way backwards.

First, the reflexivity of Unless is proven using the eliminator of $\bot$. (Recall from the Agda introduction part at the end of Chapter 3, that a negation maps to bottom, so if we have proof for a type, for which we have a negation at the same time, we can construct an impossible element of the empty type.) This is a special function, that helps in the implementation of branches that contain contradictions. (See section A.6 of Appendix A for its definition.) This is quite easy, since by unfolding the definition of Unless, we get a contradiction among the parameters.

```
▷-Reflexive : P ▷[ S ] P
▷-Reflexive (p , ¬p) = ⊥-elim (¬p p)
```

As the second step, we prove the reflexivity of Progress. For this step we need the *NonEmpty*-ness as a parameter, since there can be no progress made in an empty program. We pattern match on the list of the conditional batches in S, we eliminate the branch where it is the empty list, since that contradicts with our hypothesis, that S is *NonEmpty*, and we solve the second branch by getting the contradiction again, this time by the definition of the Progress statement.

```
≻→-NonEmpty-Reflexive : NonEmpty S → P ≻→[ S ] P
≻→-NonEmpty-Reflexive {[]} nonEmptyS = ⊥-elim (nonEmptyS refl)
≻→-NonEmpty-Reflexive {cb :: cbs} nonEmptyS = here (λ { (p , ¬p) → ⊥-elim (¬p p) })
```

Since the definition of Ensures is just a Cartesian product of Unless and Progress, we can take the next step by simply combining our previous results.

```
↦-NonEmpty-Reflexive : NonEmpty S → P ↦[ S ] P
↦-NonEmpty-Reflexive nonEmptyS =
  (▷-Reflexive , (≻→-NonEmpty-Reflexive nonEmptyS))
```

Now, that we have the reflexivity of Ensures, our goal can be directly achieved through a single constructor use, `FromEnsures`, which creates an Inevitability proof in one step.

```
↪-NonEmpty-Reflexive : NonEmpty S → P ↪[ S ] P
↪-NonEmpty-Reflexive nonEmptyS = FromEnsures (↦-NonEmpty-Reflexive nonEmptyS)
```

PSP

Our second example is the most complicated theorem we managed to prove. It was the theorem of "PSP" (see equation 5.18), meaning "Progress-Safety-Progress" (introduced in this form in [37]), which states that if from a set of starting points P a program must eventually reach a set of result states Q and the truth set of a given predicate R cannot be left without entering B then in case we begin execution from a shared point of P and R we will either get to a result (Q) while staying in R or enter an error state, B. We prove several other helpful lemmas first, that facilitated the construction of the bigger proof later.

$$\frac{P \hookrightarrow_S Q \quad R \rhd_S V}{P \wedge R \hookrightarrow_S (Q \wedge R) \vee V} \tag{5.18}$$

For the construction of this proof we used structural induction. We divided the goal into three parts based on how $P \hookrightarrow_S Q$ was constructed (from an Ensures statement or by one of the transitivity and distributivity rules).

The first goal is then solved by proving that equation 5.19 holds and then reconstructing the goal by using the `FromEnsures` constructor on the results of the lemma. The proof for the lemma can be seen in Section A.9 of Appendix A, it is constructed from two inductive parts, one for each of the Unless and the Progress statements that Ensures is made up of. They also use results of previously defined lemmas, the full source code for those is available at our repository[38].

$$\frac{P \mapsto_S Q \quad R \rhd_S V}{P \wedge R \mapsto_S (Q \wedge R) \vee V} \tag{5.19}$$

```
PSP : ((P ↪[ S ] Q) × (R ▷[ S ] V)) → (P △ R) ↪[ S ] ((Q △ R) ▽ V)

PSP (FromEnsures ensures , r▷[s]v) = FromEnsures (pspFromEnsures ensures r▷[s]v)

PSP (Transitivity (p↪[s]p₁ , p₁↪[s]q) , r▷[s]v) =
  Transitivity (
    (PSP (p↪[s]p₁ , r▷[s]v))
    ,
    Disjunctivity (
      (PSP (p₁↪[s]q , r▷[s]v))
      ,
      (↪-NonEmpty-from-⇒ (↪-NonEmpty p↪[s]p₁) inj₂)
    )
  )

PSP (Disjunctivity (p₁↪[s]q , p₂↪[s]q) , r▷[s]v) =
  ↪-⇐⇒-left
    (⇐⇒Symmetric andDistributiveRight)
    (Disjunctivity (PSP (p₁↪[s]q , r▷[s]v) , PSP (p₂↪[s]q , r▷[s]v)))
```

The second and third subgoals, whose constructors were inductive, can be solved by recursively applying the theorem (as well as a few other lemmas, for some of which the definition can be found in Appendix A) to the now structurally smaller Inevitable relations.

Safety property for union of programs

Another proof we created is regarding the **Unless** safety property of the union of two programs. The theorem states that the **Unless** relation for the union is equal to the union of the two separate relations. $\triangleright_{S_1 \cup S_2} = \triangleright_{S_1} \cup \triangleright_{S_2}$

This equality can be represented in Agda by proving the logical equivalence of two types. One representing the statement $P \triangleright_{S_1 \cup S_2} Q$ and the other claiming that $(P \triangleright_{S_1} Q) \wedge (P \triangleright_{S_2} Q)$. This logical equalence can be proven by giving two functions that represent the mapping between the two types in each direction.

First we show that the conjunction of the two statements in the individual programs implies the relation for their union. This is done by pattern matching on the first program. If the first program (S_1) is empty, we can simply just return the proof of the second one ($\triangleright_{S_2}$), if it is not, then we need to recursively reconstruct the proofs for the elements of it to be appended to the proofs for the second program. This can be achieved by gaining the parameters $(p, \neg q)$ from the definition of the **Unless** relation of the union and passing them on to the proof function of the first program ($\triangleright_{S_1}$) that we also received as an argument.

```
▷-Union-to : (P ▷[ S₁ ] Q × P ▷[ S₂ ] Q) → (P ▷[ S₁ ∪ S₂ ] Q)
▷-Union-to {S₁ = []} (▷s₁ , ▷s₂) = ▷s₂
▷-Union-to {S₁ = _ :: _} (▷s₁ , ▷s₂) (p , ¬q) with ▷s₁ (p , ¬q)
▷-Union-to {S₁ = _ :: _} (▷s₁ , ▷s₂) (p , ¬q) | px :: x =
  px :: ▷-Union-to (lessUnless ▷s₁ , ▷s₂) (p , ¬q)
```

The first part of the other direction provides the left side of the conjunciton. If S_1 is empty, then it needs no proofs, otherwise it can be recursively extracted and rebuilt from the proof of the union in a similar way as it was shown previously.

```
▷-Union-from-1 : (P ▷[ S₁ ∪ S₂ ] Q) → (P ▷[ S₁ ] Q)
▷-Union-from-1 {S₁ = []} ▷u = const []
▷-Union-from-1 {S₁ = _ :: _} ▷u (p , ¬q) with ▷u (p , ¬q)
▷-Union-from-1 {S₁ = _ :: _} ▷u (p , ¬q) | px :: x =
  px :: ▷-Union-from-1 (lessUnless ▷u) (p , ¬q)
```

The second part of the other direction can be achieved by essentially step by step recursively getting rid of the unnecessary proofs included for the first program. If S_1 is empty, then there is nothing to get rid of.

```
▷-Union-from-2 : (P ▷[ S₁ ∪ S₂ ] Q) → (P ▷[ S₂ ] Q)
▷-Union-from-2 {S₁ = []} ▷u = ▷u
▷-Union-from-2 {S₁ = _ :: _} ▷u (p , ¬q) = ▷-Union-from-2 (lessUnless ▷u) (p , ¬q)
```

At last, we just need to combine our previous results and we get a full proof for the stated theorem.

```
▷-Union : (P ▷[ S₁ ∪ S₂ ] Q) ↔ (P ▷[ S₁ ] Q × P ▷[ S₂ ] Q)
▷-Union = (λ x → ((▷-Union-from-1 x) , (▷-Union-from-2 x))) , ▷-Union-to
```

5.2.2 Correctness

We have also formalized proofs for a few properties of the concurrent bubble sort algorithm introduced in the previous section (see Subsection 5.1.2).

First element

As a quick example we managed to prove that if the first element of the array is one, it can later only change to zero. Using mathematical notation it can be expressed as seen in Equation 5.20. (Where a refers to the first variable in the environment that we defined for the program.)

$$a[0] = 1 \rhd_{BubbleSort_n} a[0] = 0 \tag{5.20}$$

The formalized versions of the predicates we need for the proof are illustrated below.

```
Before : Predicate
Before = EQ (v[ 0F ] g[ Const 0 ]) (Const 1)

After : Predicate
After = EQ (v[ 0F ] g[ Const 0 ]) (Const 0)
```

First we introduce a lemma stating that no number of the form `(suc _)` can be smaller than 1.

```
h1 : {x : ℕ} → suc x < 1 → ⊥
h1 (s≤s ())
```

After that we prove that the first instruction complies with the specification if its predicate is satisfied. (Notice, that it is the only instruction that can alter the value of the first element in the array.) To do so, we separate two cases by pattern matching on the value of the second element. If it is zero, then it is indeed smaller, so the instruction will exchange them and the `After` predicate, and thus the disjuction will be satisfied. In case it is not zero, we have a contradiction, since we assumed that it was smaller than one, but no natural number other then 0 can be smaller than one. (As we have already proven in `h1`.)

```
helper :
  (st ⊢ (Before △ ¬ After)) →
  ⟦ makePredicate 0 ⟧a st →
  ⟦ makeInstruction 0 ⟧i st ⊢ (Before ▽ After)
helper {st} (before , after) gt with (proj₂ (st 0F) 1)
helper {st} (before , after) gt | zero = inj₂ (ownRefl refl)
helper {st} (ownRefl before , after) gt | suc _ rewrite before = ⊥-elim (h1 gt)
```

Since the first instruction was the only one that could potentially change the first element, we can generalize the proof to all the instructions by calling the `helper` for the first one and stating that the `Before` predicate cannot be broken by any of the other instructions, a claim which Agda can check and automatically verify for us.

```
helper-n :
  (n : ℕ) →
  (st ⊢ (Before △ ¬ After)) →
  ⟦ makePredicate n ⟧a st →
  ⟦ makeInstruction n ⟧i st ⊢ (Before ▽ After)
helper-n {st} zero (before , after) gt = helper {st} (before , after) gt
helper-n {st} (suc n) (before , after) gt = inj₁ before
```

We can now build a function that recursively assembles the proof for all the instructions together.

```
all-helper :
  (n : ℕ) →
  (All (λ { (R , il) →
    ({st : State} → st ⊢ (Before △ ¬ After) → ⟦ R ⟧a st → (⟦ il ⟧il st) ⊢ (Before ▽ After))
  }) (bubbleSort n))
all-helper zero = []
all-helper (suc n) = ((λ {st} → helper-n {st} n) :: all-helper n)
```

Finally the `▷-proof` auxiliary function is used to construct the proof for our initial statement. (See Appendix A, more specifically Section A.10 for its implementation.)

```
test-n : (n : ℕ) → Before ▷[ bubbleSort n ] After
test-n n {st} = ▷-proof {Before} {After} (all-helper n)
```

Ordered result in fixpoint

As a more meaningful proof, we verified that if the program reaches a fixpoint, its results are indeed in order. (Note, that this does not necessarily mean, that it is correct, for that we would also need to prove that all the elements are preserved. That should be doable as well, since the only operation done on the elements is swapping two of them, which does not remove any of them, nor does it add any new ones.)

To express this property we introduced the parametric `Ordered` predicate. Given a number n it generates a predicate that claims that the first $n + 1$ elements of the array found at the first variable of the environment are in ascending order by stating that for each pair the one with the lower index is smaller or equal. Its mathematical notation can be seen in Equation 5.21, followed by its formalized representation in Agda.

$$Ordered(n) = \forall i \in \{1..n\} : a[i] \leq a[i+1] \tag{5.21}$$

```
Ordered' : ℕ → Predicate
Ordered' n = LTE (v[ 0F ] g[ Const n ]) (v[ 0F ] g[ Const (suc n) ])

Ordered : ℕ → Predicate
Ordered zero = TRUE
Ordered (suc n) = Ordered' n △ (Ordered n)
```

Using that, the statement we would like to prove can be expressed as seen in Equation 5.22.

$$\forall st \in State : Fixpoint_S(st) \rightarrow Ordered(n)(st) \tag{5.22}$$

We first introduce a lemma with the following claim. If the value of an element in the array stays the same after swapping it with the following element, their values must be equal. This can be proven by pattern matching on the values that introduce branching in the underlying semantics, eliminating (or when possible, omitting) the impossible or conflicting branches and through this process coming to the conclusion that our initial hypothesis is essentially equivalent to our goal.

```
rp-h4 : {n : ℕ} →
  proj₂ (st 0F) n ≡ proj₂ (⟦ makeInstruction n ⟧i st 0F) n →
  ⟦ v[ 0F ] g[ Const n ] ⟧e st ≡ ⟦ v[ 0F ] g[ Const (suc n) ] ⟧e st
rp-h4 {st} {n} eq with n Data.Nat.≟ (suc n)
rp-h4 {st} {n} eq | no ¬p with n Data.Nat.≟ n
rp-h4 {st} {n} eq | no ¬p | yes p = eq
rp-h4 {st} {n} eq | no ¬p | no ¬p₁ = ⊥-elim (¬p₁ refl)
```

Our next lemma transforms the results from the previous one using a congruence on the states.

```
rp-h5 : {n : ℕ} →
  st ≡ ⟦ makeInstruction n ⟧i st →
  ⟦ v[ 0F ] g[ Const n ] ⟧e st ≡ ⟦ v[ 0F ] g[ Const (suc n) ] ⟧e st
rp-h5 {st} {n} eq = rp-h4 {st} (cong (λ z → (proj₂ (z 0F) n)) eq)
```

It keeps the same outcome, but now it implies it from the hypothesis, which claims that in a certain state the instructions make no changes. This directly leads into the proof for the correct order of a single pair, which can be constructed with the help of an auxiliary function (`cb-helper`, see Section A.11 for definition), that extracts information from the fact that the execution of a conditional batch does not alter a state. This can either happen because the necessary predicate is not satisfied, or because the batch of instructions does not result in a change of any values.

If we divide the proof into these two cases, we can see, that if the condition requiring the greater than relation ($a[n] > a[n+1]$) was not true, then the opposite, a less than or equal relation ($a[n] \leq a[n+1]$) must hold. (We can deduce this conclusion with the help of a function from the Agda Standard Library[35], which provides several lemmas for mathematical properties of natural numbers.) On the other hand, if the batch gets executed and the values remain the same, we can use our previously defined lemma to determine that they must be equal ($a[n] = a[n+1]$), which in turn again implies the sought less than or equal ($a[n] \leq a[n+1]$) relation. (Again with the help of a lemma from the standard library.)

```
resultProof : {n : ℕ} →
  st ≡ ⟦ makePredicate n , [ makeInstruction n ] ⟧cb st →
  ⟦ Ordered' n ⟧a st
resultProof {st} {n} fp with (cb-helper {st} {makePredicate n} fp)
resultProof {st} {n} fp | inj₁ x with (≰⇒> x)
resultProof {st} {n} fp | inj₁ x | s≤s y = y
resultProof {st} {n} fp | inj₂ y = ≤-reflexive (rp-h5 y)
```

We can finally construct the proof for the entire array by recursively collecting the proofs for the individual pairs.

```
resultProof-n : {n : ℕ} → φ[ bubbleSort n ] ⇛ ⟦ Ordered n ⟧a
resultProof-n {zero} {st} [] = tt
resultProof-n {suc n} {st} (fp :: fps) =
  resultProof {st} {n} fp
  ,
  (resultProof-n {n} fps)
```

□

Chapter 6

Conclusions

Our formalization turned out to be adequate for proving theorems about the system, and while building proofs of correctness for more complicated programs is still a very tedious process, we have already managed to simplify it in several ways, which allowed us to verify partial properties. There might be even more opportunities available for making the implementation easier to work with that we have not noticed yet.

Find our source code at the `tdk` branch of our GitHub repository: `https://github.com/Isti115/orsi-formalization/tree/tdk`.

6.1 Further Work

So far we feel like a good foundation has been laid down, but it is not nearly complete and has many ways, in which it can and should be continued. We discuss some of these in this section.

6.1.1 Expanding the model

The original material of the subject has only been partially implemented. There are multiple pending additions and changes, that have raised questions we do not have good solutions for yet, but with more research and experimentation they should be able to be integrated into the current system.

For example while the semantics of data channels are originally described using assignments as well, but their actual behavior is sort of implicitly more constrained, they cannot be assigned arbitrary values, their operations act more like methods on an object. The best solution for expressing these constraints is not clear yet, one possibility is to implement them as separate instructions instead of expressions.

Another case, which the original material does not explicitly discuss is the out of bounds indexing of arrays. As this is basically undefined behavior, our current implementation handles it by indexing them with natural numbers and returning a default value when the index is higher than the length, but a cleaner solution could be achieved by indexing them with the appropriate finite set for their length. This would require changing the current representation of arrays:

```
evaluateType (Array A) = ℕ × (ℕ → (evaluateType A))
```

A dependent pair would be sufficient, where the second function's domain is controlled by the first element representing the length:

```
evaluateType (Array A) = Σ ℕ (λ l → (Fin l → (evaluateType A)))
```

The problem raised by this change is that we would need to ensure that all indexing done during the execution of the program stays within the bounds of the arrays, which is hard to enforce, since indexes can not only be given by constants during the construction of a program, but also dynamically by evaluating expressions, over the results of which we have little to no control.

Some other constructions, like intersection of other programs, which we did not have enough time to address are still missing as well, these could be added and the theorems based on them could be proven.

6.1.2 Optimization for education

Since our long term goals include the potential introduction of this formalization as a tool for teaching the subject, we aim to create ways in which it can be suitable for such a purpose. A specific type of task that is taught in practical lessons and serves as an exercise during evaluations is a problem, where the students are given certain parametric statements, the truth of which are assumed, and from those they have to decide if the validity of another statement can be proven, and if so, they have to give a deduction.

The definition of such tasks can already be performed in our current system, for example by giving a function with the function signature describing the problem, with its parameters being the assumptions and its result type representing the statement in question. If the body of the function, and thus the deduction can be constructed, it means that the implication holds. This could be utilized for either providing opportunity for the students to practice on their own with their computers providing them interactive help and being able to check the correctness of their own work, or if it is successfully introduced in classes, it may even be suitable for examinations with automatic grading.

The raw proofs for theorems and program properties are currently probably too complicated to be easily explained, but we have already started developing several helper functions that hide the lower level implementation and aim to provide a higher level interface, that could be used by students without having to understand all the underlying details.

6.1.3 Automation

Agda in itself already provides quite a good experience for interactively developing proofs using holes placed into the code and giving information about the types that are needed to fill them, but its auto solver is fairly limited to simple cases where the goal can easily be filled with an expression generated from the variables available in the close context. By expanding its search using the reflection capabilities of the language with methods specific to these tasks, more arguments could potentially be made implicit, and the verbosity of the otherwise sometimes inconveniently long proofs could be reduced.

6.1.4 Generating executable code

Another possible addition to the current system would be a way to export formal programs into a format that can either be further compiled or executed directly in an interpreter. This way it would be possible to experiment with the programs after proving their adherence to the given specification.

Chapter 7

Acknowledgements

7.1 Personal

I would like to express my gratitude towards my two supervisors, namely Melinda Tóth and Ambrus Kaposi, who have both motivated me towards this project, provided the necessary background knowledge and were available when I was asking for advice, even under unusual circumstances.

The constructive criticism of the manuscript by Mária Donkó and all the other help from others are also whole-heartedly appreciated.

7.2 EFOP

The author was supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.3-VEKOP-16-2017-00002).

Bibliography

[1] Alan M Turing. Lecture to the london mathematical society on 20 february 1947.

[2] Robin Candy. Towards concurrent hoare logic. Nov 2012.

[3] Dániel Horpácsi, Péter Bereczky, István Donkó, Ambrus Kaposi, and Dávid János Németh. Interactive teaching of programming language theory with a proof assistant. *Central-European Journal of New Technologies in Research, Education and Practice*, 2020.

[4] Erik Bergsten, Oskar Bjärne Larsson, Tobias Rastemo, Oskar Rutqvist, and Anderas Standar. Methods for using Agda to prove safety and liveness for concurrent programs. 2017.

[5] Jayadev Misra. A foundation of parallel programming. In *Constructive Methods in Computing Science*, pages 397–445. Springer, 1989.

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

[7] Häggström Johan. Proof checker for extended linear time temporal logic proofs about small concurrent programs. 2018.

[8] Sidney Amani, June Andronick, Maksym Bortin, Corey Lewis, Christine Rizkallah, and Joseph Tuong. Complx: A verification framework for concurrent imperative programs. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2017, page 138–150, New York, NY, USA, 2017. Association for Computing Machinery.

[9] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.

[10] Norbert Schirmer. *Verification of sequential imperative programs in Isabelle/HOL*. PhD thesis, Technische Universität München, 2006.

[11] Prensa Nieto et al. *Verification of parallel programs with the Owicki-Gries and rely-guarantee methods in Isabelle/HOL*. PhD thesis, Technische Universität München, 2002.

[12] Susan Owicki and David Gries. An axiomatic proof technique for parallel programs i. *Acta informatica*, 6(4):319–340, 1976.

[13] Reynald Affeldt, Naoki Kobayashi, and Akinori Yonezawa. Verification of concurrent programs using the coq proof assistant: A case study. *IPSJ Digital Courier*, 1:117–127, 2005.

[14] Reynald Affeldt and Naoki Kobayashi. Formalization and verification of a mail server in coq. In Mitsuhiro Okada, Benjamin C. Pierce, Andre Scedrov, Hideyuki Tokuda, and Akinori Yonezawa, editors, *Software Security — Theories and Systems*, pages 217–233, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[15] Reynald Affeldt and Naoki Kobayashi. A coq library for verification of concurrent programs. *Electronic Notes in Theoretical Computer Science*, 199:17 – 32, 2008. Proceedings of the Fourth International Workshop on Logical Frameworks and Meta-Languages (LFM 2004).

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

[17] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, ii. *Information and Computation*, 100(1):41 – 77, 1992.

[18] Davide Sangiorgi and David Walker. *The pi-calculus: a Theory of Mobile Processes*. Cambridge university press, 2003.

[19] Barbara Heyd and Pierre Crégut. A modular coding of UNITY in Coq. In *TPHOLs*, 1996.

[20] Sjouke Mauw and Gert J. Veltink. A proof assistant for psf. In *CAV*, 1991.

[21] Ákos Fóthi and Zoltán Horváth. *Párhuzamos és elosztott programozás (English: Parallel and Distributed Programming)*. ELTE, 2005.

[22] Ákos Fóthi. *Bevezetés a programozáshoz (English: Introduction to programming)*. ELTE, 1983.

[23] Charles Antony Richard Hoare. Proof of correctness of data representations. In *Programming methodology*, pages 269–281. Springer, 1978.

[24] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

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

[26] E Allen Emerson and Jai Srinivasan. Branching time temporal logic. In *Workshop/School/Symposium of the REX Project (Research and Education in Concurrent Systems)*, pages 123–172. Springer, 1988.

[27] Ákos Fóthi. A mathematical approach to programming. *Annales Uni. Sci. Budapest de R. Eötvös Nom. Sectio Computatorica*, 9:105–114, 1988.

[28] Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *J. Funct. Program.*, 23:552–593, 2013.

[29] The Coq Development Team. The coq proof assistant, version 8.11.0, January 2020.

[30] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. Version 8.0.

[31] Agda Development Team. Agda.

[32] Ulf Norell. *Towards a practical programming language based on dependent type theory.* PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

[33] Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of agda–a functional language with dependent types. In *International Conference on Theorem Proving in Higher Order Logics*, pages 73–78. Springer, 2009.

[34] Ulf Norell. Dependently typed programming in agda. In *International school on advanced functional programming*, pages 230–266. Springer, 2008.

[35] Nils Anders Danielsson, Matthew Daggitt, and Guillaume Allais. The Agda standard library.

[36] Zoltán Horváth. Párhuzamos programozás reláció alapú modellje (English: Relational Model of Parallel Programming). 1996.

[37] K. Mani Chandy and Jayadev Misra. Parallel program design - a foundation. 1988.

[38] István Donkó, Ambrus Kaposi, and Melinda Tóth. An Agda formalization for a model of concurrent programs.

Appendix A

Extra parts of the source code

A.1 Definition of the type representing finite sets

```
data Fin : ℕ → Set where
  zero : {n : ℕ} → Fin (suc n)
  suc : {n : ℕ} (i : Fin n) → Fin (suc n)
```

A.2 Example implementation of **evaluateType**

```
evaluateType : Types → Set
evaluateType Nat = ℕ
evaluateType (Array A) = ℕ × (ℕ → (evaluateType A))
evaluateType (DataChannel A) = QueueWithHistory A
```

A.3 Assertion decidability

```
assertionDecidability : {P : Predicate} → {st : State} → ((¬ ⟦ P ⟧a st) ⊎ (⟦ P ⟧a st))
assertionDecidability {P} {st} with (⟦ P ⟧d st)
assertionDecidability {P} {st} | yes p = inj₂ p
assertionDecidability {P} {st} | no ¬p = inj₁ ¬p
```

A.4 De Morgan's law of union

```
notAndToOrNotNot : ¬ (P △ Q) ⇒ (¬ P) ▽ (¬ Q)
notAndToOrNotNot {st = st} ¬_p∧q_ with assertionDecidability {st = st}
notAndToOrNotNot ¬_p∧q_ | inj₁ ¬p = inj₁ ¬p
notAndToOrNotNot ¬_p∧q_ | inj₂ p = inj₂ (λ q → ¬_p∧q_ (p , q))
```

A.5 De Morgan's law of intersection

```
notOrToAndNotNot : ¬ (P ▽ Q) ⇒ (¬ P) △ (¬ Q)
notOrToAndNotNot ¬_pvq_ = ((¬_pvq_ ∘ inj₁) , (¬_pvq_ ∘ inj₂))
```

A.6 Eliminator of the empty type

```
⊥-elim : {A : Set} → ⊥ → A
⊥-elim ()
```

A.7 The Unless property is true for subsets of the program

```
lessUnless : (P ▷[ cb :: cbs ] Q) → (P ▷[ cbs ] Q)
lessUnless p▷[cb::cbs]q = λ p△¬q → lessCBLWP (p▷[cb::cbs]q p△¬q)
```

A.8 The right-hand distributivity of disjunction over conjunction

```
andDistributiveToRight : ((P ▽ Q) △ R) ⇒ ((P △ R) ▽ (Q △ R))
andDistributiveToRight (inj₁ p , r) = inj₁ (p , r)
andDistributiveToRight (inj₂ q , r) = inj₂ (q , r)

andDistributiveFromRight : ((P ▽ Q) △ R) ⇐ ((P △ R) ▽ (Q △ R))
andDistributiveFromRight (inj₁ (p , r)) = (inj₁ p , r)
andDistributiveFromRight (inj₂ (q , r)) = (inj₂ q , r)

andDistributiveRight : ((P ▽ Q) △ R) ⇐⇒ ((P △ R) ▽ (Q △ R))
andDistributiveRight = (andDistributiveToRight , andDistributiveFromRight)
```

A.9 PSP - `fromEnsures`

```
pspFromEnsures₁ : P ▷[ S ] Q → R ▷[ S ] V → (P △ R) ▷[ S ] (Q △ R ▽ V)
pspFromEnsures₁ p▷[s]q r▷[s]v _p△r_△¬q△r▽v_@((p , r) , ¬q△r▽v_)
  with (notOrToAndNotNot ¬q△r▽v_)
... | ¬q△r_ , ¬v with (r▷[s]v (r , ¬v))
... | [] = []
... | inj₁ r' :: rest with (p▷[s]q (p , λ q → ¬q△r_ (q , r)))
... | inj₁ p' :: rest' =
           inj₁ (p' , r')
           ::
           pspFromEnsures₁
             (lessUnless p▷[s]q) (lessUnless r▷[s]v) _p△r_△¬q△r▽v_
... | inj₂ q' :: rest' =
           inj₂ (inj₁ (q' , r'))
           ::
           pspFromEnsures₁
             (lessUnless p▷[s]q) (lessUnless r▷[s]v) _p△r_△¬q△r▽v_
pspFromEnsures₁ p▷[s]q r▷[s]v _p△r_△¬q△r▽v_@((p , r) , ¬q△r▽v_) | ¬q△r_ , ¬v
    | inj₂ v_ :: rest =
         inj₂ (inj₂ v_)
         ::
         pspFromEnsures₁
           (lessUnless p▷[s]q) (lessUnless r▷[s]v) _p△r_△¬q△r▽v_

pspFromEnsures₂ : P ▷[ S ] Q → R ▷[ S ] V → P ↣[ S ] Q → Progress S (P △ R) (Q △ R ▽ V)
pspFromEnsures₂ {P} {S = (cb :: cbs)} {Q} {R} {V} p▷[s]q r▷[s]v (here p△¬q⇛bcwp) = here f
  where
    f : ⟦ (P △ R) △ (¬ (Q △ R ▽ V)) ⟧a ⇛ CBWP (cb , Q △ R ▽ V)
    f ((p , r) , ¬q△r▽v_) with (notOrToAndNotNot ¬q△r▽v_)
    ... | ¬q△r_ , ¬v with (r▷[s]v (r , ¬v))
    ...   | inj₁ r' :: rest with (p▷[s]q (p , λ q → ¬q△r_ (q , r)))
    ...     | inj₁ p' :: rest' = inj₁ (p△¬q⇛bcwp (p , (λ q → ¬q△r_ (q , r))) , r')
    ...     | inj₂ q' :: rest' = inj₁ (q' , r')
    f ((p , r) , ¬q△r▽v_) | ¬q△r_ , ¬v -- ...
          | inj₂ v_ :: rest = inj₂ v_
pspFromEnsures₂ p▷[s]q r▷[s]v (there rest) =
  there (pspFromEnsures₂ (lessUnless p▷[s]q) (lessUnless r▷[s]v) rest)

pspFromEnsures : P ↦[ S ] Q → R ▷[ S ] V → (P △ R) ↦[ S ] (Q △ R ▽ V)
pspFromEnsures (p▷[s]q , p↣[s]q) r▷[s]v =
  (pspFromEnsures₁ p▷[s]q r▷[s]v , pspFromEnsures₂ p▷[s]q r▷[s]v p↣[s]q)
```

A.10 Auxiliary function for proving Unless (▷) statements

```
▷-proofHelper :
  (st ⊢ (P △ ¬ Q)) →
  ([[ R ]]a st → ([[ b ]]b st) ⊢ (P ▽ Q)) →
  (([[ (R , b) ]]cb st) ⊢ (P ▽ Q))
▷-proofHelper {st = st} {R = R} pq f with ([[ R ]]d st)
▷-proofHelper pq f | yes p = f p
▷-proofHelper pq f | no ¬p = inj₁ (proj₁ pq)

▷-proof :
  (All (λ { (R , b) →
    ({st : State} → st ⊢ (P △ ¬ Q) → [[ R ]]a st → ([[ b ]]b st) ⊢ (P ▽ Q))
  }) S) →
  P ▷[ S ] Q
▷-proof [] (p , ¬q) = []
▷-proof (prf :: prfs) (p , ¬q) =
  ▷-proofHelper (p , ¬q) (prf (p , ¬q))
    :: (▷-proof prfs (p , ¬q))
```

A.11 Auxiliary function for fixed point proofs

```
cb-helper : (st ≡ [[ (P , b) ]]cb st) → (¬([[ P ]]a st) ⊎ (st ≡ [[ b ]]b st))
cb-helper {st} {P} eq with [[ P ]]d st
cb-helper eq | yes p = inj₂ eq
cb-helper eq | no ¬p = inj₁ ¬p
```