School of Computer Science and Engineering

Faculty of Engineering

The University of New South Wales

Mechanised Bilateral Proof in Isabelle/HOL

by

Jessica Theodosius

Thesis submitted as a requirement for the degree of
Bachelor of Engineering in Software Engineering

Submitted:	May 2019	Student ID:	z5057844
Supervisor:	Gerwin Klein	Topic ID:	N/A

Abstract

This thesis attempts to prove the soundness of Bilateral Proof by mechanising a representative subset of Bilateral Proof in Isabelle/HOL.

Bilateral Proof is a verification framework written by Jayadev Misra for concurrent programs with shared variables. This framework appears to be simpler than the existing frameworks that prove concurrent programs. Bilateral Proof also offers some nice properties, namely compositional and the ability to prove both safety and progress properties. However, no one had attempted to prove the soundness of this logic prior to this work.

Results show that the rules formalised are sound under certain assumptions found throughout the formalisation.

Acknowledgements

I would like to thank Gerwin Klein and Carroll Morgan for their help and guidance throughout this thesis. A very special thank you to my family who have been extremely supportive in this journey. Finally, a big thank you to all my friends who have always helped me in a lot of ways.

Contents

Chapter 1

Introduction

The concept of concurrent programming has been around for at least 50 years. With concurrency, multiple programs are able to run simultaneously. Concurrency also allows an increase in CPU utilisation and enables us to write responsive systems.

Despite the benefits, writing concurrent programs is incredibly hard, let alone proving their correctness. This is due to the fact that the executions of concurrent programs are almost always non-deterministic. There is a very large, or even an exponential number of possible executions a program can take. Therefore, testing methods such as traditional software testing and exhaustively analysing all possible behaviours (model checking) would not always work.

To overcome this, formal verification has been utilised as a means to prove the correctness of concurrent programs. One of the first methods was introduced in 1976 by Susan Owicki and David Gries, referred to as the Owicki-Gries method [OG76]. Owicki-Gries extended Hoare logic [Hoa69], which proves the correctness of sequential programs, to include parallelism. Various other frameworks had then been discovered, such as the Rely/Guarantee method [Jon81] which is compositional, and UNITY [CM88] that proves safety and progress properties for a limited set of program constructs.

As recently as 2017, a new verification method for concurrent programs was published by Jayadev Misra, named Bilateral Proof [Mis17]. This compositional framework allows verification of programs for a given specification with the ability to also prove safety and progress properties. However, this logic had not been proved sound. Soundness of a logic means that all statements which can be derived are true. That is, the logic had been proposed, but it had not yet been proved that it actually works.

This thesis aims to first mechanise a representative subset of Bilateral Proof in Isabelle/HOL [NPW02]. Afterwards, Isabelle is used to prove the soundness of the formalised rules.

Chapter 2 provides some background on concurrency. Chapter 3 reviews the existing proof methods for concurrent programs. Chapter 4 outlines a summary of Bilateral Proof. Chapter 5 gives a brief introduction to Isabelle/HOL in order to understand the work in this thesis. Chapter 6 presents the formalisation and soundness proof of Bilateral Proof in Isabelle/HOL. Chapter 7 evaluates the formalisation and the soundness proof. Finally, Chapter 8 summarises the thesis and gives suggestions for any future work beyond this thesis.

Chapter 2

Concurrency

2.1 Background

In concurrent programs, several tasks are carried out simultaneously by having multiple threads. Each of the threads has its own program counter that keeps track of which instruction to execute next [Tan09, p. 99].

There are three different categories of concurrent applications: multithreaded, multiprocessor, and distributed.

A multithreaded application is executed in a single processor. It is decomposed into multiple threads and executed in an interleaving manner. Here, each thread acts as if it is being executed alone when in fact the processor is switching rapidly between the threads.

A multiprocessor application utilises multiple processors to run multiple threads. Here, we treat the concurrency of multithreaded and multiprocessor applications the same way.

Finally, in a distributed setting, an application is split into several separate machines.

2.2 Scheduling and Fairness

Unlike sequential programs, concurrent programs may have multiple threads that are ready to be executed at the same time [Tan09, p. 143-148]. For that reason, there is a need for a method to decide which thread to run at a subsequent time, i.e. scheduling. The part of the operating system that performs this is called the scheduler.

One of the features a scheduling algorithm should have is fairness, i.e. to give a fair share of a processor for each thread. A typical scheduling algorithm to ensure fairness is Round-Robin scheduling [Tan09, p. 152], whereby each thread is assigned a time interval during which it is allowed to run. At the end of each interval, the processor executes the next thread, and so on.

The concept of fairness will be useful later when discussing progress properties.

2.3 Communication

In order for the threads to work with each other, they need to communicate and synchronise with each other. This can be done with either shared variables or message passing.

Shared variables are used for multithreaded and multiprocessor applications. They have a common memory that each thread can read and write to. These operations allow threads to interfere with each other, and hence unexpected behaviours might occur.

On the other hand, interference is not a concern in message passing. This is due to the fact that the threads communicate by exchanging messages over a network. This method is used by distributed applications.

This thesis concerns only concurrent programs with shared variables.

2.4 Safety and Liveness Properties

To prove the correctness of a concurrent program, Lamport [Lam77] introduced two types of properties, safety and liveness.

A safety property asserts that something bad will not happen. A real-life example of safety is “vehicles must stop when the traffic light is red”. An example of a safety property in programming is partial correctness. A program is said to be partially correct if, whenever the program begins in a state that satisfies the precondition, then the postcondition holds if the program terminates.

A liveness property states that something good must happen. A possible liveness property for traffic lights is "a traffic light will eventually turn green". In terms of programs, an example is "a program will produce a result". In other words, the program must terminate.

There is also a restricted class of liveness called progress. A progress property asserts that an action is eventually executed. Fairness of the scheduler is required to ensure progress. If fairness is not enforced, a process might never be executed and thus it will not be able to make any progress.

Chapter 3

Existing Proof Methods for Concurrent Programs

3.1 Owicki-Gries

3.1.1 Background

The following is a simple program where two threads execute an atomic action $x := x+1$ concurrently. An atomic action is either performed completely without any interference from other threads or not performed at all.

$$\begin{array}{ccc} \{x = 0\} & & \{x = 0\} \\ x := x + 1 & || & x := x + 1 \\ \{x = 1\} & & \{x = 1\} \end{array}$$

$||$ is a parallel composition and works by choosing a thread non-deterministically and executing its current instruction.

In the program above, the statements express what the program must do and the assertions enclosed in curly braces tell what must be true when the program is at that point. These are the concepts from Hoare logic. The assertion above a statement is called precondition and the assertion below a statement is called postcondition.

It is rather intuitive to take the conjunction of the pre and postconditions in order to prove the correctness of the whole program, as shown below.

$$\begin{array}{c}\{x = 0 \wedge x = 0\} \\ x := x + 1 \quad || \quad x := x + 1 \\ \{x = 1 \wedge x = 1\}\end{array}$$

However, this is incorrect. We know that after the execution of the program, it will terminate with $x = 2$. Simply taking the conjunction does not work because the threads interfere with each other. A more sophisticated proof method is required, and the Owicki-Gries method [OG76] was one of the very first to solve this problem and is perhaps the best known. It was published in 1976 by Susan Owicki and David Gries, with the idea of extending Hoare logic for parallel programs.

3.1.2 Proof Theory

There are two steps in proving the correctness of parallel programs using the Owicki-Gries method. First, prove local correctness and then prove *interference freedom.*

Local correctness refers to the correctness of each thread as seen before with the Hoare logic assertions. This can easily be proved using Hoare logic rules.

Interference freedom is required to prove global correctness, which is the correctness of the whole program. With interference freedom, assertions must remain valid even upon any execution of a statement in another thread. As a consequence, each statement in the program needs to be annotated.

Here is an example of both forms of correctness of the program $x := x + 1 \mid\mid x := x + 2$ with precondition $x = 0$.

$$\begin{array}{ccc} & \{x = 0\} & \\ \{P1 : x = 0 \vee x = 2\} & & \{P2 : x = 0 \vee x = 1\} \\ S1 : x := x + 1 & || & S2 : x := x + 2 \\ \{Q1 : x = 1 \vee x = 3\} & & \{Q2 : x = 2 \vee x = 3\} \\ & \{x = 3\} & \end{array}$$

It can be observed that each statement has its own assertions and proving the local correctness of each thread is trivial. There are however four more proof obligations for interference freedom, which are

$$\{P1 \wedge P2\}\ S2\ \{P1\}, \{P2 \wedge P1\}\ S1\ \{P2\}, \{Q1 \wedge P2\}\ S2\ \{Q1\}, \{Q2 \wedge P1\}\ S1\ \{Q2\}$$

These proof obligations are immediate and therefore the global correctness of this program is proved.

3.1.3 Limitations

While it is a nice and simple method, Owicki-Gries has two limitations. First, since each assertion needs to be checked with all statements from other threads, the number of proof obligations for testing interference freedom is quadratic in the number of statements across all threads. Second, this method is not compositional because it requires us to know the implementation of the whole program. A proof method referred to as Rely-Guarantee that overcomes these drawbacks will be discussed next.

3.2 Rely/Guarantee

3.2.1 Background

Jones [Jon81] proposed a compositional proof method referred to as Rely/Guarantee in 1981. He introduced a new notion of *rely* and *guarantee* conditions alongside the pre and postconditions from Hoare logic. Rely corresponds to the interference from the environment that a component can tolerate. This so-called environment refers to other threads/components that are running in parallel. Guarantee describes the interference of the component that can affect the environment.

In contrast to Owicki-Gries, there is no need for intermediate assertions to prove global correctness. Here, each component is required to have a single predicate consisting of its pre, rely, guarantee, and postcondition. These predicates are enough to prove the correctness of a program, and no internal representation of the program is needed. Thus, they make a compositional proof method.

3.2.2 Proof Theory

The specification of Rely/Guarantee consists of a quadruple $\{P, R, G, Q\}$ where P is the precondition, R is the rely condition, G is the guarantee condition, and Q is the postcondition. As discussed in [XdRH97], we say a component C satisfies such specification $C\ \{P, R, G, Q\}$
if

1. C starts in a state that satisfies P, and
2. any environment transition satisfies R,

then

3. any component transition satisfies G, and
4. if C terminates, the final state satisfies Q.

While a complete list of the proof rules will not be shown here, it is worth looking at a rule for parallel composition and how it deals with interference, taken from [Vaf08].

$$\frac{c_1\ \{P, R \vee G_2, G_1, Q_1\} \quad c_2\ \{P, R \vee G_1, G_2, Q_2\}}{c_1 \parallel c_2\ \{P, R, G_1 \vee G_2, Q_1 \wedge Q_2\}}$$

Let ε denote the environment of $c_1 \parallel c_2$. Because c_1 can be interfered with by either c_2 or ε, c_1 must be able to tolerate both interferences. Hence, the rely condition of c_1 is $R \vee G_2$. Similarly, the rely condition of c_2 is $R \vee G_1$.

Since every statement in $c_1 \parallel c_2$ is either from c_1 or c_2, it satisfies $G_1 \vee G_2$. In addition, the precondition P needs to hold for each component. It also follows that if both components terminate, then the conjunction of their postconditions, $Q_1 \wedge Q_2$, holds.

3.2.3 Limitations

Compared to Owicki-Gries, Rely/Guarantee proposes a better solution to prove the correctness of a program with respect to the specifications. The compositionality of Rely/Guarantee allows a separation between the implementation of a component and the proof of correctness. Additionally, safety properties proposed by Lamport [Lam77] can easily be proved by utilising the guarantee conditions to include safety properties. However, this method does not provide a way to prove liveness properties. A proof method called UNITY overcomes this issue.

3.3 UNITY

Because UNITY is the method most closely related to Bilateral Proof, we will look at it in some detail.

3.3.1 Background

Chandy and Misra [CM88, Mis95] introduced a framework called UNITY in 1988 which is able to prove safety and progress properties. In sequential programs, assertions are associated with specific program points. This is not the case with UNITY as it does not have program counters. Instead, safety and progress properties are associated with the entire program.

Unlike the other proof methods, UNITY does not support control flow. There is no sequential composition, if/else statements, nor loops. It only supports assignment statements to a list of variables and they can be guarded. We call these assignment statements *actions*. An example of a valid action is $x, y := 0, 1$ if $z > 0$.

In UNITY, a program consists of a number of threads running in parallel where each thread consists of a single action. During each execution step of the program, an action is selected non-deterministically and gets executed. This execution is also constrained by the fairness rule, in which each thread is selected infinitely often.

3.3.2 Safety Properties

Misra [Mis95] introduced an operator called **co** (stands for constraint) to express safety properties. p **co** q means that for any action, if p holds, then q holds after the execution of the action.

Formally, $p \textbf{ co } q \equiv \forall t.\ \{p\}\ t\ \{q\}$, where t is any action in the program and $\{p\}\ t\ \{q\}$ is a Hoare triple. Recall that the actions in UNITY are assignment statements to a list of variables that can be guarded.

Suppose we want to express a safety property "x never decreases". In other words, if x has a certain value m, it continues to hold until the value exceeds m. Using the **co** operator, we can write this as $\forall m.\ x = m \textbf{ co } x \geq m$.

3.3.3 Progress Properties

The primary operator for progress is called *leads-to*. In order to define leads-to, two simpler predicates, *transient* and *ensures*, were introduced.

A predicate is transient if it is guaranteed to be falsified by an execution of some single action in the program. Formally, $p \textbf{ transient} \equiv \exists s.\ \{p\}\ s\ \{\neg p\}$, where s is an action in the program.

With ensures or **en**, p **en** q means that if p holds at any point, then it continues to hold until q holds, and q holds eventually. Formally, $p \textbf{ en } q \equiv ((p \wedge \neg q) \textbf{ co } (p \vee q)) \wedge ((p \wedge \neg q) \textbf{ transient})$.

Finally, p leads-to q or $p \mapsto q$ means that if p holds at any point, then q holds eventually. Unlike ensures, it does not require p to hold until q holds.

3.3.4 Limitations

UNITY's approach to concurrent programs made it easier to prove correctness in terms of safety and progress. As programs are decomposed into single actions, proofs concern only the initial state, a single action, and its final state. However, the absence of control flow and program counter makes it very difficult to write a conventional sequential program. Explicit program counters are needed in order to do so.

Chapter 4

Bilateral Proof

This chapter summarises a proof method called Bilateral Proof which takes the best out of the existing proof methods presented so far.

4.1 Background

Bilateral Proof [Mis17] was published by Misra in 2017. It is a framework to prove concurrent programs with shared variables. The main features are its compositionality and the ability to verify program code. This method is largely inspired by UNITY and uses concepts from UNITY as we will see later.

This framework proves two types of properties, referred to as *terminal* and *perpetual* properties. A terminal property describes the postcondition for a given precondition. Perpetual properties are those that hold throughout an execution, which are safety and progress properties.

Finally, the proof itself is called *bilateral* because some of the rules employ terminal properties to derive perpetual properties and vice versa.

4.2 Program and Execution Model

4.2.1 Structure

The syntax of the programs according to Misra [Mis17] is described below.

$$\begin{array}{rll} & \textit{action} & ::= \text{guard} \rightarrow \text{body} \\ f,\ g :: & \textit{component} & ::= \textit{action} \mid f\ [\,]\ g \mid \text{seq}(f_0, f_1, ..., f_n) \\ & \textit{program} & ::= f \end{array}$$

A component can either be an *action*, a *join* of the form $f\ [\,]\ g$, or *seq*. An action consists of a guard predicate and a body. Join denotes parallel composition || we have seen before. We use the word "parallel composition" when talking about "join" in this document as they convey the same meaning. Seq, on the other hand, corresponds to any sequential language construct for which proof rules are available. Examples of seq include sequential composition (;), if/else statements and while loops.

A program is a component that is meant to be executed alone. That is, without any interference from other programs executing in parallel.

There is also a notion of *local variables*. A variable is local to a component if the component has exclusive write-access to that variable at any point of its executions. A *local predicate* of a component is a predicate in which all variables are local variables of that component.

4.2.2 Execution

In the execution of an action $b \rightarrow \alpha$, b is evaluated and if it is true, α is executed. An execution where the guard b holds is called an *effective execution*. In contrast, the guard does not hold in an *ineffective execution* and hence the program state is unaltered.

When a program is executed, initially the program counter is at the entry point of the program. In any given state, the scheduler chooses which thread to run next, and executes the action effectively or ineffectively. The execution follows the fairness constraint, in which every thread is chosen eventually, so that no component is ignored forever.

4.2.3 Example: Distributed Counter

The following is a program from [Mis17] that implements a distributed counter *ctr*.

```
**initially** $ctr = 0$
$f_j$ ::
  **initially** $old_j,\ new_j = 0,\ 0$
  **loop**
    $new_j := old_j + 1;$
    **if** [ $ctr = old_j \rightarrow ctr := new_j$
      | $ctr \neq old_j \rightarrow old_j := ctr$ ]
  **forever**
```

This program is a parallel composition of a finite number of threads f_j. It is particularly interesting as this way of implementing a counter is usually done in distributed computing. A common implementation in shared variables concurrency uses an atomic statement $ctr := ctr + 1$. In this case, unlike the program above, the correctness proof is straightforward since the atomicity gets rid of the interference problem.

As the program never terminates, we are not concerned about terminal properties in this example. However, we want to prove some perpetual properties. First, we want to show that *ctr* never decreases and it increases only by 1 at a time. This is a safety property. We also want to prove a progress property showing that *ctr* increases eventually.

4.3 Proof Theory

4.3.1 Program Specification

In the Bilateral Proof method, the specification of component f is of the form of the quadruple $\{r\}\ f\ \{Q \mid s\}$. r is the precondition, Q is the set of perpetual properties, and s is the postcondition.

For any execution of f starting in an r-state,

1. if the execution terminates, the end state is an s-state, and
2. every property in Q holds throughout the entire execution.

In terms of terminology, we write $\{r\}\ f\ \{s\}$ when Q is irrelevant in the discussion.

4.3.2 Local Annotation

A very important concept in Bilateral Proof is *local annotations.* Local annotations of a component associate predicates local to each point of the component.

Local annotations are very similar to the annotations in Owicki-Gries with the exception of being local. Similar to Owicki-Gries, each annotation acts as a precondition for the execution of each action in the component. Moreover, the annotations also yield a valid pre and postcondition for the component.

The main advantage of having local annotations is that the locality ensures there is no interference on the annotations. Therefore, unlike Owicki-Gries which requires us to prove both local correctness and interference freedom, it is enough to prove only the former in the Bilateral Proof method.

Proof Rules

There is a separate rule to construct local annotations for action, seq, and parallel composition.

The rules for seq are similar to the rules in Hoare Logic [Hoa69]. For example, here is the proof rule for sequential composition.

$$\frac{\{p\}\ f\ \{mid\} \qquad \{mid\}\ g\ \{q\}}{\{p\}\ f; g\ \{q\}}\ (semi)$$

Unlike in Hoare Logic, the rule here requires the assertions, i.e. p, mid, and q, and the annotations of each component, f and g, to be local.

The remaining rules for action and parallel composition are presented below.

$$\frac{\{p \wedge b\}\ \alpha\ \{q\}}{\{p\}\ b\ \rightarrow\ \alpha\ \{q\}}\ (action)$$

$$\frac{\{r\}\ f\ \{s\} \qquad \{r'\}\ g\ \{s'\}}{\{r \wedge r'\}\ f\ []\ g\ \{s \wedge s'\}}\ (parallel)$$

Similarly, all the assertions and annotations in these rules must be local. The parallel rule suggests the compositionality of parallel composition. In this rule, we first derive the specification for f and g separately. We can then take a parallel composition of the two components without the need to know the implementation of f nor g.

4.3.3 Example: Distributed Counter

Below is the distributed counter example with its local annotation, taken from [Mis17].

```
$f_j$ ::
  **initially** $old_j,\ new_j = 0,\ 0$
  $\{true\}$
  **loop**
    $\{true\}$
      $\alpha_j :: new_j := old_j + 1;$
    $\{new_j = old_j + 1\}$
      **if** [ $\beta_j :: \{new_j = old_j + 1\}\ ctr = old_j \to ctr := new_j\ \{true\}$
        | $\gamma_j :: \{new_j = old_j + 1\}\ ctr \neq old_j \to old_j := ctr\ \{true\}$
        ]
    $\{true\}$
  **forever**
```

Observe that all variables in each assertion are local to f_j. That is, the mentioned variables, old_j and new_j, are both local to f_j. Moreover, there is no mention of *ctr*, which is not a local variable.

4.3.4 Meta-Rules

The following rules, called *meta-rules*, are general rules about program specifications.

$$\frac{\{r\}\ f\ \{Q \mid s\} \qquad r' \Rightarrow r,\ s \Rightarrow s',\ Q' \subseteq Q,\ r' \text{ and } s' \text{ are local to } f}{\{r'\}\ f\ \{Q' \mid s'\}}\ (strengthen/weaken)$$

$$\frac{\{r\}\ f\ \{Q \mid s\} \qquad \{r'\}\ f\ \{Q' \mid s'\}}{\{r\ \wedge\ r'\}\ f\ \{Q\ \cup\ Q' \mid s\ \wedge\ s'\}}\ (conjunction)$$

$$\frac{\{r\}\ f\ \{Q \mid s\} \qquad \{r'\}\ f\ \{Q' \mid s'\}}{\{r\ \vee\ r'\}\ f\ \{Q\ \cap\ Q' \mid s\ \vee\ s'\}}\ (disjunction)$$

4.4 Safety Properties

Similar to UNITY, safety properties in the Bilateral Proof method are expressed using the **co** operator.

4.4.1 co

p **co** q in component f, where p and q do not need to be local to f, means that an effective execution of any action of f in a p-state establishes a q-state. Once p holds, it continues to hold until q is established. Additionally, **co** is compositional, i.e. p **co** q holds in f iff it holds in every subcomponent of f.

The following rule defines **co** for an annotated component f.

$$\frac{\{r\}\ f\ \{s\} \quad \forall\ b \rightarrow \alpha \text{ of } f.\ \{pre \wedge b \wedge p\}\ \alpha\ \{q\}}{\{r\}\ f\ \{p\ \mathbf{co}\ q \mid s\}}\ (co)$$

where *pre* is the annotation of the action $b \rightarrow \alpha$.

4.4.2 Special Cases of co

There are three special cases of **co**, namely **stable**, **constant**, and **invariant**. They are defined as follows.

$$\begin{aligned} \mathbf{stable}\ p &\equiv p\ \mathbf{co}\ p \\ \mathbf{constant}\ e &\equiv \forall c.\ \mathbf{stable}\ (c = e) \\ \mathbf{invariant}\ i &\equiv \mathbf{initially}\ i \text{ and } \mathbf{stable}\ i \end{aligned}$$

stable p states that once p holds, it will always continue to hold. **constant** e means that the value of e never changes. Finally, **invariant** i holds if i is always *true*. **initially** i in a component means the precondition of the component implies i.

4.4.3 Inheritance Rule

The inheritance meta-rule, or simply the inheritance rule, presented below defines the compositionality of safety properties.

$$\frac{\dfrac{\forall i.\ \{r_i\}\ f_i\ \{s_i\}}{\{r\}\ f\ \{s\}} \quad \forall i.\ \{r_i\}\ f_i\ \{\sigma \mid s_i\}}{\{r\}\ f\ \{\sigma \mid s\}}\ (inheritance)$$

If any safety property σ holds in all subcomponents f_i of f, then σ also holds in f. This holds in seq as well as parallel composition.

4.4.4 Invariance Rule

Recall that the parallel composition rule in Section 4.3.2 only allows annotations that are local. Therefore, each annotation is only allowed to mention variables that are local to f or g but not both. The invariance rule is used to overcome this problem.

The invariance rule states that,

1. A local invariant of a component, i.e. a local predicate that is invariant in the component, can be substituted for *true*, and vice versa, in any predicate in an annotation or property of the component.
2. Any invariant can be conjoined to an assertion including the postcondition.

The first statement suggests that we are allowed to add or remove a local invariant into any annotation or perpetual property. Here is a rule to add an invariant, which does not need to be a local invariant, into safety property **co**.

$$\frac{\{r\}\ f\ \{p\ \mathbf{co}\ q,\ \mathbf{invariant}\ i\ |\ s\}}{\{r\}\ f\ \{(p\ \wedge\ i)\ \mathbf{co}\ (q\ \wedge\ i)|\ s\}}\ (co\_invariance)$$

Recall that from Section 4.4.1, for any p **co** q in a component f, p and q do not need to be local to f. Therefore, the invariant, i, does not need to be local to f in this rule.

The second statement can be interpreted as follows.

$$\frac{\{r\}\ f\ \{\mathbf{invariant}\ i\ |\ s\}}{\{r\}\ f\ \{\mathbf{invariant}\ i\ |\ s\ \wedge\ i\}}\ (post\_invariance)$$

4.4.5 Example: Distributed Counter

We would like to prove the safety property

$$\sigma :: \forall m \in \mathbb{Z}.\ ctr = m\ \mathbf{co}\ (ctr = m \vee ctr = m + 1)$$

in the distributed counter example shown in Section 4.3.3.

The proof would be as follows.

1. Using the inheritance rule, it is sufficient to prove σ in every component f_j.
2. Using the **co** rule from Section 4.4.1, we must prove the assertion $\{pre \wedge b \wedge p\}\ \alpha\ \{q\}$ for α_j, β_j and γ_j. Here is the proof obligation for each action.

- α_j: $\{true \wedge true \wedge ctr = m\}\ new_j := old_j + 1\ \{ctr = m \vee ctr = m + 1\}$
- β_j: $\{new_j = old_j + 1 \wedge ctr = old_j \wedge ctr = m\}\ ctr := new_j\ \{ctr = m \vee ctr = m + 1\}$
- γ_j: $\{new_j = old_j + 1 \wedge ctr \neq old_j \wedge ctr = m\}\ old_j := ctr\ \{ctr = m \vee ctr = m + 1\}$

3. All the proof obligations above are immediate. Hence, we have proved the distributed counter program satisfies the safety property σ.

4.5 Progress Properties

Progress properties in the Bilateral Proof method use the same operators as UNITY. Unlike ensures and leads-to which have the same definition in UNITY, transient has a slightly different definition.

4.5.1 Transient

Recall that in UNITY, a predicate is transient if it is falsified by an execution of some single action. In Bilateral Proof, **transient** p means that if p holds at any point, $\neg p$ holds eventually. The following rule called *basis* rule defines transient for an annotated component f.

$$\frac{\{r\}\ f\ \{s\} \quad \forall\ b \to \alpha \text{ of } f.\ (pre \wedge p \Rightarrow b) \wedge \{pre \wedge p\}\ \alpha\ \{\neg p\}}{\{r\}\ f\ \{\mathbf{transient}\ (p \wedge \neg post_f) \mid s\}}\ (basis)$$

where *pre* is the annotation of the action $b \to \alpha$.

The rule guarantees that the guard of each action of f is enabled whenever p holds, and the execution establishes $\neg p$. $post_f$ is a local predicate of f that is initially *false* and becomes *true* only on the termination of f. If f never terminates, then $post_f$ is *false*.

Other Transient Rules

Besides the basis rule, there are also three other rules for transient, namely *sequencing*, *concurrency*, and *inheritance*.

$$\frac{\{r\}\ f\ \{\textbf{transient}\ (p \wedge \neg post_f) \mid post_f\} \qquad \{post_f\}\ g\ \{\textbf{transient}\ p \mid s\}}{\{r\}\ f; g\ \{\textbf{transient}\ p \mid s\}}\ (\textit{sequencing})$$

$$\frac{\{r\}\ f\ \{\textbf{transient}\ p \mid s\}}{\{r\}\ f\ [\!]\ g\ \{\textbf{transient}\ p \mid s\}}\ (\textit{concurrency})$$

$$\frac{\dfrac{\forall i.\ \{r_i\}\ f_i\ \{s_i\}}{\{r\}\ f\ \{s\}} \qquad \forall i.\ \{r_i\}\ f_i\ \{\textbf{transient}\ p \mid s_i\}}{\{r\}\ f\ \{\textbf{transient}\ p \mid s\}}\ (\textit{inheritance})$$

The sequencing rule establishes transient in sequential composition. Consider the case where f always terminates, in which $post_f$ will eventually hold. If p holds at the termination of f, then g will eventually establish $\neg p$ as **transient** p holds in g. Thus, **transient** p holds in $f; g$. On the other hand, if f does not always terminate, we require **transient** p to hold in both f and g.

The concurrency rule establishes transient in parallel composition. It is enough that the transient property only holds in one thread as long as the scheduler is fair. Consider the case where p holds at some point in the execution. Since fairness is enforced, f will always eventually get executed. As **transient** p holds in f, $\neg p$ will eventually hold.

Finally, transient is compositional as the inheritance rule suggests. This rule is very similar to the one for safety properties discussed in Section 4.4.3. If **transient** p holds in all subcomponents f_i of f, then **transient** p also holds in f. The rule applies for both in seq and parallel composition. However, there is actually no need to have an inheritance rule for parallel composition as the concurrency rule implies this rule.

4.5.2 Ensures

Presented below is the rule that defines **en**, which utilises both **co** and **transient**.

$$\frac{\{r\}\ f\ \{(p \wedge \neg q)\ \textbf{co}\ (p \vee q), \textbf{transient}\ (p \wedge \neg q) \mid s\}}{\{r\}\ f\ \{p\ \textbf{en}\ q \mid s\}}\ (\textit{en})$$

4.5.3 Leads-to

Similar to UNITY, leads-to or $\mapsto$ is defined using **en**. There are three rules that establish leads-to, which are *basis*, *transitivity*, and *disjunction*.

$$\frac{\{r\}\ f\ \{p\ \mathbf{en}\ q \mid s\}}{\{r\}\ f\ \{p\ \mapsto\ q \mid s\}}\ (basis)$$

$$\frac{\{r\}\ f\ \{p\ \mapsto\ q \mid s\} \quad \{r\}\ f\ \{q\ \mapsto\ r \mid s\}}{\{r\}\ f\ \{p\ \mapsto\ r \mid s\}}\ (transitivity)$$

$$\frac{\{r\}\ f\ \{(\forall p \in S.\ p\ \mapsto\ q) \mid s\}}{\{r\}\ f\ \{(\vee p \in S.\ p)\ \mapsto\ q \mid s\}}\ (disjunction)$$

The basis rule allows us to derive $p\ \mapsto\ q$ if p **en** q holds. As the name suggests, the transitivity rule indicates that leads-to is transitive. Lastly, the disjunction rule states that if every predicate p in a finite (or infinite) set S leads-to q, then the disjunction of S also leads-to q.

4.6 Summary

The safety property proof of the distributed counter example in Section 4.4.5 is fairly simple. While both Bilateral Proof and Owicki-Gries require annotations, Owicki-Gries requires an extra step of interference freedom testing. There is also no need to introduce extra conditions such as the rely and guarantee in the Rely/Guarantee method. The limitations of only having local variables can be overcome by using the invariance rule. Additionally, this method provides a way to write sequential programs easily unlike in UNITY, while still being able to prove both safety and progress properties.

Chapter 5

Isabelle/HOL

This chapter explains the syntax of Isabelle/HOL that is used commonly in this thesis. Additionally, a brief explanation of how proofs work in Isabelle is also presented. Isabelle [Pau94] is a generic theorem prover, whereas Isabelle/HOL [NPW02] is a specialisation of Isabelle for higher-order logic (HOL).

5.1 Types

The type system in HOL resembles functional programming languages. There are predefined types such as `bool`, `nat`, `list`, etc. We can create an alias of the existing types using the keyword **type_synonym**.

5.2 Datatype

In Isabelle, the keyword **datatype** is used to define new data types. A datatype is defined by a list of constructors with argument types, separated by |. The general form is

$$\textbf{datatype}\ (\alpha_1, \ldots, \alpha_n)\, t \ = \ C_1\ \tau_{1,1}\ \ldots\ \tau_{1,k_1}\ \mid\ \ldots\ \mid\ C_m\ \tau_{m,1}\ \ldots\ \tau_{m,k_m}$$

where α_i are distinct type variables (the parameters), C_i are distinct constructor names and $\tau_{i,j}$ are the argument types.

For example, one can define the type of natural numbers as
datatype `nat` = `0 | Suc nat`

It uses two constructors, `0` and `Suc`. The values of type `nat` are `0`, `Suc 0`, `Suc (Suc 0)`, etc.

5.3 Functions

Non-recursive functions can be defined using **definition** or **abbreviation**. The only difference is, when used, definitions need to be expanded explicitly, while abbreviations are expanded automatically.

```
definition mult :: "nat ⇒ nat ⇒ nat" where "mult x y ≡ x * y"
abbreviation mult' :: "nat ⇒ nat ⇒ nat" where "mult' x y ≡ x * y"
```

Recursive functions are usually defined using **primrec** or **fun**. **primrec** or primitive recursion indicates that each recursive call peels off a datatype constructor from one of the arguments. This allows Isabelle to automatically prove termination. **fun** is more general than **primrec**. It allows arbitrary pattern matching in all parameters and termination can usually be proved automatically by Isabelle.

```
primrec add :: "nat ⇒ nat ⇒ nat" where
  "add 0 y = y"
| "add (Suc x) y = Suc 0 + add x y"

fun ack2 :: "nat ⇒ nat ⇒ nat" where
  "ack2 n 0 = Suc n"
| "ack2 0 (Suc m) = ack2 (Suc 0) m"
| "ack2 (Suc n) (Suc m) = ack2 (ack2 n (Suc m)) m"
```

5.4 Inductive Definitions

The keyword **inductive** defines the least predicate that is closed under a set of axioms and inference rules. It is analogous to an inductively defined set except that it is a predicate.

For example, we define an inductive predicate below that describes even numbers.

```
inductive even_pred :: "nat ⇒ bool" where
  zero: "even_pred 0"
| step: "even_pred n ⟹ even_pred (Suc (Suc n))"
```

Isabelle generates an induction principle referred to as *rule induction.* This allows us to prove a property for every value that satisfies the inductive predicate. It works similarly to mathematical induction, where we first prove the base cases, then the inductive steps which are formed by our base set of rules. Most of the proofs in this thesis use rule induction.

5.5 Theorems and Proofs

5.5.1 Theorems

A theorem in Isabelle, in general, has the following form.

$\llbracket$ $\texttt{A}_1$; ...; $\texttt{A}_n$ $\rrbracket \Longrightarrow$ B

The implication $\Longrightarrow$ is used to separate the premises and conclusion of the theorem. The brackets above are simply expanded to $\texttt{A}_1 \Longrightarrow \ldots \Longrightarrow \texttt{A}_n \Longrightarrow \texttt{B}$.

5.5.2 Proofs

The general schema of the proofs is

lemma `name: <goal>`
apply `(<method>)`
apply `(<method>)`
...
done

The keyword **lemma** (or **theorem**) establishes a new theorem to be proved as well as giving a name to the theorem.

In Isabelle, the proof state consists of a list of subgoals to prove. When we first declare a lemma, we have one subgoal to prove, namely the goal itself.

The keyword **apply** is used to apply a particular proof strategy on the current subgoal. The current subgoal is defined to be the subgoal that is first in the list. Every **apply** operation will do one of the following behaviours depending on the strategies used.

- proves a subgoal, in which the subgoal is then removed from the list of subgoals
- splits a subgoal into multiple subgoals

- transforms a subgoal to a simpler version of it, making it easier to prove

The following proof strategies are those used most often in this thesis.

- `rule`, `erule`, `frule`, `drule` apply theorems with forward or backward reasoning.
- `simp` uses term rewriting in order to simplify the terms in the subgoals.
- `induction` performs an induction, suitable for any inductive definitions. We can specify which rule to use, e.g. `apply (induction rule: nat.induct)`.
- `case` or `case_tac` splits the current subgoal into different cases.
- `subgoal_tac` inserts a formula as an additional premise.
- Automated methods such as `clarsimp`, `auto`, `blast`, and `fastforce` attempt to automatically prove a subgoal.

The keyword `done` indicates that all subgoals have been proved and hence the lemma is proved.

Chapter 6

Formalisation and Soundness Proof of Bilateral Proof in Isabelle/HOL

This chapter presents the formalisation and soundness proof of Bilateral Proof in Isabelle/HOL. The formalisation uses a similar approach from previous work on Owicki-Gries and Rely/Guarantee formalisation in Isabelle/HOL [PN02].

The formalisation is broken down into five main steps. First, the programming language is formalised. Afterwards, we define what it means for the program specification of the form $\{r\}\ f\ \{Q \mid s\}$ and the perpetual properties to be valid. The concept of local annotations is then formalised. The next step is formalising the proof rules. These include the local annotation rules, meta-rules and rules in perpetual properties. The final step is to carry out the soundness proofs of these rules.

The complete list of the Isabelle theory files can be found here.

6.1 Formalisation of the Language

We want to formalise the programming language shown in Section 4.2.1. There are two approaches to formalise a language in Isabelle, *deep embeddings* and *shallow embeddings*. In contrast to the former where language syntax/terms and semantics are represented separately, the latter expresses a term directly as its semantics.

Although a shallow embedding may simplify reasoning since it deals with the semantics directly without worrying about the syntax, a deep embedding is more favourable for Bilateral Proof. The syntactic layer in deep embeddings allows us to retrieve all actions of a particular component. This is particularly useful because the rules to define safety and progress use the actions of each component.

Additionally, the semantics of the programs will be expressed using small-step operational semantics. This semantics captures step by step executions, and therefore allows us to observe the interleaving of concurrent programs.

6.1.1 State Space

We choose `state` to be a function from variable names (string) to values (natural number). Boolean expressions, annotations, and invariants are represented as predicates over states.

```
type_synonym vname = string
type_synonym state = "vname ⇒ nat"

type_synonym bexp = "state ⇒ bool"
type_synonym ann = "state ⇒ bool"
type_synonym inv = "state ⇒ bool"
```

6.1.2 Syntax

We define the syntax of the language with **datatype** `com`.

```
type_synonym state_rel = "(state × state) set"

datatype com =
    DONE
  | ABORTED
  | Action ann state_rel           ("⦃_⦄ ACTION _")
  | Semi com com                   ("_;;_")
  | If ann bexp com com            ("⦃_⦄ IF _ THEN _ ELSE _")
  | While ann ann bexp inv com     ("⦃_⦄ ⦃_⦄ WHILE _ _ DO _")
  | Parallel "com list" "ann list" ("PARALLEL _ _")
  | Post_ann ann                   ("⦃_⦄ POSTANN")
```

We can think of the text inside the brackets on RHS as an alias to each construct. Each component (excluding `DONE` and `ABORTED`) has its local annotation embedded in the syntax. They are represented inside the brackets ⦃_⦄.

We discuss each component further below.

- `DONE` indicates the program has terminated.
- `ABORTED` indicates the program does not satisfy some of its local annotations.
- `Action` is represented as a relation over states. As a consequence, the actions in this system can have non-determinism. The actions are done this way to allow the if statement with multiple guards shown in the distributed counter example (Section 4.3.3).
- `Semi` is the sequential composition (or semicolon) of two components. It does not have its own local annotation but it gets the annotation from the first component.
- `If` is the usual if/else statement.
- `While` is the standard while loop with invariant. A while loop has two annotations at the front. The first annotation is the local annotation. The second annotation represents the local part of the while loop's conditional, which is there solely to help prove progress properties.
- `Parallel` is the parallel composition which takes in a list of `com` rather than limiting it to two components. It also takes in a list of postconditions (`ann list`) to simplify the soundness proof of parallel composition.
- `Post_ann` is used to help prove the soundness of parallel composition. Every time a component running in parallel terminates, it will be substituted with a `Post_ann`, with the annotation being the same as the terminated component's postcondition. As `Post_ann` is intended only to help prove the soundness of parallel composition, no program should be written using a `Post_ann`.

6.1.3 Small-Step Semantics

Small-step semantics defines the execution of a program to be consecutive transitions between *configurations*. A configuration is a program fragment and state pair, which in Isabelle has the type `com` $\times$ `state`.

These transitions are of the form $(c, s) \to (c', s')$. Executing a component c in a state s produces a component c' with a new state s'. In terms of terminology, we say that (c, s) is reduced to (c', s').

In our formalisation, we assume that the executions of programs are infinite. That is, for any (c, s), it is always possible to be reduced to some (c', s').

We want to make sure that any component that does not satisfy its local annotation is reduced to `ABORTED`. Here, we define a function `com_pre` to extract a local annotation/precondition of a component.

```
fun com_pre :: "com ⇒ ann" where
  "com_pre DONE = true"
| "com_pre ABORTED = false"
| "com_pre (⦃pre⦄ ACTION _) = pre"
| "com_pre (c₁;;_) = com_pre c₁"
| "com_pre (⦃pre⦄ IF _ THEN _ ELSE _) = pre"
| "com_pre (⦃pre⦄ ⦃local_b⦄ WHILE _ _ DO _) = pre"
| "com_pre (PARALLEL Ps Ts) = And (map com_pre Ps)"
| "com_pre (⦃pre⦄ POSTANN) = pre"
```

Though `DONE` and `ABORTED` do not have their own local annotation, we assign `true` and `false` as their local annotation respectively. We choose to do so based on the idea that any state should satisfy the local annotation of `DONE`, and the opposite for `ABORTED`. Another alternative is to use an `option` type, where the annotation of `DONE` and `ABORTED` is `None`. However, this will make proofs tedious.

The local annotation of a parallel component is simply the conjunction of the local annotation of each subcomponent.

Semantics

We define the small-step semantics inductively using a set of axioms and rules about transitions between configurations.

```
inductive small_step :: "com × state ⇒ com × state ⇒ bool" (infix "→" 55) where
  Abort:   "¬ com_pre c s ⟹ (c, s) → (ABORTED, s)"

| DoneR:   "(DONE, s) → (DONE, s)"

| Action:  "⟦ (s, s') ∈ state_rel; pre s ⟧ ⟹
           (⦃pre⦄ ACTION state_rel, s) → (DONE, s')"
| ActionR: "⟦ ∀s'. (s, s') ∉ state_rel; pre s ⟧ ⟹
           (⦃pre⦄ ACTION state_rel, s) → (⦃pre⦄ ACTION state_rel, s)"

| Semi1:   "⟦ (c₁, s) → (DONE, s') ⟧ ⟹ (c₁;;c₂, s) → (c₂, s')"
| Semi2:   "⟦ (c₁, s) → (c₁', s'); com_pre c₁ s; c₁' ≠ DONE ⟧ ⟹
           (c₁;;c₂, s) → (c₁';;c₂, s')"

| IfT:     "⟦ b s; pre s ⟧ ⟹ (⦃pre⦄ IF b THEN c₁ ELSE c₂, s) → (c₁, s)"
| IfF:     "⟦ ¬ b s; pre s ⟧ ⟹ (⦃pre⦄ IF b THEN c₁ ELSE c₂, s) → (c₂, s)"

| WhileT:  "⟦ b s; pre s ⟧ ⟹
           (⦃pre⦄ ⦃local_b⦄ WHILE b i DO c, s) →
           (c;;⦃i⦄ ⦃local_b⦄ WHILE b i DO c, s)"
| WhileF:  "⟦ ¬ b s; pre s ⟧ ⟹
           (⦃pre⦄ ⦃local_b⦄ WHILE b i DO c, s) →
           (⦃i and not local_b⦄ ACTION {(s, s'). s = s'}, s)"

| ParA:    "⟦ i < length Ps; ∀j<length Ps. j ≠ i ⟶ Ps!j = ⦃Ts!j⦄ POSTANN;
           Ps!i = c; (c, s) → (DONE, s'); com_pre (PARALLEL Ps Ts) s ⟧ ⟹
           (PARALLEL Ps Ts, s) → (DONE, s')"

| ParD:    "⟦ i < length Ps;  ∃j<length Ps. j ≠ i ∧ Ps!j ≠ ⦃Ts!j⦄ POSTANN;
           Ps!i = c; (c, s) → (DONE, s'); Ps' = Ps[i:=(⦃Ts!i⦄ POSTANN)];
           com_pre (PARALLEL Ps Ts) s ⟧ ⟹
           (PARALLEL Ps Ts, s) → (PARALLEL Ps' Ts, s')"

| Par:     "⟦ i < length Ps; Ps!i = c; (c, s) → (c', s'); c' ≠ DONE;
           Ps' = Ps[i:=c']; com_pre (PARALLEL Ps Ts) s ⟧ ⟹
           (PARALLEL Ps Ts, s) → (PARALLEL Ps' Ts, s')"
```

We will go through each of the rules in more detail.

- `Abort` ensures a component is reduced to `ABORTED` when its local annotation is false. Note that `(ABORTED, s) → (ABORTED, s)` satisfies this rule.
- `DoneR` is there to achieve infinite executions.
- `Action` and `ActionR` define the different reduction for actions depending on the value of the guard. If the guard is true, it will be reduced to `DONE` and the state is changed accordingly from `state_rel`. Otherwise, the action is blocking and nothing is changed.
- `Semi1` and `Semi2` state that in a sequential composition, if the first component terminates, only the second component remains to be executed. Otherwise, we can substitute the first component with its reduction.
- `IfT` and `IfF` state that we reduce an if statement to the component in the `IF`

branch if the condition `b` holds in the state `s`, or with the component in the `ELSE` branch otherwise. In either case, the state is left unchanged.

- `WhileT` and `WhileF` state that if the condition `b` holds in the state `s`, it is reduced to the body followed by the original while loop. Otherwise, it is reduced to an action that does nothing, called *skip*. This will then be reduced to `DONE` in the next step.

  One might think that there is no point in adding a skip as the execution goes to `DONE` regardless. However, this is not true in Bilateral Proof. The skip actually plays a big role in a progress property rule, which will be discussed later.

- `ParA`, `ParD`, and `Par` are the rules for parallel composition. Each rule states how the parallel composition makes a step when the i^{th} component makes a step.

  `ParA` states that the parallel composition is reduced to `DONE` if the i^{th} component is reduced to `DONE` and all the other components are some `POSTANN`, i.e. has terminated.

  In `ParD`, the i^{th} component is also reduced to `DONE`, but there exist some components which have not terminated. In this case, we substitute the i^{th} component with a `POSTANN`.

  `Par` explains the case when the i^{th} component takes a step and the resulting component is not `DONE`. The parallel composition step substitutes the i^{th} component with the new component.

  In each of these rules, the state of the parallel composition changes the same way as the state change in the execution of the i^{th} component.

We also define a reflexive transitive closure of the small-step semantics to represent the execution of a program, written as $\rightarrow*$.

```
abbreviation
  small_steps_star :: "com × state ⇒ com × state ⇒ bool" (infix "→*" 55)
where
  "x →* y ≡ star small_step x y"
```

6.2 Validity of Program Specifications

We need to define what it means for a program specification of the form $\{r\}\ f\ \{Q \mid s\}$ from Section 4.3.1 to be correct or *valid*. It is usually written as $\models \{r\}\ f\ \{Q \mid s\}$.

Before being able to do so, we need to formalise the meaning of each of the perpetual properties discussed in Section 4.4 and 4.5.

6.2.1 Perpetual Properties

Similar to the language formalisation, we separate the syntax and the semantics of perpetual properties. Below is a **datatype** called `perpetual` that represents the syntax.

```
datatype perpetual =
    Co ann ann          ("_ CO _")
  | Stable ann          ("STABLE _")
  | Constant vname      ("CONSTANT _")
  | Invariant ann       ("INVARIANT _")
  | Transient ann       ("TRANSIENT _")
  | Ensure ann ann      ("_ EN _")
  | Leads_to ann ann    ("_ ↦ _")
```

Recall that perpetual properties are properties that hold throughout an execution of a program. We define a helper predicate called `reachable_sat`, where `reachable_sat P f s` means that every reachable configuration, i.e. program and state pair, starting from `(f, s)` satisfies `P`.

```
definition reachable_sat :: "(com ⇒ state ⇒ bool) ⇒ com ⇒ state ⇒ bool" where
  "reachable_sat P f s ≡ ∀f' s'. (f, s) →* (f', s') ⟶ P f' s'"
```

Safety Properties

In our formalisation, **co** means that for any reachable program that starts from the initial program and state pair, if the program makes a step and `p` holds before, then `q` holds after the step.

```
definition co :: "ann ⇒ ann ⇒ com ⇒ state ⇒ bool" where
  "co p q f s ≡
    reachable_sat
      (λf' s'. ∀f'' s''. p s' ∧ (f', s') → (f'', s'') ⟶ q s'') f s"
```

The special cases of **co**, namely **stable**, **constant**, and **invariant**, are defined using the **co** definition.

```
definition stable :: "ann ⇒ com ⇒ state ⇒ bool" where
  "stable p ≡ co p p"
```

```
definition "constant" :: "vname ⇒ com ⇒ state ⇒ bool" where
  "constant e f s ≡ ∀c. (stable (λs. s e = s c)) f s"

definition invariant :: "ann ⇒ com ⇒ state ⇒ bool" where
  "invariant p f s ≡ p s ∧ stable p f s"
```

Progress Properties

All progress properties are concerned with *eventuality.* Eventuality requires a property to be satisfied in whichever path the program takes. Therefore, we need to define all possible paths within a program in order to define progress properties.

Here, paths are functions from indices (natural number) to configurations. The i^{th} index represents the configuration produced after taking i steps from the initial configuration.

We define `is_path P f s` to be true iff `P` is a valid path that starts from `(f, s)`. These valid paths are collected to a set in the `paths` definition.

```
definition is_path :: "(nat ⇒ com × state) ⇒ com ⇒ state ⇒ bool" where
  "is_path P f s ≡ P 0 = (f, s) ∧ (∀n. P n → P (n + 1))"

definition paths :: "com ⇒ state ⇒ (nat ⇒ com × state) set" where
  "paths f s = {P. is_path P f s}"
```

We have discussed the meaning of **transient** p from the paper, which is if p holds at some point, then eventually $\neg p$ holds. We formalise transient slightly differently to directly include the $post_f$ mentioned in the basis rule (Section 4.5.1).

In our formalisation, transient means that in every possible path, if at some index i, p holds and the program has not terminated, then at some index j bigger or equal to i, $\neg p$ holds or the program terminates. A program is terminated if it is a `DONE` or `ABORTED`.

```
definition transient :: "ann ⇒ com ⇒ state ⇒ bool" where
  "transient p f s ≡
    (∀path∈paths f s.
      ∀i f s. path i = (f, s) ∧ p s ∧ ¬ has_terminated f ⟶
        (∃j f' s'. j ≥ i ∧ path j = (f', s') ∧ ((not p) s' ∨ has_terminated f')))"
```

Ensures and leads-to are defined in a similar way as transient.

```
definition ensures :: "ann ⇒ ann ⇒ com ⇒ state ⇒ bool" where
```

```
  "ensures p q f s ≡
    (∀path∈paths f s.
      ∀i f s. path i = (f, s) ∧ p s ⟶
        ((not q) s ⟶ (∃f' s'. path (Suc i) = (f', s') ∧ (p or q) s')) ∧
        (∃j f' s'. j ≥ i ∧ path j = (f', s') ∧ (p s' ∧ has_terminated f' ∨ q s')))"
```

```
definition leads_to :: "ann ⇒ ann ⇒ com ⇒ state ⇒ bool" where
  "leads_to p q f s ≡
    (∀path∈paths f s.
      ∀i f s. path i = (f, s) ∧ p s ∧ ¬ has_terminated f ⟶
        (∃j f' s'. j ≥ i ∧ path j = (f', s') ∧ (q s' ∨ has_terminated f')))"
```

Notice that all `transient` and `leads_to` properties trivially hold in programs that terminate. However, for `ensures`, it requires `p` to hold upon termination if it does not establish `q`.

Syntax to Semantics

We can now define a function that translates the syntactic perpetual property to its semantics.

```
primrec eval :: "perpetual ⇒ com ⇒ state ⇒ bool" where
  "eval (p CO q) = co p q"
| "eval (STABLE p) = stable p"
| "eval (CONSTANT e) = constant e"
| "eval (INVARIANT i) = invariant i"
| "eval (TRANSIENT p) = transient p"
| "eval (p EN q) = ensures p q"
| "eval (p ↦ q) = leads_to p q"
```

6.2.2 Valid Specification

The valid specification definition is presented below.

```
definition
  valid_spec :: "ann ⇒ com ⇒ perpetual set ⇒ ann ⇒ bool" ("⊨ {_} _ {_ | _}")
where
  "⊨ {r} f {Q | t} ≡
    (∀s. r s ⟶
      reachable_sat (λf' s'. has_terminated f' ⟶ holds t f' s') f s ∧
      (∀op ∈ Q. eval op f s))"
```

Here, `holds q f s` $\equiv$ `q s` $\wedge$ `(not (=) f) ABORTED`.

`valid_spec` means that if we start in a state `s` which satisfies the precondition `r`,

1. For any reachable program from the initial configuration `(f, s)`, if it terminates, then the postcondition `t` holds and the program is not `ABORTED`. In other words, the program never violates any of its local annotations during its execution.
2. Every perpetual property in `Q` holds for any execution of `f` starting in an `s`-state.

6.3 Formalisation of Local Annotations

Before formalising the proof rules, we need to formalise the concept of local annotations, discussed in Section 4.3.2. More specifically, we want to be able to check whether a program is annotated correctly, i.e. every annotation is local.

Any arbitrary annotation in a program without parallel composition is automatically local. As there is only one thread running at a time, it must have exclusive write-access to any variable. Thus, it is enough to check only the annotations for parallel composition.

There are a few approaches to do this.

1. In each thread, retrieve all variables that are mentioned in the annotations and all variables that are being written to. The annotations of a thread are local if there is no other thread that writes to a variable in the annotations. This approach requires the formalisation of annotations and actions to be syntactic.
2. Instead of retrieving the variables as the first approach suggests, we ask the users who write the program to declare the variables mentioned in the annotations and variables written in the actions for each thread.
3. Use an approach inspired by Owicki-Gries' interference freedom proof. For each annotation, we check if it remains valid upon execution of any action in another thread, i.e. $\models \{ann\}\ action\ \{ann\}$. This is similar to **stable** in safety properties.

The first approach is the one resembling the definition of locality, i.e. having exclusive write-access by observing the variables. However, the current formalisation of the programming language does not have the syntactic layer. The second approach does not require the syntactic layer, but it relies on the user's input, which is not ideal. Therefore, we choose to use the last approach. This approach has a weaker definition of locality but works well with the current formalisation.

One might argue that the complexity of the method we choose will be quadratic similar to Owicki-Gries. While this is true, if we can prove that Bilateral Proof works with this weaker definition of locality, then the definition of locality from the paper also works. It is just a matter of implementing the syntactic layer.

The function below, `all_anns`, collects all annotations of a component.

```
fun all_anns :: "com ⇒ ann set" where
  "all_anns DONE = {true}"
| "all_anns ABORTED = {false}"
| "all_anns (⦃pre⦄ ACTION _) = {pre}"
| "all_anns (c1;;c2) = all_anns c1 ∪ all_anns c2"
| "all_anns (⦃pre⦄ IF b THEN c1 ELSE c2) = {pre} ∪ all_anns c1 ∪ all_anns c2"
| "all_anns (⦃pre⦄ ⦃local_b⦄ WHILE b i DO c) =
  {pre, i, i and not local_b} ∪ all_anns c"
| "all_anns (PARALLEL Ps Ts) = all_anns_par (map all_anns Ps) Ts"
| "all_anns (⦃pre⦄ POSTANN) = {pre}"
```

Notice that the annotations of `While` include `i` and `(i and not local_b)`. These are the annotations we get after we execute a while loop and hence included here. The local annotations of a parallel composition are defined to be the conjunction between any annotation or postcondition of each thread.

The next step is to retrieve all actions of a particular component. This is done by the function `actions_of` below.

```
fun actions_of :: "com ⇒ com set" where
  "actions_of (⦃pre⦄ ACTION c) = {⦃pre⦄ ACTION c}"
| "actions_of (c1;;c2) = actions_of c1 ∪ actions_of c2"
| "actions_of (⦃_⦄ IF _ THEN c1 ELSE c2) = actions_of c1 ∪ actions_of c2"
| "actions_of (⦃_⦄ ⦃local_b⦄ WHILE b i DO c) =
  actions_of c ∪ {⦃i and not local_b⦄ ACTION {(s, s'). s = s'}}"
| "actions_of (PARALLEL Ps Ts) = (⋃c ∈ set Ps . actions_of c)"
| "actions_of _ = {}"
```

Recall that in the `WhileF` small-step semantics rule, the while loops gets reduced to skip. Therefore, while loops have an extra action (skip) in order to match the semantics.

The function `action_state_rel` extracts the local annotation and the state relation from an action.

```
fun action_state_rel :: "com ⇒ (ann × state_rel)" where
  "action_state_rel (⦃pre⦄ ACTION c) = (pre, c)"
| "action_state_rel _ = (true, {})"
```

Now we define `is_ann_stable` to be a predicate that asserts if an annotation or a postcondition is valid upon the execution of actions in a component, i.e. stable.

```
definition is_ann_stable :: "ann ⇒ com ⇒ bool" where
  "is_ann_stable p f ≡
    (∀a pre state_rel. a ∈ actions_of f ⟶
      (pre, state_rel) = action_state_rel a ⟶
        (∀s s'. (s, s') ∈ state_rel ⟶ p s ⟶ p s'))"
```

Building on top of `is_ann_stable`, we can define if the annotations of a component are stable in another component.

```
definition is_com_stable :: "com ⇒ com ⇒ bool" where
  "is_com_stable f g ≡ ∀p. p ∈ all_anns f ⟶ is_ann_stable p g"
```

Finally, `valid_ann` takes in a list of components running in parallel and a list of their postconditions. We check if the annotations and the postcondition of each component are stable in the other components.

```
definition valid_ann' :: "com ⇒ ann ⇒ com ⇒ bool" where
  "valid_ann' f t g ≡ is_ann_stable t g ∧ is_com_stable f g"

definition valid_ann :: "com list ⇒ ann list ⇒ bool" where
  "valid_ann Ps Ts =
    (∀i. i < length Ps ⟶
      (∀j. j < length Ps ⟶ i ≠ j ⟶ valid_ann' (Ps!i) (Ts!i) (Ps!j)))"
```

6.4 Formalisation of Proof Rules

We have decided to formalise a representative subset of the proof rules. These proof rules are stratified into two layers. The first layer concerns rules that do not involve perpetual properties. The second layer includes rules that mention perpetual properties as well as the rules from the first layer.

This is done because of two reasons.

- The Bilateral Proof paper is structured such that local annotations come before introducing any perpetual property. It seems to be the case that whenever we want to derive a specification of a program, we apply the rules in two steps. The first step is to apply the local annotation rules. After that, we can apply the perpetual properties rules.
- Having the local annotation rule for parallel composition and the invariance rule in the same layer causes the soundness proof of parallel composition to be especially difficult, and would make the corresponding proof sketch in Misra's paper [Mis17] unsound or at least highly incomplete.

6.4.1 First Layer: `biloof_no_perpetual`

The first layer contains the local annotation rules and the meta-rules specific to only pre and postcondition. It is inductively defined using the `biloof_no_perpetual` predicate below. In Isabelle, we write the specifications as `⊢ {r} f {t}`.

```
inductive biloof_no_perpetual:: "ann ⇒ com ⇒ ann ⇒ bool" ("⊢ {_} _ {_}") where
  b_action[intro]:
  "⟦ ∀s s'. (s, s') ∈ state_rel ∧ pre s ⟶ t s' ⟧ ⟹
  ⊢ {pre} ⦃pre⦄ ACTION state_rel {t}"

| b_semi[intro]:
  "⟦ ⊢ {r} c₁ {com_pre c₂}; ⊢ {com_pre c₂} c₂ {t} ⟧ ⟹ ⊢ {r} c₁;;c₂ {t}"

| b_if[intro]:
  "⟦ ∀s. (pre and b) s ⟶ com_pre c₁ s; ∀s. (pre and not b) s ⟶ com_pre c₂ s;
  ⊢ {com_pre c₁} c₁ {t}; ⊢ {com_pre c₂} c₂ {t} ⟧ ⟹
  ⊢ {pre} ⦃pre⦄ IF b THEN c₁ ELSE c₂ {t}"

| b_while[intro]:
  "⟦ ∀s. pre s ⟶ i s; ∀s. (i and b) s ⟶ com_pre c s; ⊢ {com_pre c} c {i};
  ∀s. (not b) s ⟶ (not local_b) s; ∀s. (i and not local_b) s ⟶ t s ⟧ ⟹
  ⊢ {pre} ⦃pre⦄ ⦃local_b⦄ WHILE b i DO c {t}"

| b_par:
  "⟦ valid_ann Ps Ts; length Ps = length Ts; length Ps > 0;
  ∀i<length Ps. ⊢ {com_pre (Ps!i)} Ps!i {Ts!i} ∨ Ps!i = ⦃Ts!i⦄ POSTANN;
  ∃i<length Ps. Ps!i ≠ ⦃Ts!i⦄ POSTANN ⟧ ⟹
  ⊢ {com_pre (PARALLEL Ps Ts)} PARALLEL Ps Ts {And Ts}"
```

```
| b_strengthen_weaken:
  "⟦ ⊢ {r} f {t}; ∀s. r' s ⟶ r s; ∀s. t s ⟶ t' s  ⟧ ⟹ ⊢ {r'} f {t'}"

| b_conjunction:
  "⟦ ⊢ {r} f {t}; ⊢ {r'} f {t'} ⟧ ⟹ ⊢ {r and r'} f {t and t'}"

| b_disjunction:
  "⟦ ⊢ {r} f {t}; ⊢ {r'} f {t'} ⟧ ⟹ ⊢ {r or r'} f {t or t'}"
```

`b_action`, `b_semi`, `b_if`, `b_while`, and `b_par` are the rules to construct local annotations. The remaining rules, `b_strengthen_weaken`, `b_conjunction`, and `b_disjunction`, are meta-rules. We explore `b_par` in more detail below.

Local Annotation Rule for Parallel Composition

The local annotation rule for parallel composition from Section 4.3.2 is defined using the rule `b_par`.

```
⟦valid_ann Ps Ts; length Ps = length Ts; 0 < length Ps;
 ∀i<length Ps.
   ⊢ {com_pre (Ps ! i)} Ps ! i {Ts ! i} ∨ Ps ! i = ⦃Ts ! i⦄ POSTANN;
 ∃i<length Ps. (not (=) (Ps ! i)) (⦃Ts ! i⦄ POSTANN)⟧
⟹ ⊢ {com_pre (PARALLEL Ps Ts)} PARALLEL Ps Ts {And Ts}
```

The original rule assumes the derivability of each component's specification and the locality of each component's annotations. Here, we modify the premises in order to help prove soundness. We explain some of the premises in more details below.

- `valid_ann Ps Ts` formalises the locality of the annotations and postconditions.
- `0 < length Ps` requires the number of components running in parallel to be anything except 0. In other words, we do not want the components to be an empty list.
- `∀i<length Ps. ⊢ {com_pre (Ps ! i)} Ps ! i {Ts ! i} ∨ Ps ! i = ⦃Ts ! i⦄ POSTANN` states that for each component, either we can derive a specification or it has terminated. This is there to help prove the soundness of this rule.
- `∃i<length Ps. (not (=) (Ps ! i)) (⦃Ts ! i⦄ POSTANN)` does not allow every component to be a `POSTANN`. If all the components have terminated, the program should be a `DONE` instead of `PARALLEL`.

6.4.2 Second Layer: `biloof`

As briefly mentioned, every specification derived in the first layer is included in this second layer. On top of that, the meta-rules that contain perpetual properties, safety properties rules, and progress properties rules are included in this layer. We inductively define the rules using the `biloof` predicate below. In Isabelle, we write the specifications as `⊩ {r} f {Q | t}`.

```
inductive biloof:: "ann ⇒ com ⇒ perpetual set ⇒ ann ⇒ bool" ("⊩ {_} _ {_ | _}")
where
  bl_biloof_no_perpetual:
  "⊢ {r} f {t} ⟹ ⊩ {r} f {{} | t}"

| bl_strengthen_weaken:
  "⟦ ⊩ {r} f {Q | t}; ∀s. r' s ⟶ r s; Q' ⊆ Q; ∀s. t s ⟶ t' s ⟧ ⟹
  ⊩ {r'} f {Q' | t'}"

| bl_conjunction:
  "⟦ ⊩ {r} f {Q | t}; ⊩ {r'} f {Q' | t'} ⟧ ⟹ ⊩ {r and r'} f {Q ∪ Q' | t and t'}"

| bl_disjunction:
  "⟦ ⊩ {r} f {Q | t}; ⊩ {r'} f {Q' | t'} ⟧ ⟹ ⊩ {r or r'} f {Q ∩ Q' | t or t'}"

| b_co:
  "⟦ ⊩ {r} f {Q | t}; ∀s. p s ⟶ q s;
  ∀a pre state_rel. a ∈ actions_of f ⟶ (pre, state_rel) = action_state_rel a
  ⟶ (∀s s'. (s, s') ∈ state_rel ⟶ (pre and p) s ⟶ q s') ⟧ ⟹
  ⊩ {r} f {Q ∪ {p CO q} | t}"

| b_co_inheritance_semi:
  "⟦ ⊢ {r} c₁ {com_pre c₂}; ⊢ {com_pre c₂} c₂ {t}; ⊩ {r} c₁ {Q | com_pre c₂};
  (p CO q) ∈ Q; ⊩ {com_pre c₂} c₂ {Q'| t}; (p CO q) ∈ Q' ⟧ ⟹
  ⊩ {r} c₁;;c₂ {{p CO q} | t}"

| b_co_inheritance_if:
  "⟦ ∀s. (pre and b) s ⟶ com_pre c₁ s; ∀s. (pre and not b) s ⟶ com_pre c₂ s;
  ⊢ {com_pre c₁} c₁ {t}; ⊢ {com_pre c₂} c₂ {t}; ⊩ {com_pre c₁} c₁ {Q | t};
  (p CO q) ∈ Q; ⊩ {com_pre c₂} c₂ {Q'| t}; (p CO q) ∈ Q' ⟧ ⟹
  ⊩ {pre} ⦃pre⦄ IF b THEN c₁ ELSE c₂ {{p CO q} | t}"

| b_co_inheritance_while:
  "⟦ ∀s. pre s ⟶ i s; ∀s. (i and b) s ⟶ com_pre c s;
  ⊢ {com_pre c} c {i}; ⊩ {com_pre c} c {Q | t}; (p CO q) ∈ Q;
  ∀s. (not b) s ⟶ (not local_b) s; ∀s. (i and not local_b) s ⟶ t s ⟧ ⟹
  ⊩ {pre} ⦃pre⦄ ⦃local_b⦄ WHILE b i DO c {{p CO q} | t}"

| b_co_inheritance_parallel:
  "⟦ valid_ann Ps Ts; length Ps = length Ts; length Ts = length Qs;
  length Ps > 0; ∀i<length Ps. ⊢ {com_pre (Ps!i)} Ps!i {Ts!i};
  ∀i<length Ps. ⊩ {com_pre (Ps!i)} Ps!i {Qs!i | Ts!i};
  ∀i<length Qs. (p CO q) ∈ (Qs!i) ⟧ ⟹
  ⊩ {com_pre (PARALLEL Ps Ts)} PARALLEL Ps Ts {{p CO q} | And Ts}"

| b_invariant:
  "⟦ ⊩ {r} f {Q | t}; ∀s. r s ⟶ i s;
  ∀a pre state_rel. a ∈ actions_of f ⟶ (pre, state_rel) = action_state_rel a
  ⟶ (∀s s'. (s, s') ∈ state_rel ⟶ (pre and i) s ⟶ i s') ⟧ ⟹
  ⊩ {r} f {Q ∪ {INVARIANT i} | t}"

| b_invariant_pre_post:
  "⟦ ⊩ {r} f {Q | t}; (INVARIANT i) ∈ Q ⟧ ⟹ ⊩ {r and i} f {Q | t and i}"

| b_invariant_co:
  "⟦ ⊩ {r} f {Q | t}; (p CO q) ∈ Q; (INVARIANT i) ∈ Q ⟧ ⟹
  ⊩ {r} f {Q ∪ {p and i CO q and i} | t}"

| b_transient_basis:
  "⟦ ⊩ {r} f {Q | t};
  ∀a pre state_rel. a ∈ actions_of f ⟶ (pre, state_rel) = action_state_rel a
  ⟶ (∀s. (pre and p) s ⟶ (∃s'. (s, s') ∈ state_rel)) ∧
  (∀s s'. (s, s') ∈ state_rel ⟶ (pre and p) s ⟶ (not p) s') ⟧ ⟹
  ⊩ {r} f {Q ∪ {TRANSIENT p} | t}"

| b_transient_sequencing:
  "⟦ ⊢ {r} c₁ {com_pre c₂}; ⊢ {com_pre c₂} c₂ {t}; ⊩ {r} c₁ {Q | com_pre c₂};
  ∀s. r s ⟶ (∀path∈paths c₁ s. path_will_terminate path);
  ⊩ {com_pre c₂} c₂ {Q'| t}; (TRANSIENT p) ∈ Q' ⟧ ⟹
  ⊩ {r} c₁;;c₂ {{TRANSIENT p} | t}"

| b_transient_inheritance_semi:
  "⟦ ⊢ {r} c₁ {com_pre c₂}; ⊢ {com_pre c₂} c₂ {t}; ⊩ {r} c₁ {Q | com_pre c₂};
  (TRANSIENT p) ∈ Q; ⊩ {com_pre c₂} c₂ {Q'| t}; (TRANSIENT p) ∈ Q' ⟧ ⟹
  ⊩ {r} c₁;;c₂ {{TRANSIENT p} | t}"

| b_transient_inheritance_if:
  "⟦ ∀s. (pre and b) s ⟶ com_pre c₁ s; ∀s. (pre and not b) s ⟶ com_pre c₂ s;
  ⊢ {com_pre c₁} c₁ {t}; ⊢ {com_pre c₂} c₂ {t}; ⊩ {com_pre c₁} c₁ {Q | t};
  (TRANSIENT p) ∈ Q; ⊩ {com_pre c₂} c₂ {Q'| t}; (TRANSIENT p) ∈ Q' ⟧ ⟹
  ⊩ {pre} ⦃pre⦄ IF b THEN c₁ ELSE c₂ {{TRANSIENT p} | t}"

| b_ensures:
  "⟦ ⊩ {r} f {Q | t}; ((p and not q) CO (p or q)) ∈ Q;
  (TRANSIENT (p and not q)) ∈ Q ⟧ ⟹
  ⊩ {r} f {Q ∪ {p EN q} | t}"

| b_leads_to_ensures:
  "⟦ ⊩ {r} f {Q | t}; (p EN q) ∈ Q ⟧ ⟹ ⊩ {r} f {Q ∪ {p ↦ q} | t}"

| b_leads_to_transitive:
  "⟦ ⊩ {r} f {Q | t}; (p ↦ q) ∈ Q; (q ↦ r) ∈ Q ⟧ ⟹
  ⊩ {r} f {Q ∪ {p ↦ r} | t}"

| b_leads_to_disjunction:
  "⟦ ⊩ {r} f {Q | t}; ∀p∈set S. (p ↦ q) ∈ Q ⟧ ⟹
  ⊩ {r} f {Q ∪ {(Or S) ↦ q} | t}"
```

Most of the formalisation follows exactly from the rules we have discussed in Chapter 4. We explain the formalisation of the rules that are either not straightforward or have some changes below.

co Rule

The following rule called `b_co` formalises the **co** rule (Section 4.4.1).

```
⟦⊩ {r} f {Q | t}; ∀s. p s ⟶ q s;
 ∀a pre state_rel.
    a ∈ actions_of f ⟶
    (pre, state_rel) = action_state_rel a ⟶
    (∀s s'. (s, s') ∈ state_rel ⟶ (pre and p) s ⟶ q s')⟧
⟹ ⊩ {r} f {Q ∪ {p CO q} | t}
```

In this rule, we add an assumption that $p \Rightarrow q$ as we have skips in our language, which Misra left unspecified.

Safety Inheritance Rules

While Misra only states one meta-rule for inheritance (Section 4.4.3), we explicitly instantiate it for seq and parallel composition as we have fixed a concrete sequential sublanguage to work with. In our formalisation, the rules are `b_co_inheritance_semi`, `b_co_inheritance_if`, `b_co_inheritance_while`, and `b_co_inheritance_par`.

To explain how the formalisation works, let us take the inheritance rule for sequential composition as an example, shown below. The formalisation of the other language constructs follows a similar pattern.

$$\frac{\dfrac{\{r\}\ c_1\ \{m\} \quad \{m\}\ c_2\ \{s\}}{\{r\}\ c_1; c_2\ \{s\}}\ (1) \qquad \{r\}\ c_1\ \{\sigma \mid m\}\ (2) \quad \{m\}\ c_2\ \{\sigma \mid s\}\ (3)}{\{r\}\ c_1; c_2\ \{\sigma \mid s\}}\ (\mathit{inheritance\_semi})$$

The following is our Isabelle formalisation called `b_co_inheritance_semi`.

```
⟦⊢ {r} c₁ {com_pre c₂}; ⊢ {com_pre c₂} c₂ {t}; ⊩ {r} c₁ {Q | com_pre c₂};
 (p CO q) ∈ Q; ⊩ {com_pre c₂} c₂ {Q' | t}; (p CO q) ∈ Q'⟧
⟹ ⊩ {r} c₁;;c₂ {{p CO q} | t}
```

We formalise (1) using ⊢. It is done as such since the premises/conclusion do not mention any perpetual property. This is represented by ⊢ `{r} c₁ {com_pre c₂}` and ⊢ `{com_pre c₂} c₂ {t}`, which are the premises of the local annotation rule for sequential composition (`b_semi` in Section 6.4.1).

We formalise (2) and (3) using ⊩, i.e. ⊩ `{r} c₁ {Q | com_pre c₂}`, `(p CO q) ∈ Q`, ⊩ `{com_pre c₂} c₂ {Q' | t}`, and `(p CO q) ∈ Q'`.

Invariance Rules

We add a rule to introduce invariants in a specification, called `b_invariant`.

```
⟦⊩ {r} f {Q | t}; ∀s. r s ⟶ i s;
 ∀a pre state_rel.
    a ∈ actions_of f ⟶
    (pre, state_rel) = action_state_rel a ⟶
    (∀s s'. (s, s') ∈ state_rel ⟶ (pre and i) s ⟶ i s')⟧
⟹ ⊩ {r} f {Q ∪ {INVARIANT i} | t}
```

It is defined in a similar way as the **co** rule, with an additional assumption $r \Rightarrow i$, where r is the precondition and i is the invariant.

The remaining invariance rules, `b_invariant_pre_post` and `b_invariant_co`, strengthen the pre and postcondition or **co** with an invariant.

Transient Basis Rule

We formalise the transient basis rule from Section 4.5.1 using the rule `b_transient_basis`.

```
⟦⊩ {r} f {Q | t};
 ∀a pre state_rel.
    a ∈ actions_of f ⟶
    (pre, state_rel) = action_state_rel a ⟶
    (∀s. (pre and p) s ⟶ (∃s'. (s, s') ∈ state_rel)) ∧
    (∀s s'. (s, s') ∈ state_rel ⟶ (pre and p) s ⟶ (not p) s')⟧
⟹ ⊩ {r} f {Q ∪ {TRANSIENT p} | t}
```

The original rule has **transient** $(p \wedge \neg post_f)$ in the conclusion, while in our formalisation, we have **transient** p. This is done because $\neg post_f$ is encoded directly inside the formalisation of transient, which allows us to leave out $\neg post_f$ in the formalisation of this rule.

Transient Sequencing Rule

The following rule called `b_transient_sequencing` formalises part of the transient sequencing rule from Section 4.5.1.

```
⟦⊩ {r} c₁ {com_pre c₂}; ⊢ {com_pre c₂} c₂ {t}; ⊩ {r} c₁ {Q | com_pre c₂};
 ∀s. r s ⟶ (∀path∈paths c₁ s. path_will_terminate path);
 ⊩ {com_pre c₂} c₂ {Q' | t}; (TRANSIENT p) ∈ Q'⟧
⟹ ⊩ {r} c₁;;c₂ {{TRANSIENT p} | t}
```

We have discussed that if f does not always terminate, we require **transient** p to hold in both f and g. On the other hand, if we know f always terminates no matter which path it takes, it is enough for **transient** p to only hold in g. The former is formalised using `b_transient_inheritance_semi`, while the latter is what `b_transient_sequencing` formalises.

6.5 Soundness

The final step of the formalisation is proving soundness. Soundness means all statements which can be derived are true or valid. More specifically, we want to prove that any program specification derived from `biloof_no_perpetual` or from `biloof` is valid. We prove the soundness of these two layers separately.

6.5.1 Soundness of `biloof_no_perpetual`

We want to prove the soundness of the specifications derived from `biloof_no_perpetual` (Section 6.4.1) using the theorem below.

theorem `biloof_no_perpetual_sound`: `⊢ {r} f {t} ⟹ ⊨ {r} f {{} | t}`

The proof is by induction on `⊢ {r} f {t}`. Isabelle generates eight subgoals, each requires proving soundness of a particular rule in `biloof_no_perpetual`.

Recall that the rules in `biloof_no_perpetual` consist of meta-rules and local annotation rules. The meta-rules are proved automatically, while the local annotation rules require induction or case analysis on the execution of small-step semantics ($\rightarrow*$). We prove the soundness of the rules for action and seq in a very similar way to the soundness of Hoare logic. Here, we will discuss the soundness proof of parallel composition further.

Soundness of Parallel Composition

The corresponding subgoal generated by Isabelle is as follows.

```
⟦valid_ann Ps Ts; length Ps = length Ts; 0 < length Ps;
 ∀i<length Ps.
    ⊢ {com_pre (Ps ! i)} Ps ! i {Ts ! i} ∨ Ps ! i = ⦃Ts ! i⦄ POSTANN;
 ∀i<length Ps.
    ⊨ {com_pre (Ps ! i)} Ps ! i {{} | Ts ! i} ∨ Ps ! i = ⦃Ts ! i⦄ POSTANN⟧
⟹ ⊨ {And (map com_pre Ps)} PARALLEL Ps Ts {{} | And Ts}
```

The other soundness proofs in this thesis utilise the fact that the program specifications in the premises are valid (⊨). However, this is not the case for the parallel composition rule, where the fact that the program specifications are derivable (⊢) is more important than being valid. After removing validity from our premises and simplifying the terms, we get

```
⟦(f, s) →* (f', s'); f = PARALLEL Ps Ts; valid_ann Ps Ts; has_terminated f';
 length Ps = length Ts; (not (=) Ts) []; And (map com_pre Ps) s;
 ∀j<length Ts.
    ⊢ {com_pre (Ps ! j)} Ps ! j {Ts ! j} ∨ Ps ! j = ⦃Ts ! j⦄ POSTANN⟧
⟹ And Ts s' ∧ (not (=) f') ABORTED
```

The subgoal above requires us to prove that if we execute a parallel program and it terminates, then the postcondition holds and the resulting program is not `ABORTED`.

The proof is by induction on →*. The base case is an execution that takes zero steps, whereby `f` = `f'` and `s` = `s'`. Therefore, we have `has_terminated (PARALLEL Ps Ts)` in the premises, which is false. The base case is trivially proved.

In the inductive step, we have `(PARALLEL Ps Ts, s) → (f'', s'')` and `(f'', s'') →* (f', s')` for some `f''` and `s''`. Recall that we have three small-step semantics rules for parallel composition, namely `ParA`, `ParD`, and `Par`. Hence, there are three ways `PARALLEL Ps Ts` can make a step.

We present one representative case here, which is the one that makes a step using `Par`. In `Par`, we have `(Ps ! i, s) → (c', s')` in our premises and `(PARALLEL Ps Ts, s) → (PARALLEL Ps' Ts, s')` as our conclusion where `Ps' = Ps[i := c']`.

We prove the soundness of this case by using the induction hypothesis. This requires us to prove

```
valid_ann Ps' Ts ∧
length Ps' = length Ts ∧
And (map com_pre Ps') s' ∧
(not (=) Ts) [] ∧
(∀j<length Ts.
    ⊢ {com_pre (Ps' ! j)} Ps' ! j {Ts ! j} ∨ Ps' ! j = ⦃Ts ! j⦄ POSTANN)
```

We break the conjunction and prove each of the five assumptions one-by-one.

- `valid_ann Ps' Ts`

  We have `valid_ann Ps Ts` as our assumption, which means that each component is stable in any other component. As the i^{th} component is the only component that changes after the parallel composition makes a step, we only need to prove that `Ps' ! i` is stable in any other component, and vice versa. We prove two similar lemmas, each for each direction of stability.

  – `⟦(g, s) → (g', s'); is_com_stable g h; com_pre g s; ⊢ {com_pre g} g {t}⟧`

    ⟹ `is_com_stable g' h`

  – ⟦`(h, s)` → `(h', s'); is_com_stable g h`⟧ ⟹ `is_com_stable g h'`

  As `Ps' ! i` is the resulting reduction from `Ps ! i`, we can prove that the annotations and the actions of `Ps' ! i` are a subset of the annotations and the actions of `Ps ! i` respectively. The property about annotations is used to prove the first lemma whereas the property about actions is used to prove the second lemma.

- `length Ps' = length Ts`

  Since the number of components does not change, this is trivial.

- `(not (=) Ts) []`

  This is immediate as we have `0 < length Ts` in the premises.

- `And (map com_pre Ps') s'`

  This is equivalent to proving ∀`j<length Ps'. com_pre (Ps' ! j) s`. We prove two cases, `i = j` and `i` ≠ `j`.

  For the first case, we prove the following lemma, which is proved by induction on the small-step relation (→).

  ⟦`(c, s)` → `(c', s'); com_pre c s;` ⊢ `{com_pre c} c {t}`⟧ ⟹ `com_pre c' s'`

  The second case requires us to prove that executing `Ps ! i` does not invalidate `com_pre (Ps ! j)`. The proof is immediate once we prove

  ⟦`(f, s)` → `(f', s'); is_ann_stable p f; p s`⟧ ⟹ `p s'`

  The proof of this lemma is by induction on → and is intuitively obvious based on how we define stability.

- ∀`j<length Ts.` ⊢ `{com_pre (Ps' ! j)} Ps' ! j {Ts ! j}` ∨ `Ps' ! j =` ⦃`Ts ! j`⦄ `POSTANN`

  We again break this down into two cases, `i = j` and `i` ≠ `j`. The `i` ≠ `j` case is trivial as the j^{th} component remains the same. We prove the `i = j` case using the following lemma.

  ⟦`(f, s)` → `(f', s'); com_pre f s;` ⊢ `{com_pre f} f {t}; (not (=) f') DONE`⟧
  ⟹ `com_pre f' s'` ∧ ⊢ `{com_pre f'} f' {t}`

  The lemma suggests that the derivability of components is preserved throughout any execution of a program. It is worth noting that this lemma would not work if the rules were not stratified, i.e. this lemma is not true if invariance rules can be applied to strengthen the postcondition. The `POSTANN` construct is also introduced solely in order to help prove this particular lemma.

The lemma is proved by checking that the inverse of each local annotation rule holds. For language constructs that are not mentioned in the rules, we prove that it is impossible to derive a specification of the constructs. For example,

- `⟦⊢ {r} f {t}; f = DONE⟧ ⟹ False`
- `⟦⊢ {r} f {t}; f = c₁;;c₂⟧ ⟹ ⊢ {r} c₁ {com_pre c₂} ∧ ⊢ {com_pre c₂} c₂ {t}`

6.5.2 Soundness of `biloof`

We define the soundness of the specifications derived from `biloof` (Section 6.4.2) below.

theorem `biloof_sound`: `⊩ {r} f {Q | t} ⟹ ⊨ {r} f {Q | t}`

We prove by induction on `⊩ {r} f {Q | t}`. Isabelle generates a subgoal for each rule in `biloof`, which are 20 of them. We discuss four representative cases below.

Soundness of co Rule

The following subgoal is the subgoal generated for the **co** rule (`b_co`).

```
⟦⊨ {r} f {Q | t}; ∀s. p s ⟶ q s; actions_co f p q⟧
⟹ ⊨ {r} f {Q ∪ {p CO q} | t}
```

where `actions_co f p q` means that executing any action of `f` in a `p`-state establishes a `q`-state. The definition in Isabelle is as follows.

```
actions_co f p q ≡
∀a. a ∈ actions_of f ⟶
    (∀pre state_rel.
        (pre, state_rel) = action_state_rel a ⟶
        (∀s s'. (s, s') ∈ state_rel ⟶ (pre and p) s ⟶ q s'))
```

Notice that this is an assumption in `b_co`, which we abbreviate to `actions_co f p q` throughout this section for clarity.

After some simplification, we obtain

```
⟦(f, s) →* (f', s'); (f', s') → (f'', s''); ∀s. p s ⟶ q s; p s';
 actions_co f p q⟧
⟹ q s''
```

The subgoal states that if we start an execution from `(f, s)` and reach a configuration `(f', s')` where `p` holds in `s'`, then if `(f', s')` makes a step to `(f'', s'')`, `q` holds in `s''`.

We have mentioned that `actions_of f' ⊆ actions_of f`. Using that fact, it is now enough to prove the following simpler lemma.

```
⟦(f', s') → (f'', s''); ∀s. p s ⟶ q s; p s'; actions_co f' p q⟧ ⟹ q s''
```

The proof is by induction on →, and the subgoals are proved automatically by Isabelle.

Soundness of co Inheritance Rules

We will explain how the soundness proof works in the inheritance rule for sequential composition (`b_co_inheritance_semi`), shown below. The same approach works for the other language constructs.

```
⟦⊢ {r} c1 {com_pre c2}; ⊢ {com_pre c2} c2 {t}; ⊩ {r} c1 {Q | com_pre c2};
 ⊨ {r} c1 {Q | com_pre c2}; (p CO q) ∈ Q; ⊩ {com_pre c2} c2 {Q' | t};
 ⊨ {com_pre c2} c2 {Q' | t}; (p CO q) ∈ Q'⟧
⟹ ⊨ {r} c1;;c2 {{p CO q} | t}
```

We apply the local annotation rule for sequential composition (`b_semi`) forward from `⊢ {r} c1 {com_pre c2}` and `⊢ {com_pre c2} c2 {t}`, and we obtain `⊢ {r} c1;;c2 {t}`. As ⊢ is sound, we can conclude that `⊨ {r} c1;;c2 {{} | t}`.

Now the lemma looks as following.

```
⟦⊨ {r} c1;;c2 {{} | t}; ⊩ {r} c1 {Q | com_pre c2};
 ⊨ {r} c1 {Q | com_pre c2}; (p CO q) ∈ Q; ⊩ {com_pre c2} c2 {Q' | t};
 ⊨ {com_pre c2} c2 {Q' | t}; (p CO q) ∈ Q'⟧
⟹ ⊨ {r} c1;;c2 {{p CO q} | t}
```

We can prove this using the fact that the **co** rule is sound. That is, it is enough to prove that `actions_co (c1;;c2) p q` holds. To show this, we first show that `actions_co f p q` holds if we have `⊩ {r} f {Q | t} ∧ (p CO q) ∈ Q`.

```
⟦⊩ {r} f {Q | t}; (p CO q) ∈ Q⟧ ⟹ actions_co f p q
```

This is proved by induction on ⊩. Using this fact, we obtain `actions_co` c_1 `p q` and `actions_co` c_2 `p q`. Hence, `actions_co (`c_1`;;`c_2`) p q` holds and `b_co_inheritance_semi` is sound.

Soundness of Invariance Rules

We would like to prove that strengthening the pre and postcondition with an invariant (`b_invariant_pre_post`) preserves the validity of the specification. The other rule which strengthens **co** is proved in a similar way.

```
⟦⊨ {r} f {Q | t}; (INVARIANT i) ∈ Q⟧ ⟹ ⊨ {r and i} f {Q | t and i}
```

After doing some simplification, the corresponding subgoal is

```
⟦r s; i s; invariant i f s;
 reachable_sat (λf' s'. has_terminated f' ⟶ holds t f' s') f s⟧
⟹ reachable_sat (λf' s'. has_terminated f' ⟶ holds (t and i) f' s') f s
```

We can prove this by proving a stronger lemma below which states that `i` must hold at any point in the program.

```
invariant i f s ⟹ reachable_sat (λf'. i) f s
```

This is proved by first unfolding the definitions of `invariant` and `reachable_sat`, then applying induction on →∗.

Soundness of Transient Basis Rule

We want to prove the soundness of the transient basis rule (`b_transient_basis`) using the lemma below.

```
⟦⊩ {r} f {Q | t}; ⊨ {r} f {Q | t}; actions_transient f p⟧
⟹ ⊨ {r} f {Q ∪ {TRANSIENT p} | t}
```

where

```
actions_transient f p ≡
∀a pre state_rel.
   a ∈ actions_of f ⟶
   (pre, state_rel) = action_state_rel a ⟶
   (∀s. (pre and p) s ⟶ (∃s'. (s, s') ∈ state_rel)) ∧
   (∀s s'. (s, s') ∈ state_rel ⟶ (pre and p) s ⟶ (not p) s')
```

Similar to the soundness of the **co** rule, we have the abbreviation `actions_transient f p` for clarity.

Simplifying the terms, we have

```
⟦⊢ {r} f {t}; r s; actions_transient f p; is_path path f s;
 path i = (f', s'); p s'; (not has_terminated) f'⟧
⟹ ∃j≥i. ∃f' s'. path j = (f', s') ∧ ((not p) s' ∨ has_terminated f')
```

Consider the following facts.

- From the `is_path` definition, we can deduce `(f, s) →* (f', s')`. We also know that `f'` has not terminated. Using these two facts, we can prove that there exists some `r'` and `t'` such that `⊢ {r'} f' {t'} ∧ r' s'`.
- For any valid path, it is easy to see that any suffix of that path is also a valid path starting from the later configuration. That is, `λn. path (n + i)` is a valid path that starts from `(f', s')`. We can then add `path' = (λn. path (n + i)) ∧ is_path path' f' s'` in our premises, and rewrite the conclusion as `∃j f'' s''. path' j = (f'', s'') ∧ ((not p) s'' ∨ has_terminated f'')`.
- We have also mentioned that `actions_of f' ⊆ actions_of f` for any `(f, s) → (f', s')`. We can extend this fact to the reflexive transitive closure, i.e. if `(f, s) →* (f', s')`, then `actions_of f' ⊆ actions_of f`. From `(f, s) →* (f', s')` and `actions_transient f p`, we can deduce `actions_transient f' p`.

Using these facts, it is enough to prove

```
⟦⊢ {r'} f' {t'}; r' s'; p s'; actions_transient f' p; is_path path' f' s'⟧
⟹ ∃j f'' s''. path' j = (f'', s'') ∧ (not p) s''
```

While it is intuitive to prove this by induction, it is not very clear what to apply the induction on. If we apply induction on ⊢, then in the parallel composition case, we need to reason about the paths of f [] g from the paths of f and the paths of g. As f and g might interfere with each other when running in parallel, there can be a path in f [] g which f or g will not take if they are to run individually.

Another idea to approach this is to remember the fact that a program has finite text. We know that the reduction of an `If` or `While` to its body does not change the state. However, as there is only a finite amount of program text, it is impossible to have an infinite number of `If` or `While` sitting in any program. Therefore, at some point in the execution, it must reach an action and execute the action. This action will establish `not p`.

We implement the idea above by defining a function that overapproximates how many steps to take at most until we reach an action.

```
fun num_same_state :: "com ⇒ nat" where
  "num_same_state DONE = 1"
| "num_same_state ABORTED = 1"
| "num_same_state (⦃_⦄ ACTION _) = 0"
| "num_same_state (c₁;;c₂) = num_same_state c₁"
| "num_same_state (⦃_⦄ IF _ THEN c₁ ELSE c₂) =
  1 + max (num_same_state c₁) (num_same_state c₂)"
| "num_same_state (⦃_⦄ ⦃_⦄ WHILE b i DO c) = 1 + num_same_state c"
| "num_same_state (PARALLEL Ps Ts) = sum_list (map num_same_state Ps)"
| "num_same_state (⦃_⦄ POSTANN) = 0"
```

The values for `DONE`, `ABORTED`, and `POSTANN` are adjusted accordingly to work with the proofs. As `num_same_state f` is a finite number, we can have `num_same_state f < n` for some `n` as an additional premise.

```
⟦num_same_state f' < n; ⊢ {r'} f' {t'}; r' s'; p s'; actions_transient f' p;
 is_path path' f' s'⟧
⟹ ∃j f'' s''. path' j = (f'', s'') ∧ (not p) s''
```

The proof proceeds by induction on `n`. The base case, `n = 0`, is trivial. In the inductive step, we prove that either the next step of `f'` establishes `not p` (`f'` executes an action) or `num_same_state` is reduced. If `f` establishes `not p`, then we are done. Otherwise, we can use the induction hypothesis and we are also done. Here is the lemma that describes `num_same_state f'' < num_same_state f' ∨ (not p) s''` holds after `f'` makes a step, proved by induction on →.

```
⟦(f', s') → (f'', s''); ⊢ {r'} f' {t'}; r' s'; p s'; actions_transient f' p⟧
⟹ num_same_state f'' < num_same_state f' ∨ (not p) s''
```

Discussion

This transient basis rule is not sound if we have an arbitrary small-step semantics. More specifically, in our small-step semantics for a while loop when the condition is false (`WhileF` in Section 6.1.3), if we reduce it to `DONE` instead of skip, then this rule is not sound.

Consider the program below.

```
WHILE true DO
 WHILE false DO
   true → x := x + 1
```

If `WhileF` reduces the while loop to `DONE`, then we can prove that **transient** $(x = 0)$ holds in this example. There is only one action in this program, i.e. `true → x := x + 1`. It is easy to see that $x = 0$ implies the guard `true`. Additionally, executing `x := x + 1` when $x = 0$ holds establishes $x \neq 0$.

However, this is not true, i.e. **transient** $(x = 0)$ does not hold in this program. If $x = 0$ holds at the start of the program, then no matter how long we execute the program, we will never reach a state in which $x \neq 0$ holds.

To fix this, we introduce a skip to be executed after we get out of a loop. This is exactly what the rule `WhileF` in the small-step semantics does. As the skip appears in the actions of a while loop, this rule can be proved sound.

In the example above, we now have two actions, namely the action that is in the program text, and a "ghost" action, skip. Executing a skip when $x = 0$ holds does not change the state. Hence, we can no longer conclude that **transient** $(x = 0)$ holds in the program.

Chapter 7

Evaluation

This chapter presents an evaluation of the results of this thesis, i.e. the formalisation and soundness proof of Bilateral Proof. A mechanisation of the safety property proof of the distributed counter example using the formalisation we have in Isabelle will also be shown.

7.1 Formalisation and Soundness Proof

The formalisation and soundness proof are automatically evaluated by Isabelle. As a theorem prover, Isabelle automatically verifies if the proofs are correct. This prevents any error in the proofs that might occur if we were doing pen-and-paper proofs instead.

7.2 Example: Distributed Counter

We have successfully mechanised the safety property proof of the distributed counter example from Section 4.4.5 in Isabelle using the formalisation we now have.

7.2.1 Distributed Counter Formalisation

Recall the distributed counter program to be a parallel composition of a finite number of threads. For simplicity, we assume that there are only two threads running in parallel. The proof for an arbitrary number of threads works the same way.

We formalise the distributed counter as follows. We call the first thread `dist_ctr_com1` and the second thread `dist_ctr_com2`. The distributed counter program is defined using the abbreviation `dist_ctr`.

```
definition Assign :: "bexp ⇒ vname ⇒ (state ⇒ nat) ⇒ state_rel" where
  "Assign G n v ≡ {(s, s'). G s ∧ s' = s(n := v s)}"

definition dist_ctr_com :: "vname ⇒ vname ⇒ vname ⇒ com" where
  "dist_ctr_com ctr old new ≡
    ⦃true⦄
    ACTION {(s, s'). s' = (s(old := 0))(new := 0)};;
    ⦃true⦄ ⦃true⦄
    WHILE true true DO
      (⦃true⦄
      ACTION (Assign true new (λs. s old + 1));;
      ⦃λs. s new = s old + 1⦄
      ACTION
        ( Assign (λs. s ctr = s old) ctr (λs. s new)
        ∪ Assign (λs. s ctr ≠ s old) old (λs. s ctr)))"

definition dist_ctr_com1 where
  "dist_ctr_com1 ≡ dist_ctr_com ''ctr'' ''old1'' ''new1''"

definition dist_ctr_com2 where
  "dist_ctr_com2 ≡ dist_ctr_com ''ctr'' ''old2'' ''new2''"

abbreviation dist_ctr where
  "dist_ctr ≡ PARALLEL [dist_ctr_com1, dist_ctr_com2] [true, true]"
```

7.2.2 Safety Property Proof

We wish to prove the safety property

$$\forall m \in \mathbb{Z}.\ \mathit{ctr} = m\ \textbf{co}\ (\mathit{ctr} = m \vee \mathit{ctr} = m + 1)$$

in our distributed counter program.

The following is the formalisation in Isabelle.

```
⟦p = (λs. s ''ctr'' = m); q = (λs. s ''ctr'' = m ∨ s ''ctr'' = m + 1)⟧
⟹ ⊨ {true} dist_ctr {{p CO q} | true}
```

Since we know `biloof` is sound, we can substitute the conclusion with
`⊩ {true} dist_ctr {{p CO q} | true}`

We are not able to apply the inheritance rule for parallel composition straight away as the format of the conclusion is a bit different. We need to rewrite it as follows.

```
⟦Ps = [dist_ctr_com1, dist_ctr_com2]; Ts = [true, true];
 p = (λs. s ''ctr'' = m); q = (λs. s ''ctr'' = m ∨ s ''ctr'' = m + 1)⟧
⟹ ⊩ {com_pre (PARALLEL Ps Ts)} PARALLEL Ps Ts {{p CO q} | And Ts}
```

Applying `b_co_inheritance_parallel`, we need to prove these three subgoals.

- `valid_ann [dist_ctr_com1, dist_ctr_com2] [true, true]`

-
  ```
  ⟦p = (λs. s ''ctr'' = m); q = (λs. s ''ctr'' = m ∨ s ''ctr'' = m + 1)⟧
  ⟹ ⊩ {com_pre dist_ctr_com1} dist_ctr_com1 {{p CO q} | true}
  ```

-
  ```
  ⟦p = (λs. s ''ctr'' = m); q = (λs. s ''ctr'' = m ∨ s ''ctr'' = m + 1)⟧
  ⟹ ⊩ {com_pre dist_ctr_com2} dist_ctr_com2 {{p CO q} | true}
  ```

The first subgoal concerns the locality of the assertions and is proved automatically by Isabelle.

The second and third subgoal are analogous to each other. We will discuss only the second subgoal here. By rearranging the terms and applying `b_co`, we now need to prove

-
  ```
  ⟦p = (λs. s ''ctr'' = m); q = (λs. s ''ctr'' = m ∨ s ''ctr'' = m + 1)⟧
  ⟹ ∀a. a ∈ actions_of dist_ctr_com1 ⟶
           (∀pre state_rel.
               (pre, state_rel) = action_state_rel a ⟶
               (∀s s'. (s, s') ∈ state_rel ⟶ (pre and p) s ⟶ q s'))
  ```

-
  ```
  ⟦p = (λs. s ''ctr'' = m); q = (λs. s ''ctr'' = m ∨ s ''ctr'' = m + 1)⟧
  ⟹ ⊩ {com_pre dist_ctr_com1} dist_ctr_com1 {{} | true}
  ```

The first subgoal is automatically proved by Isabelle.

In the second subgoal, we apply `bl_biloof_no_perpetual` to the conclusion and obtain `⊢ {com_pre dist_ctr_com1} dist_ctr_com1 {true}` as the new conclusion. The proof is fairly straightforward, using the local annotation rules in `biloof_no_perpetual`.

Chapter 8

Summary

This chapter summarises the work in this thesis, the contribution this thesis made, and potential future work.

8.1 Results

The thesis contains a formalisation and soundness proof of 28 rules in Bilateral Proof worth 2000 lines of Isabelle code. These 28 rules represent a large subset of the rules in Bilateral Proof which are useful for any future verification using Bilateral Proof. Thus, the aim of this thesis, which is to mechanise a representative subset of Bilateral Proof and prove the soundness of the rules, has been met.

Assumptions

The soundness proof relies upon these two assumptions.

1. The rules are stratified into two layers. The first layer includes rules that do not mention perpetual properties. The second layer contains the rules in the first layer as well as rules that introduce or use perpetual properties.
2. There is always an extra action in a while loop, i.e. skip. This skip is not something a programmer would write, but we add it in our semantics because otherwise, the transient basis rule is not sound.

Limitations and Approach

The formalisation in this thesis does not include two of the transient rules. They are the concurrency rule and the inheritance rule for while loops.

To implement the concurrency rule, we need a definition of fairness and program counter. The challenges include finding the right definition of fairness and to change the whole program semantics with a program counter, which is a heavyweight operation.

The inheritance rule for while loops can be defined similarly as the inheritance rule for sequential composition and if/else, in which we formalised. However, the extra skip that we add in while loops may require some modification to the rule.

8.2 Contribution

First of all, this thesis contributes by providing the soundness proof of Bilateral Proof. Jayadev Misra published the Bilateral Proof paper without proving the soundness. As briefly discussed in Chapter 4, this framework is able to provide a simpler way to prove the correctness of concurrent programs. However, the logic could not be relied on until there is a proof whether it is actually correct, i.e. the soundness proof.

We have also found a couple of assumptions that we need in order to prove the soundness. Finally, the formalisation we have in Isabelle can be used for any future verification using Bilateral Proof in Isabelle.

8.3 Future Work

There are a lot of opportunities for future work based on this thesis. The first one is to add the formalisation and soundness proof of the remaining transient rules. We can then investigate what progress properties Bilateral Proof can or cannot prove.

Additionally, it would be interesting to know whether the proofs in Bilateral Proof are always simpler than Owicki-Gries or Rely-Guarantee. In order to do so, a comparison of the proofs in the three frameworks for various programs is needed.

A completeness proof of Bilateral Proof and Owicki-Gries/Rely-Guarantee can also be conducted, i.e. whether Bilateral Proof is able to verify every program that Owicki-Gries/Rely-Guarantee are able to.

Finally, once all the work is done, we hope Bilateral Proof can be a useful new framework that allows scalable verification of concurrent programs, given its simplicity and compositionality.

Bibliography

[CM88] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[Jon81] C. B. Jones. *Development methods for computer programs including a notion of interference.* PhD thesis, Oxford University Computing Laboratory, 1981.

[Lam77] L. Lamport. Proving the correctness of multiprocess programs. *IEEE transactions on software engineering*, (2):125–143, 1977.

[Mis95] J. Misra. A logic for concurrent programming: Safety. *Journal of Computer and Software Engineering*, 3(2):239–272, 1995.

[Mis17] J. Misra. Bilateral proofs of safety and progress properties of concurrent programs. 2017.

[NPW02] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS.* Springer, 2002.

[OG76] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs i. *Acta informatica*, 6(4):319–340, 1976.

[Pau94] L. C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science.* Springer Science & Business Media, 1994.

[PN02] L. Prensa Nieto. *Verification of parallel programs with the Owicki-Gries and rely-guarantee methods in Isabelle/HOL.* PhD thesis, Technische Universität München, 2002.

[Tan09] A. S. Tanenbaum. *Modern Operating Systems.* Pearson Education, Inc., 2009.

[Vaf08] V. Vafeiadis. Modular fine-grained concurrency verification. Technical report, University of Cambridge, Computer Laboratory, 2008.

[XdRH97] Q. Xu, W.-P. de Roever, and J. He. The rely-guarantee method for verifying shared variable concurrent programs. *Formal Aspects of Computing*, 9(2):149–174, 1997.