NECESSARY LIBERAL PRECONDITIONS:
A PROOF SYSTEM

Master's Thesis in Informatics

ANRAN WANG
School of Computation, Information and Technology - Informatics
Technical University of Munich

Anran Wang: *Necessary Liberal Preconditions:*
A Proof System

NECESSARY LIBERAL PRECONDITIONS: A PROOF SYSTEM
NOTWENDIGE LIBERALE VORBEDINGUNGEN: EIN BEWEISSYSTEM

Master's Thesis in Informatics

ANRAN WANG, B.SC.
School of Computation, Information and Technology - Informatics
Technical University of Munich

Examiner:	Prof. Jan Křetínský
Supervisors:	Prof. Benjamin Lucien Kaminski
	Lena Verscht, M.Sc.
Submission date:	

Anran Wang: *Necessary Liberal Preconditions:*
A Proof System

DECLARATION

Ich versichere, dass ich diese Masterarbeit selbstständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich,

Anran Wang

For my parents Shizhu Wang and Derun Yuan, who love me patiently.

For Christian Schuler, who loves me funnily.

For my friends, who like me.

For me, who?

ABSTRACT

This thesis investigates the necessary liberal precondition, an overapproximation of Dijkstra's weakest liberal precondition (wlp) transformer. A discussion of the semantics of wlp and its relatives hints at a scenario where the necessary liberal precondition is useful: When the entirety of undesirable postconditions is unknown, by further relaxing wlp, the programmer ends up with a precondition that is likely to lead the program to known or unknown undesired final states upon termination.

ZUSAMMENFASSUNG

Diese Arbeit untersucht die notwendige liberale Vorbedingung, eine Überannäherung an Dijkstras schwächsten liberalen Vorbedingungstransformator (wlp). Eine Diskussion der Semantik von wlp und seinen Verwandten deutet auf ein Szenario hin, in dem die notwendige liberale Vorbedingung nützlich ist: Wenn die Gesamtheit der unerwünschten Nachbedingungen unbekannt ist, erhält der Programmierer durch weitere Lockerung von wlp eine Vorbedingung, die das Programm bei Beendigung wahrscheinlich in bekannte oder unbekannte unerwünschte Endzustände führt.

摘要

本论文研究了必要自由前提条件，即 Dijkstra 的最弱自由前提条件 (wlp) 的涵盖近似。 首先讨论 wlp 及其近似方法的语义，这暗示了一个必要自由前提条件有用的场景： 当我们不知道所有不好的后置条件时，程序员可以通过进一步放松wlp来找到一个前提条件，这个条件很可能会导致程序在终止时进入不好的最终状态，不论这些状态是已知或未知的。

CONTENTS

Part I

HOARE TRIPLES AND DIJKSTRA'S PREDICATE TRANSFORMERS

In this part, I explain the definition for the formalism used in this thesis, discuss some of the definitions, and demonstrate the semantics of the predicate transformers using graphs and operational semantics.

1 BACKGROUND

MOTIVATION In 1739, the Scottish philosopher David Hume questioned why we know that the sun will rise tomorrow, "*tho' 'tis plain we have no further assurance of these facts, than what experience affords us*" [11]. Hume's question about causality is daunting, yet most of us are not in crisis because we doubt if the sun rises tomorrow. The reason is probably that we believe in physics, astrology, and the rules and formulas that assure us the universe works in a certain way, hence the sun rises tomorrow. It is exactly the rules and formulas this thesis attempts to investigate, in the realm of computer programs, with which we are certain that the equivalent version of the sun in a program will rise tomorrow.

Computer programs are ubiquitous in almost every aspect of human life. We want them to solve our problem efficiently, and correctly. Fortunately, brilliant scientists and engineers have taken the matter into their hands. A recent example is seL4 [14], the first formally proven operating system kernel against overflows, memory leaks, non-termination, etc. using the interactive theorem prover Isabelle/HOL [21]. Its verification brings great potential to safety-critical fields like aircrafts and autonomous cars, the latter of which is to be in mass production in 2024.[1]

Imagine soon being driven by an autonomous car carrying seL4. It is desirable that it delivers us to the correct destination, and never get stuck driving around the same block without making progress. Delivering the correct result and stopping eventually is called total correctness.

To know "for sure", we could verify programs using formal methods. One famous method is Hoare triples [9]. A Hoare triple contains three parts: a precondition, a program, and a postcondition. They are written as such: $G\ \{C\}\ F$. It states that if the system starts in a state that satisfies the precondition, then the state after the execution of the program will satisfy the postcondition, provided that the program terminates. Hoare triples are elegant in that once we have appropriate preconditions, we can follow their reference rules on sequential programs with ease. But with Hoare triples in their original form, we know the program is correct, but we are not sure of its termination. This is called partial correctness.

To prove a program totally correct, Dijkstra presented the weakest precondition transformer [3] (wp): starting with a postcondition, it works backwards and calculates what the weakest precondition is that guarantees both correctness and termination. In Hoare triples, the precondition is a sufficient condition for the program to be correct in that the final state will satisfy the desired postcondition, while with wp we obtain a necessary and sufficient precondition. Relaxing the commitment for termination, Dijkstra also proposed the weakest liberal precon-

1 https://sel4.systems/news/2023#nio-skyos, accessed 10.03.2024.

dition transformer [5] (wlp) which delivers preconditions so that the program either terminates correctly or never terminates, proving partial correctness.

Since then, a plethora of research projects blossomed and yielded fruitful results. Not only did Hoare Logic receive numerous copious attention during the first decade of its proposal [1], it has become the basis for program verification now [7]. Hoare triple is applied in low-level programming languages [19], quantum programs [26], distributed real-time systems [10], and so on.

Likewise, Dijkstra's wp transformer also spawned various scientific work on probabilistic programs [13], quantum programs [16], continuous programs [2], and so much more. Especially inspiring for this thesis is the work by O'Hearn named incorrectness logic [22]. It studies an underapproximation (subset) of the strongest postcondition transformer [5] (sp) by Dijkstra, focusing on reachability of final states to prove the existence of bugs. He proposed the incorrectness triple starting from the implication

$$F \implies sp.C.G$$

(F is an underapproximation of the strongest postcondition of precondition G w.r.t. program C), whereas in this thesis, we attempt to study the implication

$$wlp.C.F \implies G$$

(G is an overapproximation of the strongest liberal precondition of postcondition F w.r.t program C). Despite focusing on different transformers and relations, incorrectness logic gives great intuition on how to approach the overapproximation over wlp.

CONTRIBUTION This thesis yields results both in pre- and postcondition transformers and the implication $wlp.C.F \implies G$. The results are listed below in chronological order:

1. We proved that the definition of the weakest precondition transformer of while-programs by Dijkstra [3] and the one using the least fixed point coincide, and gave intuition to explain the necessity of using the least fixed point instead of using any other fixed point.

2. Following the previous result, we deduce that the greatest fixed point is necessary to define the weakest liberal precondition.

3. We relate the pre- and postcondition transformers under angelic or demonic considerations over non-determinism using relations like termination, reachability, conjugation, implication, and equivalence. We then present the results as a graph like in Figure 1.1.

4. We discuss all possible scenarios of the overapproximation $wlp.C.F \implies G$, what we call the necessary liberal preconditions, concluding that without further restrictions, the overapproximate G can contain any initial states. However, we show that the only certainty is that initial states satisfying $\neg G$ will lead to termination in states satisfying $\neg F$. Following this, we demonstrate the usefulness of this overapproximation with an example.

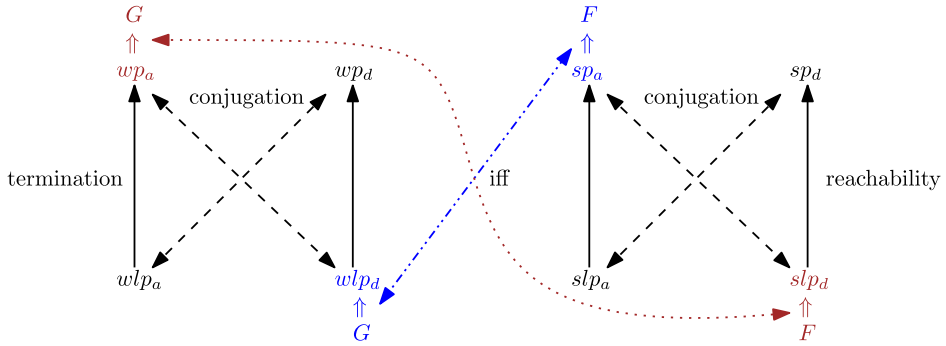


Figure 1.1: Relating wp, wlp, sp, slp

5. We also provide a proof system that captures this overapproximation triple, and prove its soundness.

6. We also notice a special group of preconditions, under whose control the execution is possible to terminate in both the desired postcondition and its opposite, like the green parts in Figure 1.2. We capture these precondi-

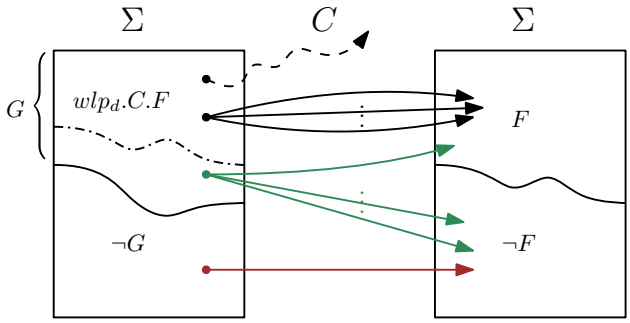


Figure 1.2: Necessarily Liberal Precondition G That Additionally Has Green Dots

tions by approaching from top and bottom, namely finding scenarios that the necessary liberal precondition G underapproximates these special preconditions, and finding scenarios where G overapproximates them. In this way, we can establish an equivalence between G and wlp added with the special preconditions, given constraints. This G is exactly wlp with angelic non-determinism.

OUTLINE After explaining our motivation in this chapter, we proceed to build the formal foundations of the formalism used in this thesis in Chapter 2. We first present a table of notations we adopt, then give a soft introduction of Hoare logic in Section 2.2, followed by Dijkstra's Guarded Command Language, which we use throughout the thesis. Then we display the weakest precondition transformer in Section 2.4, comparing it against Hoare logic, then give a detailed discussion about the use of fixed points while defining while loops. Finally, we add the notion of non-determinism to the wp transformer, and discuss the difference between angelic and demonic non-determinism and explain our design choice.

Following this, we show the weakest liberal precondition transformer in Section 2.5 and the strongest postcondition transformer in Section 2.6. Subsequently, we introduce a big-step semantics in Section 2.7 to help us define the semantics of the aforementioned transformers in Section 2.8 and list some properties in Section 2.9.

In Chapter 3, we first relate wp, wlp, sp, slp together using various relations, then illustrate them in a graph in Section 3.1. This helps us distinguish the necessary liberal precondition with other implications that are researched, so that we can analyze the scenarios of our overapproximation in Section 3.3. Then we provide an example to convey the usefulness of this overapproximation, and show its proof system as well as prove its soundness. We then discover a special scenario that is of interest, and find restraints that qualifies this special scenario, then prove their correctness in Section 3.4. Finally, we summarize our conclusions in Chapter 4 and propose possible future work.

2

PRELIMINARIES

2.1 NOTATIONS

Before proceeding, we clarify the notations used in this thesis, which are not uncommon in materials of computer science and mathematics. Readers are encouraged to skip this section and refer back to it if needed. The notations and their meaning are listed in Table 2.1.

Notation	**Meaning**
$\mathcal{X}$	set of program variables
$\mathcal{V}$	set of values
$\sigma : \mathcal{X} \to \mathcal{V}$	program state
Σ	set of program states
$\mathcal{C}$	set of programs
$\mathcal{P}$	set of predicates
$F : \Sigma \to \{true, false\}$	predicate
$F := \{\sigma \in \Sigma \mid F(\sigma)\}$ (*)	the set described by a predicate $F \in \mathcal{P}$
$F(\sigma)$ (**) $F(\sigma) = true$ (**) $\sigma \vDash F$ $\sigma \in F$	state s satisfies predicate F; F is true when system is in state σ
$\sigma \xrightarrow{c} \tau$	from initial state σ, an execution of program c terminates at final state τ
$\exists x.P : F$ $\forall x.P : F$	syntactic sugar for $\exists x.(P \wedge F)$ $\forall x.(P \wedge F)$

Table 2.1: Symbols and Notations

It is worth noting that we regard program states as total functions - we assume that we can assign some default values to variables in case they are undefined. We also simplify matters by assuming that there is only one interpretation as a total function from predicates to truth values. As a result, we can regard predicates as (total) functions from program states to truth values. We also overload the symbols for predicates and use them to identify the sets they describe as shown in Line (*).

By default, we take $F(\sigma)$ to mean the same as $F(\sigma) = true$ for convenience's sake as shown in Lines (**). We use the equation symbol $=$ to denote equivalences, and the symbols $:=$ for assignments and definitions.

The operators in descending binding power: $\neg, \in, \wedge, \vee, \implies, Q\_.\_ : \_$ where Q is a quantifier: $Q \in \{\exists, \forall\}$. Implication binds to the left: $A \implies B \implies C$ is equivalent to $(A \implies B) \implies C$. Now we can proceed to discuss proof rules and systems that are relevant for this thesis.

2.2 HOARE LOGIC

Since the beginning of the 1960s, scholars have been researching the establishment of mathematics in computation [6, 18] to have a formal understanding and reasoning of programs. One of the most known methods is Hoare logic.

In 1969, C.A.R. Hoare wrote *An Axiomatic Basis for Computer Programming* [9] to explore the logic of computer programs using axioms and inference rules to prove the properties of programs. He introduced sufficient preconditions that guarantee correct results but do not rule out non-termination. A selection of the axioms and rules are shown in Table 2.2. [1]

$\{F[x/e]\}$ is obtained by substituting occurrences of x by e.

Axiom of Assignment	$F[x/e]\ \{x := e\}\ F$
Rules of Consequence	If $G\ \{C\}\ F$ and $F \Rightarrow P$ then $G\ \{C\}\ P$
	If $G\ \{C\}\ F$ and $P \Rightarrow G$ then $P\ \{C\}\ F$
Rule of Composition	If $G\ \{C_1\}\ F_1$ and $F_1\ \{C_2\}\ F$ then $G\ \{C_1; C_2\}\ F$
Rule of Iteration	If $(F \wedge B)\ \{C\}\ F$ then $F\ \{\text{while } B \text{ do } C\}\ \neg B \wedge F$

Table 2.2: Inference Rules for Valid Hoare Triple [2]

Semantically, a Hoare triple $G\ \{C\}\ F$ is said to be valid for (partial) correctness, if the execution of the program C with an initial state satisfying the precondition G leads to a final state that satisfies the postcondition F, provided that the program terminates. The definitions in Table 2.2 indeed correspond to this intended semantics. Formal soundness proofs can be found in Krzysztof R. Apt's survey [1] in 1981. As an example, consider the rule of composition: if the execution of program C_1 changes the state from G to F_1, and C_2 changes the state from F_1 to F, then executing them consecutively should bring the program state from G to F, with the intermediate state F_1.

The missing guarantee of termination can be seen in the rule of iteration: consider the triple $x \leqslant 2\ \{\text{while } x \leqslant 1 \text{ do } x := x * 2\}\ 1 < x \leqslant 2$, it is provable in Hoare

1 Non-determinism was not considered in the original paper, so we treat the programs here as deterministic. With deterministic programs, one initial state corresponds to one final state. In case of non-termination, there is simply no final state.

2 We omit the symbol $\vdash$ in front of a Hoare triple, which denotes "valid/provable", for better readability.

logic with the following proof tree. However, this while-loop will not terminate in case $x \leqslant 0$ in the initial state.

$$\dfrac{\dfrac{}{x \leqslant 1\ \{x := x * 2\}\ x \leqslant 2}\ \textbf{Axiom of Assignment}}{x \leqslant 2\ \{\text{while } x \leqslant 1 \text{ do } x := x * 2\}\ 1 < x \leqslant 2}\ \textbf{Rule of Iteration}$$

Using style taken from Kaminski's dissertation [12], Figure 2.1 illustrates a valid Hoare triple: Σ represents the set of all states, the section denoted with G includes the states that satisfy the predicate G. The arrows from left to right denote the executions of program C. The dashed arrows denote non-terminating executions.

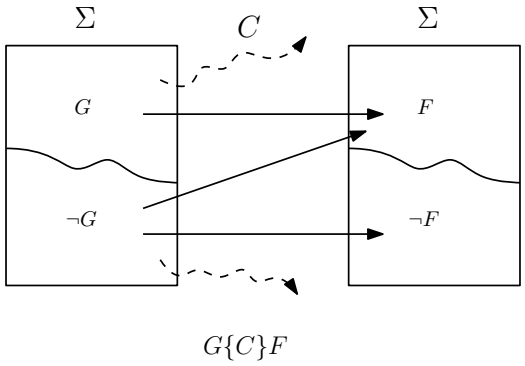


Figure 2.1: Valid Hoare Triple (Deterministic)

A sensible advancement of Hoare logic would be to also prove termination, i.e. to eliminate the arrows from G to the abyss. Supplementing Hoare logic with a termination proof is done by Zohar Manna and Amir Pnueli in 1974 [17], where they introduced what is called a loop variant, a value that decreases with each iteration. The name is in contrast to loop invariant, concretely the F in **Rule of Iteration** in Table 2.2, which is constant before and after the loop.

Another advancement would be to find the necessary and sufficient preconditions that grant us the post-properties, i.e. to eliminate the arrows from ¬G to F in Figure 2.1, which is what Edsger W. Dijkstra accomplished with his weakest precondition transformer in 1975 [3], among other things.

2.3 GUARDED COMMAND LANGUAGE

From now on we use Dijkstra's (non-deterministic) guarded command language (GCL) [3] to represent programs and to include non-determinism (starting from Section 2.4.3). For better readability, we use an equivalent[3] form of GCL that is similar to modern pseudo-code as shown in Table 2.3.

The assignment, sequential composition, conditional choice, while-loop commands conform to their usual meaning. The non-deterministic choice $\{C_1\}\square\{C_2\}$

3 Specifically, $\texttt{if}\ (\varphi)\ \{C_1\}\ \texttt{else}\ \{C_2\}$ is equivalent to $\texttt{if}\ \varphi \to C_1\ []\ \neg\varphi \to C_2\ \texttt{fi}$ in Dijkstra's original style [3]; $\{C_1\}\square\{C_2\}$ is equivalent to $\texttt{if}\ \texttt{true} \to C_1\ []\ \texttt{true} \to C_2\ \texttt{fi}$.

C ::=	$x := e$	$\mid C; C$	$\mid \{C\}\square\{C\}$
	assignment	sequential composition	non-deterministic choice
	$\mid \texttt{if}\ (\varphi)\ \{C\}\ \texttt{else}\ \{C\}$	$\mid \texttt{while}\ (\varphi)\ \{C\}$	$\mid \texttt{skip} \quad \mid \texttt{diverge}$
	conditional choice	while-loop	

Table 2.3: Guarded Command Language

chooses from two programs non-deterministically. It is however not probabilistic, meaning we do not know the probabilistic distribution of the outcome of the choice.

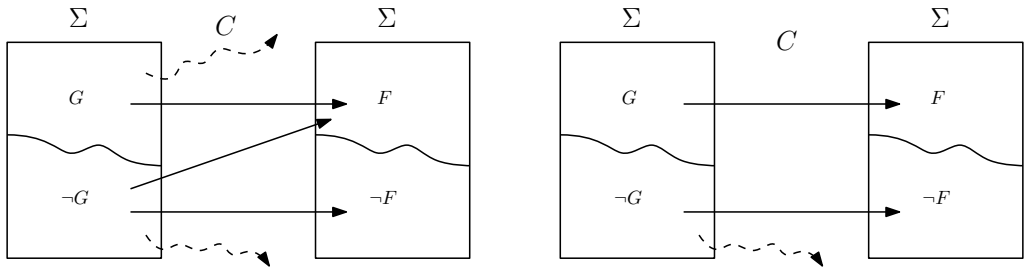
When skip is executed, the program state does not change and the consecutive part is executed. When diverge is executed, the execution never stops and the program can not reach a final state.

In our representation of GCL, non-determinism is explicitly constructed via the infix operator $\square$, whereas in its original definition, non-determinism occurs when the guards within the if and while commands are not mutually exclusive [5]. Additionally, the if statement in Dijkstra's GCL is equivalent to divergence in case non of its guards are true, but in our version this can no longer happen because of the Law of Excluded Middle: the predicate φ must be either true or false, so either the "then" branch or the "else" branch is activated. Consequently, non-termination can only originate from either the diverge or the while command.

2.4 WEAKEST PRECONDITIONS

2.4.1 *The Deterministic Case*

To better relate Hoare triples and Dijkstra's weakest precondition transformer, we first focus on deterministic programs. The goal is to find the necessary and sufficient precondition such that the program is guaranteed to terminate in a state that satisfies the postcondition. Figure 2.2 shows it graphically alongside the figure for valid Hoare triples. We can see that in Figure 2.2.2, the arrows from G to non-termination and from $\neg$G to F are absent.



$G\{C\}F$ (1) Valid Hoare Triple

$G = wp.C.F$ (2) Weakest precondition

Figure 2.2: Valid Hoare Triple vs. Weakest Precondition (Deterministic)

The definition of the weakest precondition transformer is inductively over the program structure in lambda-calculus style[4] as in Table 2.4:

C	**wp.C.F**
skip	F
diverge	false
$x := e$	$F[x/e]$
$C_1; C_2$	$wp.C_1.(wp.C_2.F)$
if (φ) $\{C_1\}$ else $\{C_2\}$	$(\varphi \wedge wp.C_1.F) \vee (\neg\varphi \wedge wp.C_2.F)$
while (φ) $\{C'\}$	$\mathrm{lfp}\ X.(\neg\varphi \wedge F) \vee (\varphi \wedge wp.C'.X)$

Table 2.4: The Weakest Precondition Transformer for Deterministic Programs [12]

$F[x/e]$ is F where every occurrence of x is syntactically replaced by e.
$\mathrm{lfp}\ X.f$ is the least fixed point of function f with variable X.
Let

$$\Phi(X) := (\neg\varphi \wedge F) \vee (\varphi \wedge wp.C'.X)$$

be the characteristic function, then wp for while-loop can be defined as:

$$wp.(\mathrm{while}(\varphi)\{C'\}).F = \mathrm{lfp}\ X.\Phi(X)$$

Most of the definitions in Table 2.4 are intuitive and correspond to their counterparts in Hoare logic, while those for diverge and while deserve special attention. Since wp aims for total correctness, a program starting in an initial state satisfying the precondition $wp.\mathrm{diverge}.F$ should terminate in a final state satisfying the postcondition F. Because diverge does not terminate, there is no such precondition and wp for diverge should be false.

4 For example, $wp.C.F$ can be seen as $wp(C, F)$ in "typical" style, where wp is treated as a function that has two parameters. The advantage of lambda-calculus style is scalability, we can simply extend the aforementioned function to $wp.C.F.\sigma$ where σ means the initial state. Here wp is treated as a function that has three parameters, if we were to write it in the "typical" style. It is then questionable whether we changed the type of wp.

The definition for the while-loop [12] is trickier, but we can verify its correctness by recalling Dijkstra's original definition in the following section.

2.4.2 *Loops and Fixed Points*

DEFINING LOOPS In Dijkstra's original paper [3], he defined wp for while-loops based on its (intended) semantics, i.e. the precondition that guarantees loop termination with the required postcondition within a certain number of iterations.

Let

$$\mathit{WHILE} = \mathit{while}(\varphi)\{C'\} \qquad \text{and} \qquad \mathit{IF} = \mathit{if}\ (\varphi)\{C'\}\ \mathit{else}\ \{\mathit{diverge}\}.$$

Note that IF is originally defined as $\underline{\text{if}}\ B_1 \to SL_1\ []\ \ldots\ []\ B_n \to SL_n\ \underline{\text{fi}}$, where B_i are guards, and SL_i are sub-programs that are executed once their corresponding guards are evaluated to true, thus the name guarded command language. If multiple guard are true, then any of the corresponding sub-programs can be executed, which is how non-determinism is realized in Dijkstra's GCL. If none of the guards is true, then the program does not terminate in a normal state, rather diverges.

In our flavor of GCL, non-determinism is denoted specifically with the operator $\square$, and $\mathit{if}\ (\varphi)\ \{C_1\}\ \mathit{else}\ \{C_2\}$ does not lead to divergence in case its guard φ evaluates to true, rather the else-branch. In essence, the if in our flavor of GCL is equivalent to $\underline{\text{if}}\ \varphi \to C_1\ []\ \neg\varphi \to C_2\ \underline{\text{fi}}$. As a result, to replicate $\mathit{IF} = \underline{\text{if}}\ B_1 \to SL_1\ []\ \ldots\ []\ B_n \to SL_n\ \underline{\text{fi}}$ where divergence is possible in case none of the guards is true, we need $\mathit{diverge}$ instead of skip in the else-branch.

Rewriting Dijkstra's definition in a form conforming to our style, he defines

$$H_0(F) = (\neg\varphi \wedge F) \qquad \text{and} \qquad H_k(F) = \mathit{wp}.\mathit{IF}.H_{k-1}(F) \vee H_0(F).$$

IF is defined in such way that $\mathit{wp}.\mathit{IF}.X$ is the weakest precondition that makes sure the guard of C' discharges once and C' is executed once, leaving the program in a state satisfying X. As a result, $H_k(F)$ corresponds to the weakest precondition such that the program terminates in a final state satisfying F after at most k iterations.

Then by definition:

$$\mathit{wp}.\mathit{WHILE}.F = (\exists k \geqslant 0 : H_k(F)) = \bigvee_{k \geqslant 0} H_k(F) \tag{2.1}$$

LEAST FIXED POINTS The definition in Table 2.4, however, uses the least fixed point of the characteristic function $\Phi(X)$. We can understand the use of fixed point in two ways.

Firstly, a precondition G being a fixed point of the characteristic function implies that

$$G = \Phi(G) = (\neg\varphi \wedge F) \vee (\varphi \wedge \mathit{wp}.C'.G) \tag{2.2}$$

This means if G is satisfied before the execution of WHILE, then termination is possible (left side of the disjunction) and repeated execution of C' is possible

(right side of the disjunction): For WHILE to enter the loop to execute C', ϕ must evaluate to true. Plugging it in Equation 2.2 results in an equation $G = wp.C'.G$, which means that G is invariant before and after the execution of C'. In other words, after one execution of C', G stays satisfied, which makes a further execution of C' possible. In short, ***all fixed points of function Φ can potentially lead to non-termination***.

Secondly, it is standard practice to require the semantics of WHILE to be equivalent to the semantics of $\mathtt{if}(\varphi)\{C';WHILE\}\ \mathtt{else}\ \{\mathtt{skip}\}$ Note that here the else-branch is defined with skip instead of diverge like the previous paragraph, because in case $\neg\varphi$ is true before the execution of WHILE, the program simply skips it and executes the next component. This corresponds to a skip in the else-branch of IF. We can then derive the need for fixed points:

$$\begin{aligned} wp.WHILE.F &\stackrel{!}{=} wp.(\text{if } (\varphi)\{C';WHILE\} \text{ else } \{skip\}).F \\ &\stackrel{!}{=} \varphi \wedge wp.(C';WHILE).F \ \vee\ \neg\varphi \wedge wp.skip.F \\ &\stackrel{!}{=} \varphi \wedge wp.C'.(wp.WHILE.F) \ \vee\ \neg\varphi \wedge F \\ &\stackrel{!}{=} \Phi(wp.WHILE.F) \end{aligned}$$

The result $wp.WHILE.F = \Phi(wp.WHILE.F)$ means that $wp.WHILE.F$ should be a fix point of function Φ by definition.

The question then arises: can we define wp with any fixed point? The answer is no. In fact, Dijkstra and Scholten [5] later also gave definitions for wp and wlp in an equivalent form of least and greatest fixed points. They called it "strongest" and "weakest solution". They also proved that it is necessary to use the extreme solutions. Here, we show the need for least fixed points by verifying that the definition in Table 2.4 coincides with Dijkstra's definition in Equation 2.1. Thanks to domain theory, we have a heuristic to calculate the least fixed point of Φ.

Theorem 2.1 *[lfp heuristics]* [12]

$$\mathrm{lfp}\ \Phi = \sup_{n\in\mathbb{N}} \Phi^n(false)$$

Coincidentally, $H_k(F)$ is the $(k+1)$−th iteration of the characteristic function Φ from the bottom element, denoted by $\Phi^{k+1}(false)$. For all predicates F and all programs C':

Lemma 2.2 *[Correspondance of H and Φ]*

$$\forall k \geqslant 0 : H_k(F) = \Phi^{k+1}(false)$$

Proof. Proof by induction.

BASE CASE:

$$\begin{aligned} \Phi(false) &= (\neg\varphi \wedge F) \vee (\varphi \wedge wp.C'.false) \\ &= (\neg\varphi \wedge F) \vee (\varphi \wedge false) \qquad\qquad | \ (***) \end{aligned}$$

$$
\begin{aligned}
&= \neg\varphi \wedge F && \mid \text{predicate calculus} \\
&= H_0(F)
\end{aligned}
$$

Line (***) is supported by the Law of Excluded Miracle [4, p.18]: for all programs C, $wp.C.false = false$. It states that it is impossible for a program to terminate in a state satisfying no postcondition.

STEP CASE:

$$
\begin{aligned}
H_{k+1}(F) &= wp.IF.H_k(F) \vee H_0(F) \\
&= wp.(\text{if } (\varphi)\{C'\} \text{ else } \{diverge\}).H_k(F) \vee H_0(F) \\
&\qquad\qquad \mid \text{unfold IF} \\
&= (\varphi \wedge wp.C'.H_k(F)) \vee (\neg\varphi \wedge wp.diverge.H_k(F)) \vee H_0(F) \\
&\qquad\qquad \mid \text{definition of wp} \\
&= (\varphi \wedge wp.C'.H_k(F)) \vee (\neg\varphi \wedge false) \vee H_0(F) \\
&\qquad\qquad \mid \text{definition of wp} \\
&= (\varphi \wedge wp.C'.\Phi^{k+1}(false)) \vee H_0(F) \\
&\qquad\qquad \mid \text{induction hypothesis} \\
&= (\varphi \wedge wp.C'.\Phi^{k+1}(false)) \vee (\neg\varphi \wedge F) \\
&= \Phi^{k+2}(false)
\end{aligned}
$$

□

Combining Theorem 2.1 and Equation 2.1, as well as the fact that $\Phi^0(false) = false$, we can arrive at the conclusion that

$$lfp\ \Phi = wp.WHILE.F$$

i.e. the definitions with the least fixed point and Dijkstra's definition are equivalent. And since least fixed points are unique if they exist, it is necessary to use the least fixed point to define wp.

Intuitively, all the fixed points of function Φ can potentially lead to non-termination, but can not guarantee termination. Only the least fixed point can guarantee termination. Remember from Theorem 2.1 that $lfp\ \Phi$ is the disjunction of $\Phi^k(false)$ for all non-negative k, whereas Lemma 2.2 tells us that $H_k(F) = \Phi^{k+1}(false)$.

Recall from Equation 2.1 that $H_k(F)$ is the weakest precondition that the program terminates satisfying F after at most k iterations, so $lfp\ \Phi$ is

WHILE terminates in at most **0** iterations
or WHILE terminates in at most **1** iterations
or WHILE terminates in at most **2** iterations
...

for all $k \in \mathbb{N}$. Now assume that some fixed point p of Φ guarantees termination in k steps, since $lfp\ \Phi$ is the disjunction of all preconditions that guarantee the

termination of WHILE in certain steps, we know that $\mathrm{lfp}\ \Phi \Leftrightarrow \cdots \vee\ p \vee$, hence $p \implies \mathrm{lfp}\ \Phi$. Since $\mathrm{lfp}\ \Phi$ is the least fixed point, there can be no other fixed points that are weaker than the least one. In conclusion, p must be the least fixed point of Φ.

This tells us that all fixed points that are not the least one can potentially lead to non-termination, but only the least fixed point guarantees termination. Since wp is concerned with total correctness, the need for lfp arises.

The advantage of using the least fixed point to define wp is that there are heuristics to find it, whereas Equation 2.1 excels at giving intuitions for the preconditions that guarantee loop termination.

GREATEST FIXED POINT Following from the statement that ***all fixed points that are not the least one can potentially lead to non-termination***, it comes naturally that we need the greatest fixed point when defining the weakest liberal precondition, a precondition that includes all initial states that can potentially lead to non-termination (see Figure 2.3). Since all the fixed points can lead to non-termination, we require that $p \implies wlp.\mathrm{WHILE.F}$, where p is a fixed point of Φ. Following the same reasoning as Equation 2.2, we can conclude that $wlp.\mathrm{C.F}$ is a fixed point of Φ. Hence, by definition, $wlp.\mathrm{WHILE.F} = \mathrm{gfp}\ \Phi$.

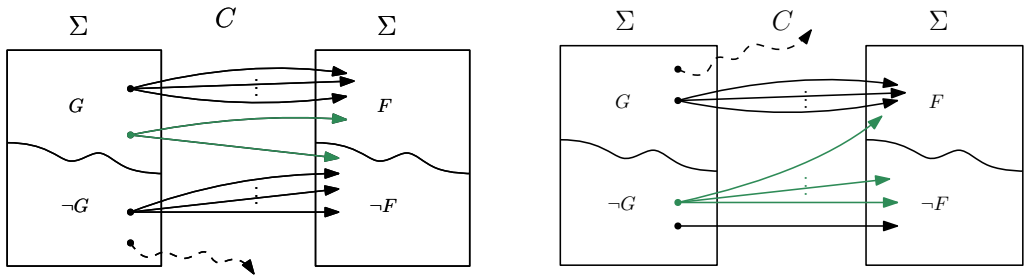
2.4.3 *The Non-deterministic Case: Angelic vs. Demonic*

Now we bring the non-deterministic choice back into the picture and add its wp to Table 2.5. Here we assume a setting with angelic non-determinism, where we assume that whenever non-determinism occurs, it will be resolved in our favor. This results in the weakest precondition for our non-deterministic choice being a disjunction of the wp for its subprograms. We are hopeful that a precondition satisfying the wp of one of the subprograms can also lead to termination in our desired postcondition. This is a design choice that is different from Dijkstra's [3], where the wp for non-deterministic choice is a conjunction, hinting at a demonic setting. Both choices are justifiable, we choose to follow Zhang and Kaminski's work, favoring the resulting Galois connection between the weakest (liberal) precondition transformers and the strongest (liberal) postcondition transformers [25].

Figure 2.3.1 shows wp with non-deterministic programs. Each arrow from left to right shows a possible execution of program C. The effects of demonic and angelic non-determinism is highlighted in green. A condition under whose control the required postcondition is reachable but not guaranteed is considered as a valid precondition in an angelic setting (Figure 2.3.1), but not in a demonic setting (Figure 2.3.2).

C	wp.C.F	wlp.C.F
skip	F	F
diverge	false	true
$x := e$	$F[x/e]$	$F[x/e]$
$C_1; C_2$	$wp.C_1.(wp.C_2.F)$	$wp.C_1.(wp.C_2.F)$
$\{C_1\}\square\{C_2\}$	$wp.C_1.F \vee wp.C_2.F$	$wlp.C_1.F \wedge wlp.C_2.F$
if (φ) $\{C_1\}$ else $\{C_2\}$	$(\varphi \wedge wp.C_1.F) \vee (\neg\varphi \wedge wp.C_2.F)$	$(\varphi \wedge wp.C_1.F) \vee (\neg\varphi \wedge wp.C_2.F)$
while (φ) $\{C'\}$	lfp X. $(\neg\varphi \wedge F) \vee (\varphi \wedge wp.C'.X)$	gfp X. $(\neg\varphi \wedge F) \vee (\varphi \wedge wlp.C'.X)$

Table 2.5: The Weakest (Liberal) Precondition Transformer for Non-deterministic Programs [12]



$G = wp.C.F$ (1) wp (angelic non-determinism)

$G = wlp.C.F$ (2) wlp (demonic non-determinism)

Figure 2.3: Weakest Precondition (Angelic Non-determinism) and Weakest Liberal Precondition (Demonic Non-determinism)

2.5 WEAKEST LIBERAL PRECONDITIONS

While the wp-transformer excludes non-termination, the wlp-transformer takes a more liberal approach. The weakest precondition delivers a precondition so that the program terminates and a state satisfying the postcondition is reachable. The weakest liberal precondition, however, delivers a precondition so that the program either terminates satisfying the postcondition, or diverges. The postcondition in the wlp setting is guaranteed upon termination, because we regard the non-deterministic choice as demonic, again favoring to establish a Galois connection [25].

The definition of the weakest liberal precondition transformer is in Table 2.5. A graphical representation can be found on Figure 2.3.2.

As preluded earlier, greatest fixed points are used to define wlp for while-loops. It is an easy choice, since wlp is semantically the weakest liberal precondition, and $wlp.WHILE.F$ should be a fixed point of its characteristic function, similar to Section 2.4.2.

Theorem 2.3 [12] $\mathrm{gfp}\ \Phi = \sup_{n \in \mathbb{N}} \Phi^n(\mathit{true})$

2.6 STRONGEST POSTCONDITIONS

Following the style to define wp and wlp, Zhang and Kaminski [25] (re-)defined strongest postconditions that capture the characteristics of all reachable states after the execution. In essence, sp.C.G is a postcondition that is satisfied by all states that are reachable from G. The definition of the predicate transformer sp is shown in Table 2.6.

C	**sp.C.G**
skip	G
diverge	*false*
$x := e$	$\exists a.x = e[x/a] \wedge G[x/a]$
$C_1; C_2$	$sp.C_2.(sp.C_1.G)$
$\{C_1\} \square \{C_2\}$	$sp.C_1.G \vee sp.C_2.G$
if $(\varphi)\ \{C_1\}$ *else* $\{C_2\}$	$sp.C_1.(\varphi \wedge G) \vee sp.C_2.(\neg\varphi \wedge G)$
while $(\varphi)\ \{C'\}$	$\neg\varphi \wedge \mathrm{lfp}\ X.G \vee sp.C.(\varphi \wedge X)$

Table 2.6: The Strongest Postcondition Transformer [25]

We can also illustrate the behavior of a program controlled by sp in Figure 2.4. Instead of discussing termination starting from a precondition, sp focuses on reachability of states satisfying postconditions. The dotted arrow points to post-conditions describing unreachable final states after the execution of C. For example, no state would satisfy $x = 2$ after the execution of $x := 1$.

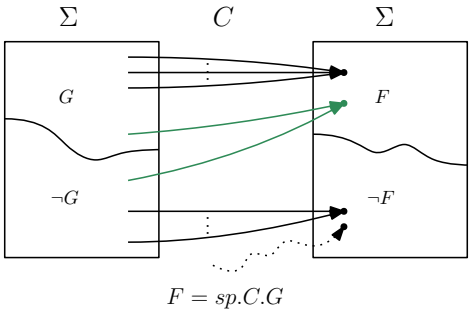


Figure 2.4: Strongest Postcondition (Angelic Non-determinism)

$$\frac{}{\sigma \xrightarrow{skip} \sigma}\,skip \qquad \frac{}{\sigma \xrightarrow{x:=e} \sigma(x := \sigma.e)}\,assign$$

$$\frac{\sigma \xrightarrow{C_1} \mu,\ \mu \xrightarrow{C_2} \tau}{\sigma \xrightarrow{C_1;C_2} \tau}\,seq \qquad \frac{\sigma \xrightarrow{C_i} \tau,\ i \in \{1,2\}}{\sigma \xrightarrow{C_1 \square C_2} \tau}\,choice_i$$

$$\frac{\sigma \in \varphi,\ \sigma \xrightarrow{C_1} \tau}{\sigma \xrightarrow{IF} \tau}\,if_1 \qquad \frac{\sigma \notin \varphi,\ \sigma \xrightarrow{C_2} \tau}{\sigma \xrightarrow{IF} \tau}\,if_0$$

$$\frac{\sigma \notin \varphi}{\sigma \xrightarrow{WHILE} \sigma}\,while_0 \qquad \frac{\sigma \in \varphi,\ \sigma \xrightarrow{C} \mu,\ \mu \xrightarrow{WHILE} \tau}{\sigma \xrightarrow{WHILE} \tau}\,while_n$$

$$\frac{\bot \notin \Sigma}{\sigma \xrightarrow{diverge} \bot}\,diverge$$

where IF=if (φ) $\{C_1\}$ else $\{C_2\}$, WHILE=while (φ) do C, and $\sigma, \tau, \mu \in \Sigma$.

Table 2.7: Big Step Semantics

2.7 BIG STEP SEMANTICS

To express the meaning of programs, we choose big-step semantics to describe executions of a program. Taking inspiration from Nipkow and Klein's book [20] we define the big-step semantics in Table 2.7.

Note that in rule assign, we have a formula $\sigma(x := \sigma.e)$. Here we overload the symbol for function application $.$ so that it applies to expressions as well, but without specifying the set of expressions hence restricting our programming language. An expression e would be evaluated in a usual way, e.g. $x + y$ at state σ would evaluate to $\sigma.x + \sigma.y$.

we also use symbols $\sigma(x := v)$ to denote a state where the value of variable x is v, and all other variables have the same values as in σ. In our big-step semantics, non-termination of C with initial state σ is denoted by the nonexistence of state $\tau \in \Sigma$ such that $\sigma \xrightarrow{C} \tau$. Either *diverge* is executed during the execution of C, in which a special state $\bot \notin \Sigma$ is reached, or the execution of C never ends, for example $C = while(true)doskip$. With the definition of big-step semantics, we can precisely express the soundness of our predicate transformers in the following section.

2.8 SOUNDNESS

Theorem 2.4 *[Soundness of wp]* [25]

$$wp.C.F = \{\sigma \in \Sigma \mid \exists \tau \in \Sigma :\ \sigma \xrightarrow{C} \tau \ \wedge\ \tau \vDash F\}$$ [5]

Theorem 2.5 *[Soundness of wlp]* [5]

$$wlp.C.F = \{\sigma \in \Sigma \mid \forall \tau \in \Sigma :\ \sigma \xrightarrow{C} \tau \implies \tau \vDash F\}$$

5 $\exists \tau \in \Sigma : P$ is short for $\exists \tau . \tau \in \Sigma : P$

Theorem 2.6 *[Soundness of sp]* [23, 25]

$$sp.C.G = \{\tau \in \Sigma \mid \exists \sigma \in \Sigma : \ \sigma \xrightarrow{C} \tau \ \wedge \ \sigma \vDash G\}$$

2.9 PROPERTIES OF WP AND WLP

Theorem 2.7 *[Conjugates] wp and wlp are each other's conjugates* [25]*:*

$$\forall C \in \mathcal{C} : \ \forall F \in \mathcal{P} : \ wp.C.F = \neg wlp.C.\neg F$$

Theorem 2.8 *[Monotonicity]* [5] *For any* $C \in \mathcal{C}$*,* $wlp.C$ *is monotonic::*

$$\forall X, Y \in \mathcal{P} : \ X \Longrightarrow Y \implies wlp.C.X \implies Y$$

Theorem 2.9 [5] $\forall C \in \mathcal{C} : \ wlp.C.true = true$

Part II

NECESSARY LIBERAL PRECONDITIONS

In this part, I discuss the possible scenarios of the inspected implication, identify a distinctive case to study before proceeding with the general setting.

3

NECESSARY LIBERAL PRECONDITIONS

We are interested in studying the necessary liberal precondition, a weakening of the weakest liberal precondition:

$$wlp.C.F \implies G$$

We show our interest, by relating all the pre- and postcondition transformers, so we can distinguish the existing triples that are already well-researched graphically.

Luckily, we can find scenarios where the overapproximation can be useful, and demonstrate it with an example, before giving a proof system. We conclude that this overapproximation can help us find more preconditions that can lead to failure, either by including states that lead to unidentified errors, or by including states that can lead to errors via non-terministic choice. The latter method then leads to finding conditions that are able to capture these special states. The side effect of this result is that we can now find the wlp transformer with angelic non-determinism using wlp with demonic non-determinism and sp.

This chapter includes our most important results:

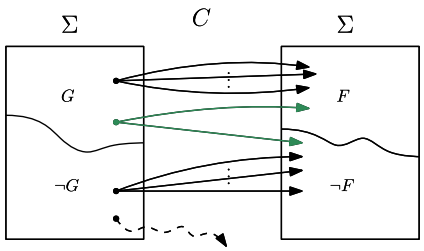
1. The predicate transformers are all related together nicely by relations like termination, reachability, conjunction, over- and under-approximation, and implication in both directions.

2. Without any additional constraints, the necessary liberal preconditions can contain anything. The only thing we are sure of is that its negation guarantees termination in the negation of the desired postcondition.

3. Necessary liberal preconditions find their usefulness in avoiding bugs.

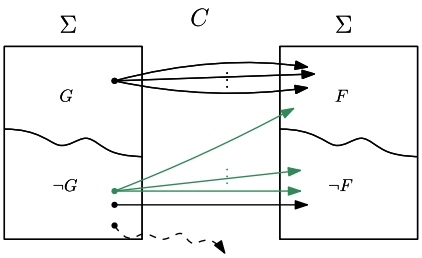
4. With necessary liberal precondition, we also can overapproximate and underapproximate wlp with angelic non-determinism (wlp_a). In the end, we can qualify wlp_a using necessary liberal precondition and strongest postcondition transformers, without having to define wlp_a.

In the upcoming sections, we first give an overview of wp, wlp, sp, slp with both angelic and demonic non-determinism, then discuss the general semantics of the necessary liberal precondition. Finally, we focus on a characteristic scenario and show properties that captures it.



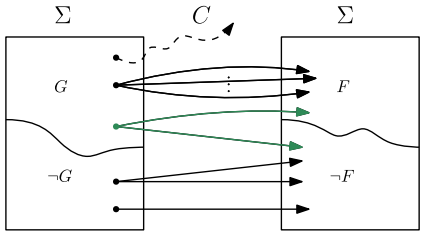
$G = wp_a.C.F$

(1) Weakest precondition (angelic non-determinism)



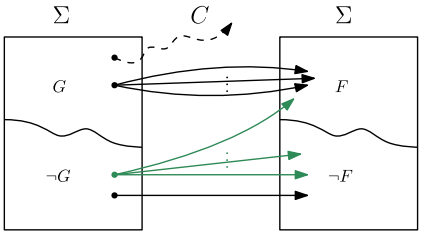
$G = wp_d.C.F$

(2) Weakest precondition (demonic non-determinism)



$G = wlp_a.C.F$

(3) Weakest liberal precondition (angelic non-determinism)



$G = wlp_d.C.F$

(4) Weakest liberal precondition (demonic non-determinism)

Figure 3.1: wp, wlp with Angelic and Demonic Non-determinism

3.1 PRECONDITIONS AND POSTCONDITIONS

3.1.1 *wp and Related*

Figure 3.1 shows the impact of the interpretation of non-determinism on the semantics of the wp and wlp transformers. The subscripts $_a$ and $_d$ correspond to angelic and demonic, respectively.

With angelic non-determinism, we consider the non-deterministic choice to resolve in our favor, which means that if one of the possible paths from this non-deterministic choice can lead to the desired postcondition, we regard the precondition as a valid precondition. Graphically, the green arrows in Figure 3.1.1 and Figure 3.1.3 are included in the weakest (liberal) precondition when we regard the non-determinism as angelic. Formally, the decision to look at the non-deterministic choice as angelic is reflected by the use of disjoints in its definition, for example,

$$wp.C.(\{C_1\}\square\{C_2\}) = wp.C_1.F \vee wp.C_2.F$$

in Table 2.5.

On the other hand, if the non-deterministic choice is regarded as demonic, we are pessimistic over the result of the non-deterministic choice: we think it will not resolve in our favor, hence we are only confident to include a condition into the valid precondition set, if all possible paths end satisfying our desired postcondition upon termination. Graphically, the green arrows are not included

in wp and wlp in Figure 3.1.2 and Figure 3.8.1. The definition is then using conjuncts instead of disjuncts, like

$$wlp.C.(\{C_1\}\square\{C_2\}) = wlp.C_1.F \wedge wlp.C_2.F$$

in Table 2.5.

The choice is somewhat artificial, depending on what properties the author wishes for their transformers. For example, Dijkstra and Scholten choose demonic non-determinism for wp and wlp transformers, acquiring a property by definition [5]:

$$wp_d.C.F = wlp_d.C.F \wedge wp_d.C.true$$

meaning that by finding the weakest liberal precondition and proving termination, one conveniently finds the weakest precondition of the same program and postcondition. Graphically, one needs only to re-place the dashed arrows indicating non-termination to get from Figure 3.1.1 to Figure 3.1.3 and vice versa, or from Figure 3.1.2 to Figure 3.8.1 and vice versa.

This is especially useful when trying to underapproximate the weakest precondition, or in other words, finding valid Hoare Triple, considering that $G\,\{C\}\,F$ is a valid Hoare Triple (with regard to total correctness, meaning that we supplement the original axioms [9] with rules to prove termination [17]) if and only if $G \implies wp.C.F$.

Conversely, if one were to choose different views on non-determinism with wp and wlp transformers, we end up with a different property that wp and wlp are each other's conjugates [4, 25]:

$$wp_a.C.F = \neg wlp_d.C.\neg F$$

this indicates a duality between wp_a and wlp_d in the sense that by defining one transformer, we can arrive at the other by negation. Graphically, we can "flip" Figure 3.1.2 to arrive at Figure 3.1.3 and vice versa, or "flip" Figure 3.1.1 to arrive at Figure 3.8.1 and vice versa, like in Figure 3.2.

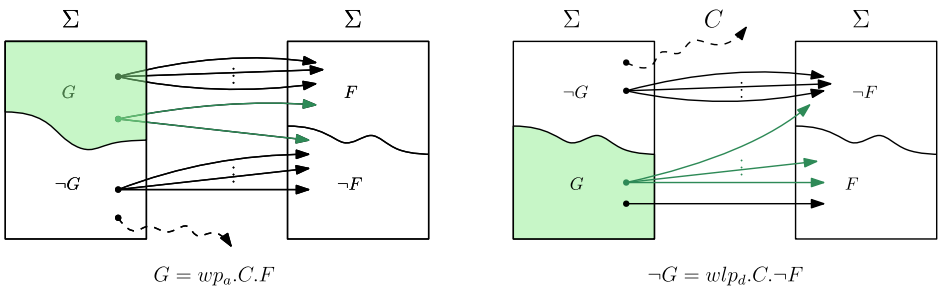


Figure 3.2: wp_a and wlp_d Are Conjugates

These relations can be summed up as in Figure 3.3. The choice whether to resolve the non-determinism angelically or demonically for wp and wlp depends then on which of the relations the author prefers.

In this thesis, we prefer wp with angelic non-determinism and wlp with demonic non-determinism, because this choice gifts us Galois connection between

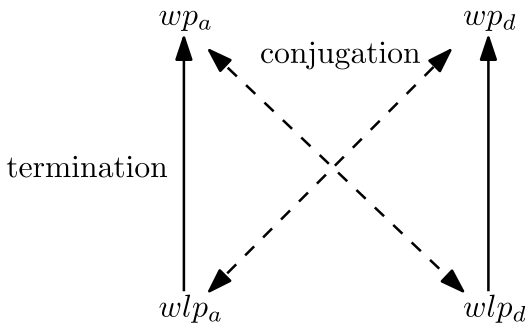


Figure 3.3: Relations in wp-Family

the precondition transformers and the postcondition transformers [25], which we will illustrate in the next section:

Theorem 3.10 *[Galois connection between wp and slp]*

$$wp_a.C.F \implies G \textit{ if and only if } F \implies slp_d.C.G$$

Theorem 3.11 *[Galois connection between wlp and sp]*

$$G \implies wlp_d.C.F \textit{ if and only if } sp_a.C.G \implies F$$

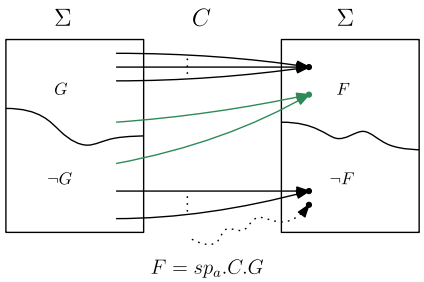
3.1.2 *sp and Related*

The strongest postcondition was originally discussed as a side-product of the wp and wlp transformers by Dijkstra and Scholten [5]. sp.S.Y is meant to describe "those final states for which there exists a computation under control of S that belongs to the class *initially* Y", but it has found more important utilities. For example, de Vries and Koutavas [23] use sp to reason with randomized algorithms, although they called it weakest postconditions. O'Hearn [22] uses sp to define incorrectness logic by underapproximating it, building the theoretical foundation to a new method of identifying errors in programs.

Instead of questioning about termination and final states upon termination, like the precondition transformers, the postcondition transformers question about reachability and initial states that can reach the desired postconditions. The choice of angelic and demonic non-determinism also plays a rule while determining the semantics of the postcondition transformers, as shown in Figure 3.4.

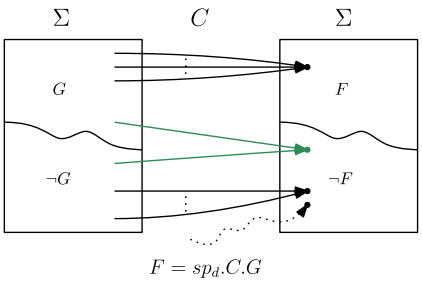
Note that while sp is well-researched, slp is less investigated to the best of our knowledge. For example, Wulandari and Plump [24] defined and proved soundness for slp, but only for loop-free programs. While Li et al. [15] proposed the definition of slp including loops, but did not include proof for soundness and completeness. Even so, the semantics of slp is rather undisputed: slp should be sp joint with all unreachable postconditions.

Following the same principles as the ones for precondition transformers, we can deduce a similar relation between the postcondition transformers with different resolution for non-determinism, as seen in Figure 3.5. The main difference is that the arrows pointing up are not labelled with termination, but reachability.



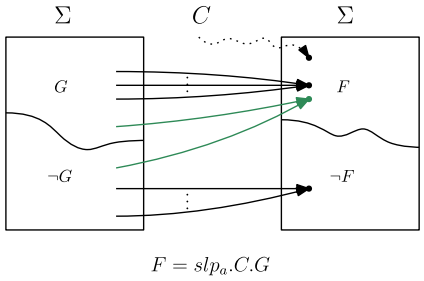
$F = sp_a.C.G$

(1) Strongest postcondition (angelic non-determinism)



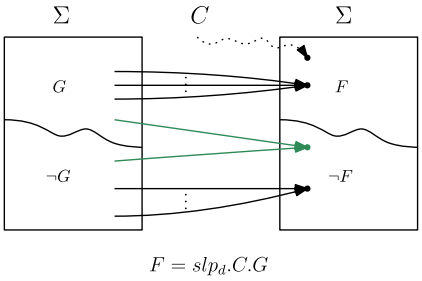
$F = sp_d.C.G$

(2) Strongest postcondition (demonic non-determinism)



$F = slp_a.C.G$

(3) Strongest liberal postcondition (angelic non-determinism)



$F = slp_d.C.G$

(4) Strongest liberal postcondition (demonic non-determinism)

Figure 3.4: sp, slp with Angelic and Demonic Non-determinism

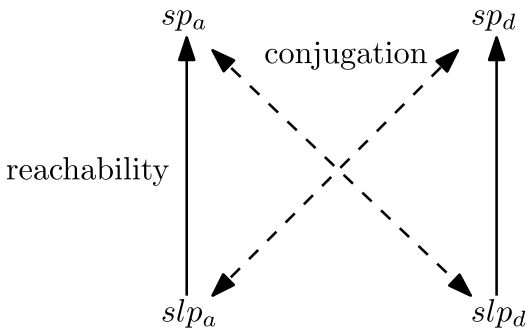


Figure 3.5: Relations in sp-Family

3.2 LITERATURE REVIEW: ALL ABOUT TRIPLES

Supplementing the Galois connection specified in Theorem 3.10 and Theorem 3.11, we arrive at Figure 3.6. It is now obvious that we chose to resolve non-determinism exactly like the colored ones in Figure 3.6, so that we can conveniently connect the pre- and postcondition transformers.

The advantage of the design choice of wp and sp with angelic non-determinism as well as wlp and slp with demonic non-determinism shows itself now. By chosing the formalism that are colored in Figure 3.6, all the under- and overapproximations over the predicate transformers are related well together, as we depicte in Figure 3.7.

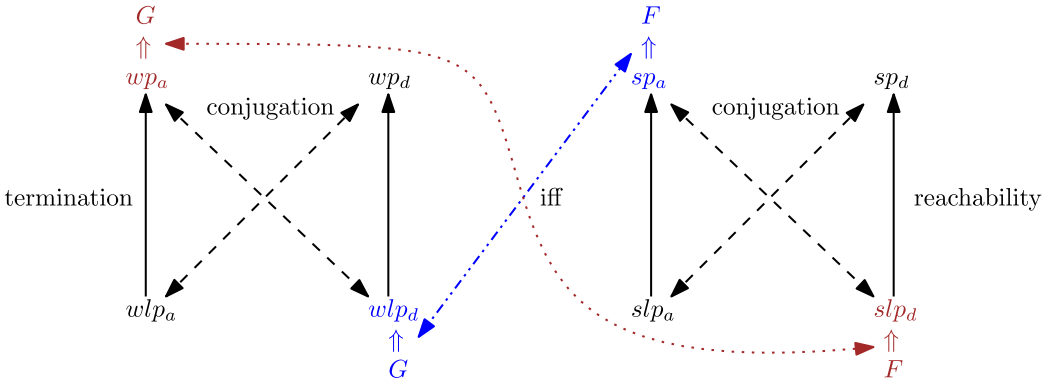


Figure 3.6: Connecting wp-Family and sp-Family

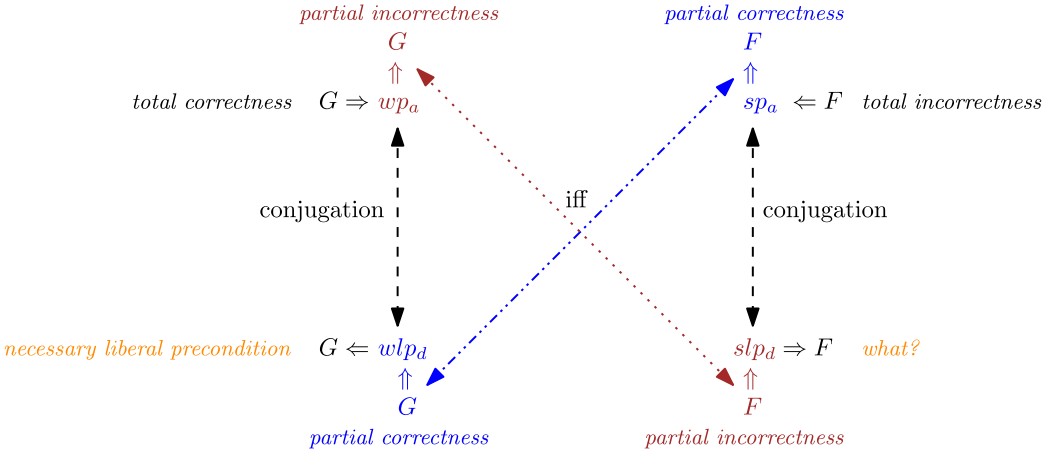


Figure 3.7: Connecting Overapproximations and Underapproximations

The implications noted in black correspond to total correctness [17] $G \implies wp_a.C.F$ and total incorrectness $F \implies sp_a.C.G$ [22] triples, respectively. The blue ones are the partial correctness triples $G \implies wlp_d.C.F$ [9] as well as $sp_a.C.G \implies F$ via Galois connection. The red triples are the partial correctness triples $F \implies slp_c.C.G$ [25] iff $wp_a.C.F \implies G$ that still require further investigation. The orange triple on the left is the triple found by the necessary liberal precondition and the center of our interest in this thesis, whereas the orange triple on the right is still an unanswered question as far as our knowledge.

3.3 THE GENERAL CASE

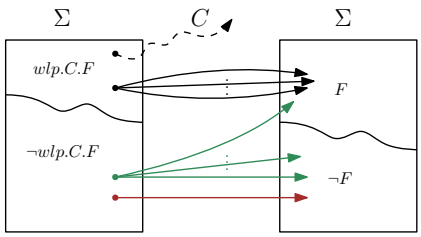
In our overapproximation, the weaker G can contain various preconditions: on the one hand, G can be so general that it is satisfied by any program state; on the other hand, a G that is barely weaker than $wlp.C.F$ is also not much different from the latter. Alternatively, G can also contain all kinds of preconditions that starting from it, any postcondition is reachable. One thing we are certain about, though, is that a program with an original state satisfying $\neg G$ will terminate, and the final state can satisfy $\neg F$:

$$wlp.C.F \implies G = \neg G \implies \neg wlp.C.F$$

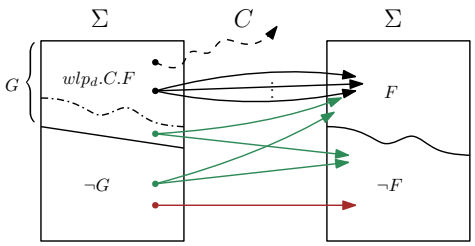
$$= \neg G \implies wp.C.\neg F \qquad \text{| Theorem 2.7}$$

In Section 2.5 we define the weakest liberal precondition and state that it characterizes all the preconditions under whose control the program either diverges or will terminate in a state satisfying F. We are certain to use "will" instead of "can", the non-deterministic choice is viewed as demonic, so the behavior of wlp can be depicted by Figure 3.8.1. We can categorize the executions of the program in four ways:

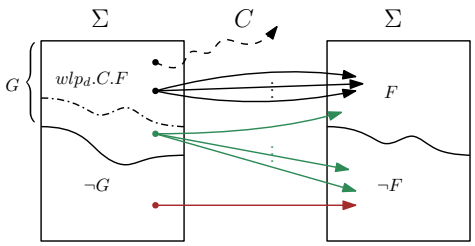
1. the dashed arrow means non-terminating executions;
2. the black arrows are executions starting from an initial state satisfying $wlp.C.F$ and only terminating in final states satisfying F;
3. the green arrows are the executions starting from an initial state satisfying $\neg wlp.C.F$ but can terminate in states either satisfying F or satisfying $\neg F$;
4. the red arrow represents executions starting from an initial state satisfying $\neg wlp.C.F$ and only terminating in final states satisfying $\neg F$.



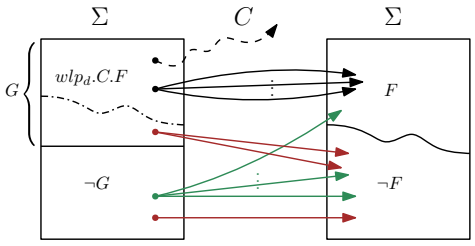
(1) Weakest liberal precondition (demonic non-determinism)



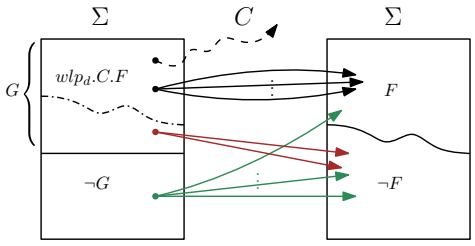
(2) Precondition G with $wlp.C.F \implies G$ and G contains some green arrows



(3) Precondition G with $wlp.C.F \implies G$ and G contains all the green arrows

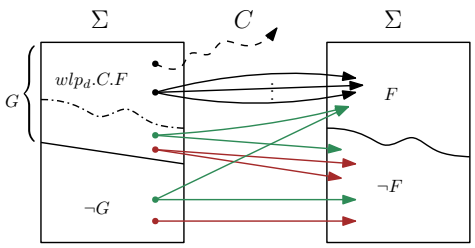


(4) Precondition G with $wlp.C.F \implies G$ and G contains some red arrows

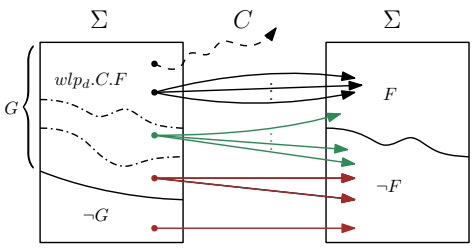


(5) Precondition G with $wlp.C.F \implies G$ and G contains all the red arrows

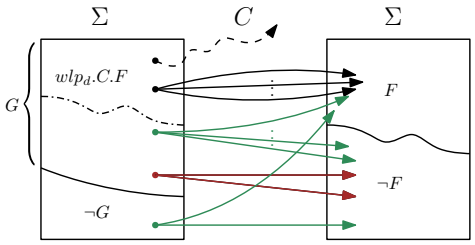
Figure 3.8: Case Distinction of Preconditions Weaker Than wlp



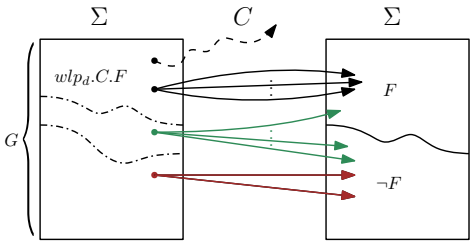
(6) Precondition G with $wlp.C.F \implies G$ and G contains some green arrows and some red arrows



(7) Precondition G with $wlp.C.F \implies G$ and G contains all the green arrows and some red arrows



(8) Precondition G with $wlp.C.F \implies G$ and G contains some green arrows and all the red arrows



(9) Precondition G with $wlp.C.F \implies G$ and G contains all the arrows

Figure 3.8: Case Distinction of Preconditions Weaker Than wlp (Cont.)

If we were to weaken the precondition, it can happen in various ways as shown in Figure 3.8.2-9.

In general, when $wlp.C.F \implies G$, G can contain all possible initial states, which can be the starting points of black, green, red, or dashed arrows, representing executions terminating in states satisfying F, F or ¬F, ¬F, or non-terminating, respectively. As a result, we can not make many statements without adding extra restrictions to G. However, we can see from Figure 3.8 that ¬G does not contain any black or dashed arrows in all cases. In other words, if program C starts in any initial state satisfying ¬G, then either G is empty, or

- its executions terminate, and
- there exists an execution that ends up in a final state that satisfies ¬F.

This corresponds to the semantics of the wp transformer as shown in Theorem 2.4, hinting that ¬G {C} ¬F is a valid Hoare triple. The question then naturally arises: why do we concern ourselves with G, if we can just prove our specifications using wp or Hoare triples? To demonstrate the answer, we analyze the example written in Listing 3.1.

```
1        ... // starting device
2        n := n + 1 □ n := n - 1 □ diverge;
3        ... // retirement
```

Listing 3.1: Door with Sensors Counting Number of People Present

In the author's student life, there was a door at university that she always finds interesting. The door is located at a lecture hall, equipped with two sensors on the left and right sides of the door frame. Each time a person walks in or out, the sensor registers and adds or decreases the number of people in the lecture hall, making a small tick sound. It seems that this door serves the purpose of helping security guards keep an eye on the lecture rooms after closing time without having to be physically there.

But can the door really distinguish from a person entering or leaving the room? And what if naughty students try to trick the door by using a backpack to pretend to exit the door multiple times? What if one of the sensors break? We can this simple scenario by putting the three parts in non-deterministic choice. It can happen that a student enters the room hence increasing the count of the number of people present: $n := n + 1$; or a person exits and decreasing the number: $n := n - 1$. Alternatively, the sensor can break because of old age and result in unforeseeable behavior, for example always detecting someone entering forever.

We know that there is definitely something wrong, in case the security sees on their device that the number of people in a lecture hall is negative. If we want to know what caused it, we can calculate the weakest liberal precondition of the program snippet with regard to $F = \{\sigma \in \Sigma \mid \sigma.n < 0\}$ in Listing 3.1. We write $F = \{n < 0\}$ in short:

```
1  ... // starting device
2  {n < −1}
3  {n < −1} ∧ {n < 1} ∧ true
4  n := n + 1 □ n := n - 1 □ diverge;
5  {n < 0}
6  ... // retirement
```

Listing 3.2: Weakest Liberal Precondition w.r.t Postcondition $F = \{\sigma \in \Sigma \mid \sigma.n < 0\}$

It seems like we should avoid starting the system in a state where n is less than -1. But it is enough? If we start the system in a state where $n = -1$, or even $n = 0$, we can still reach an erroneous state where the value of n is negative. Figure 3.9 We can see that with wlp, it is not enough to identify all preconditions

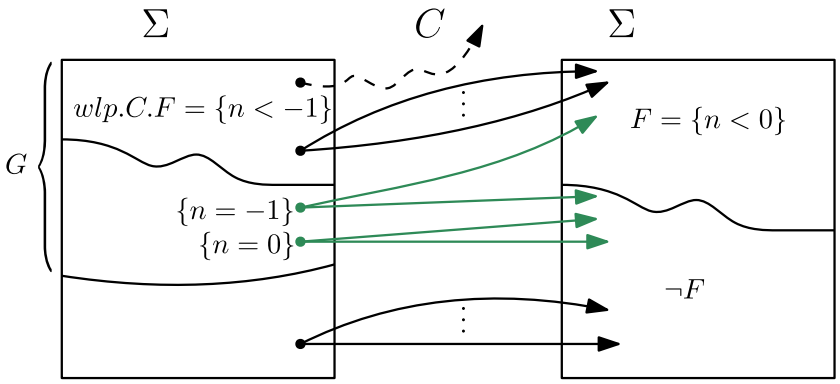


Figure 3.9: $wlp.C.F = \{n < -1\}$

that can result in errors. In other words, the green arrows in Figure 3.9 are not

recognized. But by overapproximating wlp to include preconditions like $\{n = -1\}$ and $\{n = -1\}$, we can also include all the green arrows in G.

This is, however, not the only thing that is missing. Obviously, a lecture hall does not have unlimited capacity. Once the system is showing a number higher than the maximum capacity, the security would also instantly know that something is wrong. Also, there could be conditions like "during holiday time, the lecture halls are unoccupied" that an expert of the system (the security guards in this case) would know, but not in general. So even by including all the green arrows via overapproximation, we are still missing some of the red arrows like in Figure 3.10, the hidden knowledge that seasoned users possess.

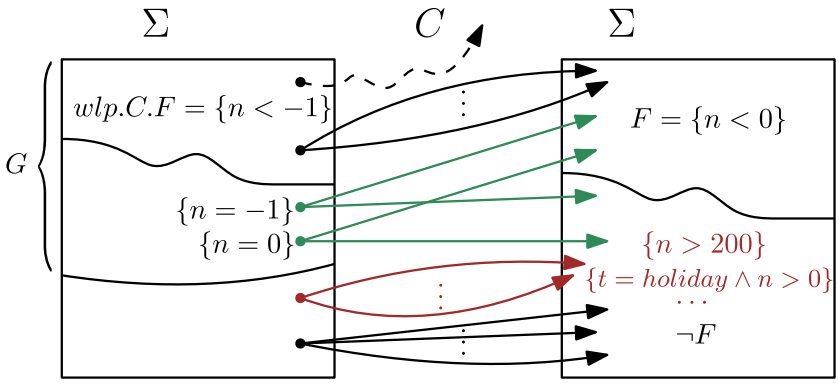


Figure 3.10: wlp.C.F $\implies$ G

This demonstrates overapproximating wlp is useful to include the green and red arrows in Figure 3.10, aka when:

- some preconditions can but are not guaranteed to lead to the erroneous final states non-deterministically;
- or the user only have insufficient knowledge of the system, but experts can deduce other erroneous final states that are unknown to the public.

To utilize this overapproximation, we either identify the preconditions that result in the green arrows, which we will do in the next section, or we first calculate the weakest liberal precondition of the erroneous postcondition that we know of, then relax this precondition by "guessing" similar erroneous postconditions with the extra knowledge from experts over the system.

3.3.1 *Proof System*

We provide the proof system in Table 3.1. The rules are mostly vanilla, except from the consequence rule:

$$\frac{P \implies G,\ \langle P\rangle\ C\ \langle Q\rangle,\ F \implies Q}{\langle G\rangle\ C\ \langle F\rangle}\text{conseq}$$

$$\frac{}{\langle F\rangle\ \mathrm{skip}\ \langle F\rangle}\mathrm{skip} \qquad \frac{}{\langle \mathrm{true}\rangle\ \mathrm{diverge}\ \langle F\rangle}\mathrm{div}$$

$$\frac{}{\langle F[x/e]\rangle\ x := e\ \langle F\rangle}\mathrm{assign} \qquad \frac{\langle G\rangle\ C_1\ \langle P\rangle,\ \langle P\rangle\ C_2\ \langle F\rangle}{\langle G\rangle\ C_1; C_2\ \langle F\rangle}\mathrm{seq}$$

$$\frac{\langle \varphi \wedge G\rangle\ C_1\ \langle F\rangle,\ \langle \neg\varphi \wedge G\rangle\ C_2\ \langle F\rangle}{\langle G\rangle\ \mathrm{IF}\ \langle F\rangle}\mathrm{if} \qquad \frac{\langle \varphi \wedge F\rangle\ C'\ \langle F\rangle}{\langle F\rangle\ \mathrm{WHILE}\ \langle \neg\varphi \wedge F\rangle}\mathrm{while}$$

$$\frac{\langle G\rangle\ C_1\ \langle F\rangle,\ \langle G\rangle\ C_2\ \langle F\rangle}{\langle G\rangle\ C_1 \square C_2\ \langle F\rangle}\mathrm{choice} \qquad \frac{P \Longrightarrow G,\ \langle P\rangle\ C\ \langle Q\rangle,\ F \Longrightarrow Q}{\langle G\rangle\ C\ \langle F\rangle}\mathrm{conseq}$$

where $\mathrm{IF} = \mathrm{if}\ (\varphi)\ \{C_1\}\ \mathrm{else}\ \{C_2\}$, $\mathrm{WHILE} = \mathrm{while}\ (\varphi)\ \mathrm{do}\ C$.

Table 3.1: The Proof System

It is the complete opposite direction from the consequence rule in Hoare logic that we referred to in Table 2.2. Rewriting it in the same natural deduction style, we arrive at:

$$\frac{G \Longrightarrow P,\ P\ \{C\}\ Q,\ Q \Longrightarrow F}{G\ \{C\}\ F}\mathrm{conseq}_h$$

This corresponds to our expectation, because we know that the overapproximation triple is exactly a Hoare triple (with respect to partial correctness) with negated pre- and postconditions:

$$\begin{aligned} & wlp.C.F \Longrightarrow G \\ \Leftrightarrow\ & \neg G \Longrightarrow \neg wlp.C.F \\ \Leftrightarrow\ & \neg G \Longrightarrow wp.C.\neg F && \text{| Theorem 2.7} \\ \Rightarrow\ & \neg G\ \{C\}\ \neg F && (*) \end{aligned}$$

Note that the last line is only an implication in one direction, because we take the version of Hoare triple regarding partial correctness, meaning that termination is not guaranteed. In other words, $\neg G\ \{C\}\ \neg F$ can be a valid Hoare triple, but an initial state satisfying $\neg G$ does not necessary guarantee termination, hence $\neg G$ is not necessarily a subset of $wp.C.\neg F$. If we were to take Hoare triple with additional rules to prove termination [17], the implication would be in both directions.

From conseq_h we can then deduce the rule conseq assuming that the triple $\langle\cdot\rangle\ \cdot\ \langle\cdot\rangle$ is sound (which we will prove later in this section), i.e. $\langle P\rangle\ C\ \langle Q\rangle \Longrightarrow (wlp.C.F \Longrightarrow G)$:

$$\cfrac{\cfrac{\cfrac{\cfrac{P \Longrightarrow G,\ \langle P\rangle\ C\ \langle Q\rangle,\ F \Longrightarrow Q}{P \Longrightarrow G,\ wlp.C.F \Longrightarrow G,\ F \Longrightarrow Q}\ \text{soundness}}{P \Longrightarrow G,\ \neg G\ \{C\}\ \neg F,\ F \Longrightarrow Q}\ (*)}{\neg G \Longrightarrow \neg P,\ \neg P\ \{C\}\ \neg Q,\ \neg Q \Longrightarrow \neg F}\ \text{first-order logic}}{\neg G\ \{C\}\ \neg F}\ \mathrm{conseq}_h$$

Lemma 3.12 *[The proof system is sound]*

$$\langle G \rangle\ C\ \langle F \rangle \implies (wlp.C.F \implies G)$$

Proof. We prove by induction and case distinction over the lowest level of the proof tree.

CASES `skip`, `div`, `assign` Straightforward.

CASE `seq` Assume we proved the triple $\langle G \rangle\ C_1; C_2\ \langle F \rangle$ with the following proof tree:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\cdots}}{\cdots\cdots}}{\vdots}}{\dfrac{\langle G \rangle\ C_1\ \langle P \rangle,\ \langle P \rangle\ C_2\ \langle F \rangle}{\langle G \rangle\ C_1; C_2\ \langle F \rangle}\ seq}$$

Then we have proven $M_1 := \langle G \rangle\ C_1\ \langle P \rangle$ and $M_2 := \langle P \rangle\ C_2\ \langle F \rangle$ as the premises of the last deduction, as well as $M_3 := \langle G \rangle\ C_1; C_2\ \langle F \rangle$ as the conclusion of the last deduction step. We also have the induction hypotheses

$$H_1 := \langle G \rangle\ C_1\ \langle P \rangle \implies (wlp.C_1.P \implies G)$$

$$H_2 := \langle P \rangle\ C_2\ \langle F \rangle \implies (wlp.C_2.F \implies P)$$

Our goal is to prove $\langle G \rangle\ C_1; C_2\ \langle F \rangle \implies (wlp.(C_1; C_2).F \implies G)$. By discharging M_1 in H_1, M_2 in H_2, and M_3 in the goal, we acquire premises $wlp.C_1.P \implies G$ and $wlp.C_2.F \implies P$ to prove goal $wlp.(C_1; C_2).F \implies G$. This is done by discharging of the definition of wlp with composition and the monotonicity of wlp:

$$\begin{aligned}
& wlp.(C_1; C_2).F = wlp.C_1.(wlp.C_2.F) && \mid \text{definition of wlp} \\
\Rightarrow\ & wlp.C_1.P && \mid \text{monotonicity of wlp} \\
\Rightarrow\ & G && \mid \text{premise}
\end{aligned}$$

CASE `if` Unrolling the definition of $wlp.IF.F$ we get that

$$wlp.IF.F = \varphi \wedge wlp.C_1.F \ \vee\ \neg\varphi \wedge wlp.C_2.F$$

Similar as before, by discharging induction hypotheses and premises, we can acquire the following conditions:

$$wlp.C_1.F \implies \varphi \wedge G \text{ and } wlp.C_2.F \implies \neg\varphi \wedge G$$

From the first condition we get:

$$\varphi \wedge wlp.C_1.F \implies wlp.C_1.F \implies \varphi \wedge G \implies G$$

and

$$\neg\varphi \wedge wlp.C_2.F \implies wlp.C_2.F \implies \neg\varphi \wedge G \implies G$$

Hence $wlp.IF.F \implies G$.

CASE *while* how to write this rule so sound?

CASE *choice* Similar to the case with *seq*, we can prove that $wlp.(C_1 \square C_2).F \implies G$ from

$$wlp.(C_1 \square C_2).F = wlp.C_1.F \wedge wlp.C_2.F$$

and

$$wlp.C_1.F \implies G \wedge wlp.C_2.F \implies G$$

CASE *conseq* This case is also not different from before: from the induction hypotheses and premises we can derive $wlp.C.Q \implies P$. Together with $P \implies G$, $F \implies Q$ and the monotonicity of $wlp.C$ we conclude that

$$wlp.C.F \implies wlp.C.Q \implies P \implies G$$

□

3.4 A SPECIAL CASE

As promised before, we will look into a way to identify the green arrows in Figure 3.9, aka the executions that start from initial states but may end non-deterministically in F or ¬F. We can easily see that if we overapproximate wlp by adding all the green arrows, we end up with Figure 3.8.3, which is exactly wlp with angelic non-determinism, as shown in Figure 3.1 at the beginning of this chapter.

When G corresponds to Figure 3.8.3, we know that under its control, the program always can reach a final state satisfying F if it terminates, while with an initial state satisfying ¬G, the program is will terminate satisfying ¬F

Dual to the semantics of wp and wlp as shown in Theorem 2.4 and Theorem 2.5, we can deduce the semantics of wlp with angelic non-determinism (denoted by wlp_a) recalling the representation for non-termination mentioned in Section 2.7:

Statement 3.13 *[Semantics of wlp_a]*

$$wlp_a.C.F = \{\sigma \in \Sigma \mid \neg(\exists \tau \in \Sigma : \sigma \xrightarrow{C} \tau) \ \vee \ (\exists \tau \in \Sigma : \sigma \xrightarrow{C} \tau \wedge \tau \vDash F)\}$$

Luckily, we can find statements using wlp and sp that captures this specific G, hence giving us a way to express wlp_a without having to define it:

Lemma 3.14 *[Angelic wlp implies G]*

if $(wlp.C.F \implies G) \wedge (sp.C.\neg G \implies \neg F)$ *then* $wlp_a.C.F \implies G$

The second prerequisite $sp.C.\neg G \implies \neg F$ states that from ¬G we are only allowed to reach ¬F, making sure that all green arrows as in Figure 3.8 are included in G.

Proof. The assumption expresses that for any state $\sigma \in \Sigma$:

$$
\begin{aligned}
wlp.C.F \implies G \;&\Leftrightarrow \sigma \in wlp.C.F \implies \sigma \in G \\
&\Leftrightarrow (\forall \tau \in \Sigma :\ \sigma \xrightarrow{C} \tau \implies \tau \in F) \implies \sigma \in G \\
&\qquad\qquad \text{| Theorem 2.5} \\
&\Leftrightarrow (\forall \tau \in \Sigma :\ \neg(\sigma \xrightarrow{C} \tau) \vee \tau \in F) \implies \sigma \in G \\
&\Leftrightarrow \neg(\exists \tau \in \Sigma : (\sigma \xrightarrow{C} \tau) \wedge \neg(\tau \in F)) \implies \sigma \in G \qquad \text{(a)}
\end{aligned}
$$

Also, for any state $\tau \in \Sigma$:

$$
\begin{aligned}
sp.C.\neg G \implies \neg F \;&\Leftrightarrow \tau \in sp.C.\neg G \implies \tau \in \neg F \\
&\Leftrightarrow (\exists \mu \in \Sigma :\ \mu \xrightarrow{C} \tau \wedge \mu \in \neg G) \implies \tau \in \neg F \\
&\qquad\qquad \text{| Theorem 2.6} \\
&\Leftrightarrow \neg(\tau \in \neg F) \implies \neg(\exists \mu \in \Sigma :\ \mu \xrightarrow{C} \tau \wedge \mu \in \neg G) \\
&\Leftrightarrow \tau \in F \implies \forall \mu \in \Sigma :\ \neg(\mu \xrightarrow{C} \tau \wedge \mu \in \neg G) \\
&\Leftrightarrow \tau \in F \implies \forall \mu \in \Sigma :\ \neg(\mu \xrightarrow{C} \tau) \vee \neg(\mu \in \neg G) \\
&\Leftrightarrow \tau \in F \implies \forall \mu \in \Sigma :\ \neg(\mu \xrightarrow{C} \tau) \vee \mu \in G \qquad \text{(b)}
\end{aligned}
$$

Our goal is to prove that for any state $\sigma \in \Sigma$:

$$
\begin{aligned}
wlp_a.C.F \implies G \;&\Leftrightarrow \sigma \in wlp_a.C.F \implies \sigma \in G \\
&\Leftrightarrow \neg(\exists \tau \in \Sigma : \sigma \xrightarrow{C} \tau) \vee (\exists \tau \in \Sigma : \sigma \xrightarrow{C} \tau \wedge \tau \in F) \\
&\qquad \implies \sigma \in G \qquad\qquad \text{| Statement 3.13} \\
&\Leftrightarrow (\neg(\exists \tau \in \Sigma : \sigma \xrightarrow{C} \tau) \implies \sigma \in G) \qquad \text{(c)} \\
&\qquad \wedge\ ((\exists \tau \in \Sigma : \sigma \xrightarrow{C} \tau \wedge \tau \in F) \implies \sigma \in G) \qquad \text{(d)}
\end{aligned}
$$

We can prove Lemma 3.14 by proving that Line (a) implies Line (c) and that Line (b) implies Line (d). For any state $\sigma \in \Sigma$, we first prove (a) $\implies$ (c):

$$
\begin{aligned}
\text{true} &\Leftrightarrow (\exists \tau \in \Sigma : (\sigma \xrightarrow{C} \tau) \wedge \neg(\tau \in F)) \implies (\exists \tau \in \Sigma : \sigma \xrightarrow{C} \tau) \\
&\Leftrightarrow \neg(\exists \tau \in \Sigma : \sigma \xrightarrow{C} \tau) \implies \neg(\exists \tau \in \Sigma : (\sigma \xrightarrow{C} \tau) \wedge \neg(\tau \in F)) \\
&\Rightarrow \neg(\exists \tau \in \Sigma : \sigma \xrightarrow{C} \tau) \implies \sigma \in G \qquad\qquad \text{| Line (a)}
\end{aligned}
$$

It is also valid that for any state $\sigma \in \Sigma$, (b) $\implies$ (d). Assume there exists $\tau \in \Sigma$ such that for some state $\sigma \in \Sigma$,

$$\sigma \xrightarrow{C} \tau \wedge \tau \in F \text{ is valid.}$$

Then we conclude from Line (b) that

$$\forall \mu \in \Sigma : \neg(\mu \xrightarrow{C} \tau) \ \vee \ \mu \in G$$

Since $\sigma \in \Sigma$, it follows that $\neg(\sigma \xrightarrow{C} \tau) \ \vee \ \sigma \in G$. We already know that $\sigma \xrightarrow{C} \tau$, hence $\sigma \in G$ must be true, therefore proving Line (d). □

Lemma 3.15 *[G implies angelic wlp]*

if $(P \Longrightarrow G) \Longrightarrow \neg(sp.C.P \Longrightarrow \neg F)$ *then* $G \Longrightarrow wlp_a.C.F$

Here, the prerequisite states that we do not allow executions starting from G that only finish in $\neg F$, making sure that G does not include the red arrows as in Figure 3.8.

Proof. The assumption expresses that for any state $\sigma \in \Sigma$:

$$\begin{aligned}
& P \Longrightarrow G \Longrightarrow \neg(sp.C.P \Longrightarrow \neg F) \\
\Leftrightarrow\ & P \Longrightarrow G \Longrightarrow \neg(\forall \tau \in \Sigma : \tau \in sp.C.P \Longrightarrow \tau \in \neg F) \\
\Leftrightarrow\ & P \Longrightarrow G \Longrightarrow \exists \tau \in \Sigma : \neg(\tau \in sp.C.P \Longrightarrow \tau \in \neg F) \\
\Leftrightarrow\ & P \Longrightarrow G \Longrightarrow \exists \tau \in \Sigma : \tau \in sp.C.P \ \wedge \ \neg(\tau \in \neg F) \\
\Leftrightarrow\ & P \Longrightarrow G \Longrightarrow \exists \tau \in \Sigma : \tau \in sp.C.P \ \wedge \ \tau \in F \\
\Leftrightarrow\ & P \Longrightarrow G \Longrightarrow \exists \tau \in \Sigma : (\exists \mu \in \Sigma : \ \mu \xrightarrow{C} \tau \ \wedge \ \mu \in P) \ \wedge \ \tau \in F \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{| Theorem 2.6} \\
\Leftrightarrow\ & \sigma \in P \Longrightarrow \sigma \in G \Longrightarrow \exists \tau \in \Sigma : (\exists \mu \in \Sigma : \ \mu \xrightarrow{C} \tau \ \wedge \ \mu \in P) \ \wedge \ \tau \in F \qquad \text{(e)}
\end{aligned}$$

Our goal is to prove that for any state $\sigma \in \Sigma$:

$$\begin{aligned}
& G \Longrightarrow wlp_a.C.F \\
\Leftrightarrow\ & \sigma \in G \Longrightarrow \sigma \in wlp_a.C.F \\
\Leftrightarrow\ & \sigma \in G \Longrightarrow \neg(\exists \tau \in \Sigma : \sigma \xrightarrow{C} \tau) \ \vee \ (\exists \tau \in \Sigma : \sigma \xrightarrow{C} \tau \wedge \tau \in F) \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{| Statement 3.13}
\end{aligned}$$

For some state $\sigma \in \Sigma$, assume $\sigma \in G$, then we can construct set $P = \{\sigma\}$ such that the prerequisites in Line (e) holds. Consequently, the postrequisite in holds. Now we can find witnesses μ and τ such that

$$\mu \xrightarrow{C} \tau \ \wedge \ \mu \in P \ \wedge \ \tau \in F$$

Since P is a singleton set, μ can only be σ. Then we have found a witness τ such that $\sigma \xrightarrow{C} \tau$ and $\tau \in F$, satisfying the postrequisite of our goal. □

Corollary 3.16 *[G equivalent to angelic wlp]*
if $wlp.C.F \Longrightarrow G \ \wedge \ sp.C.\neg G \Longrightarrow \neg F$ *and* $P \Longrightarrow G \Longrightarrow \neg(sp.C.P \Longrightarrow \neg F)$
then $G = wlp_a.C.F$

4

CONCLUSIONS

4.1 CONCLUSIONS

In this thesis, we study the weakest liberal precondition transformer and its overapproximation: G such that $wlp.C.F \implies G$. We first discuss the definitions of while-loops in its original form [3] and a variant using fixed points. We establish an equivalence between the two forms of definitions, validating the prudence of the second version. Subsequently, we investigate the G in question. Coincidentally, supplementing G with extra restraints using the strongest postcondition transformer, G coincides with the weakest liberal precondition transformer with angelic non-determinism:

$$(sp.C.\neg G \implies \neg F) \wedge (P \implies G \implies \neg(sp.C.P \implies \neg F)) \implies G = wlp_a.C.F$$

However, without extra constraints, G can be a precondition where all executions are possible. The only certainty is that $\neg G\ \{C\}\ \neg F$ is a valid Hoare Triple. Regardless, G still finds its usefulness while trying to identify preconditions that lead to erroneous final states, when there are initial states that can both lead to errors and successes non-deterministically, or when we do not have sufficient knowledge of all the undesired final states. To do so, one first finds the weakest liberal precondition with respect to the known "bad" final states, then overapproximate the found precondition by "guessing" more possible unwanted final states.

The main contribution of this thesis can be summed up in the following list:

1. We prove that the definitions of while-loops with or without the use of fixed points are equivalent, and gave intuition to the use of fixed points and the necessity of the use of the least fixed point and the greatest fixed point while defining wp and wlp transformers.

2. We give a graphic overview of the relations between the predicate transformers with angelic or demonic non-determinism, which helps demonstrate the triples spawned by underapproximations and overapproximations of the predicate transformers.

3. We conclude that the necessity liberal precondition in general, without further conditions, is possible to be satisfied by any type of initial states. However, the negation $\neg G$ forms a valid Hoare triple with the negation of the postcondition: $\neg G\ \{C\}\ \neg F$.

4. We find that the necessary liberal preconditions are useful when faced with non-deterministic choices that can lead to both error and success, or in situations where we do not have information about all the erroneous final

states. We propose a heuristic to use this triple, and provide a proof system that captures the overapproximation triple.

5. The examples make us notice a special type of initial states, under whose control the execution can terminate in both final states that satisfy the desired postconditions, or final states that oppose them.

   We capture this type of initial states by underapproximating and overapproximating them with the necessary liberal preconditions. Consequently, we find a way to approach wlp_a without having to define it previously.

4.2 FUTURE WORK

We think it is interesting and possible to find rules with denotational semantics to capture the initial states that can lead both to the desired F and its opposite $\neg F$, taking inspiration from incorrectness logic. Additionally, this thesis is only concerned with binary predicates, i.e. a predicate that evaluates to either $true$ or $false$. Albeit classic, it might be more interesting to examine the above results in a quantitative setting, where predicates evaluate to more than $true$ or $false$. In a quantitative setting, the notion of angelic or demonic non-determinism might be too extreme. Instead of regarding the non-determinism as completely in or against our favor, which are strong assumptions, what are the implications when the non-determinism resolves partially in or against our favor? As the poet Qu Yuan said:

路漫漫其修远兮,
Long, long had been my road and far, far was the journey;
吾将上下而求索。
I would go up and down to seek my heart's desire. [8]

BIBLIOGRAPHY

[1] Krzysztof R. Apt. "Ten Years of Hoare's Logic: A Survey—Part I." In: *ACM Trans. Program. Lang. Syst.* 3.4 (Oct. 1981), pp. 431–483. ISSN: 0164-0925. DOI: 10.1145/357146.357150. URL: https://doi.org/10.1145/357146.357150.

[2] Michele Boreale. "Complete algorithms for algebraic strongest postconditions and weakest preconditions in polynomial odes." In: *Science of Computer Programming* 193 (2020), p. 102441. ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2020.102441. URL: https://www.sciencedirect.com/science/article/pii/S0167642320300514.

[3] Edsger W. Dijkstra. "Guarded commands, nondeterminacy and formal derivation of programs." In: *Communications of the ACM* 18.8 (1975), pp. 453–457.

[4] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall Series in Automatic Computation. Englewood Cliffs, N.J: Prentice-Hall, 1976. ISBN: 978-0-13-215871-8.

[5] Edsger W. Dijkstra and Carel S. Scholten. *Predicate Calculus and Program Semantics*. New York, NY: Springer, 1990. ISBN: 978-1-4612-7924-2 978-1-4612-3228-5. DOI: 10.1007/978-1-4612-3228-5.

[6] Robert W. Floyd. "Assigning meanings to programs." In: *Program Verification: Fundamental Issues in Computer Science* (1993), pp. 65–81.

[7] Mike Gordon and Hélène Collavizza. "Forward with Hoare." In: *Reflections on the Work of C.A.R. Hoare*. Ed. by A.W. Roscoe, Cliff B. Jones, and Kenneth R. Wood. London: Springer London, 2010, pp. 101–121. ISBN: 978-1-84882-911-4 978-1-84882-912-1. DOI: 10.1007/978-1-84882-912-1_5. (Visited on 03/10/2024).

[8] David Hawkes, Qu Yuan, and Various. *The Songs of the South: An Anthology of Ancient Chinese Poems by Qu Yuan and Other Poets*. Reissue edition. Harmondsworth: Penguin Classics, Jan. 2012. ISBN: 978-0-14-044375-2.

[9] Charles Antony Richard Hoare. "An axiomatic basis for computer programming." In: *Communications of the ACM* 12.10 (1969), pp. 576–580.

[10] Jozef Hooman. "Extending Hoare logic to real-time." In: *Formal Aspects of Computing* 6 (1994), pp. 801–825.

[11] David Hume. *A treatise of human nature*. Clarendon Press, 1896.

[12] Benjamin Lucien Kaminski. "Advanced weakest precondition calculi for probabilistic programs." PhD thesis. RWTH Aachen University, 2019.

[13] Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Federico Olmedo. "Weakest precondition reasoning for expected run–times of probabilistic programs." In: *Programming Languages and Systems: 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2–8, 2016, Proceedings 25*. Springer. 2016, pp. 364–389.

[14] Gerwin Klein et al. "seL4: formal verification of an OS kernel." In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. SOSP '09. Big Sky, Montana, USA: Association for Computing Machinery, 2009, pp. 207–220. ISBN: 9781605587523. DOI: 10.1145/1629575.1629596. URL: https://doi.org/10.1145/1629575.1629596.

[15] Shoumei Li, Xia Wang, Yoshiaki Okazaki, Jun Kawabe, Toshiaki Murofushi, Li Guan, and Janusz Kacprzyk, eds. *Nonlinear Mathematics for Uncertainty and Its Applications*. Vol. 100. Advances in Intelligent and Soft Computing. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011. ISBN: 978-3-642-22832-2 978-3-642-22833-9. DOI: 10.1007/978-3-642-22833-9. (Visited on 04/01/2024).

[16] Junyi Liu, Li Zhou, Gilles Barthe, and Mingsheng Ying. "Quantum Weakest Preconditions for Reasoning about Expected Runtimes of Quantum Programs." In: *Proceedings of the 37th Annual ACM/IEEE Symposium on Logic in Computer Science*. LICS '22. Haifa, Israel: Association for Computing Machinery, 2022. ISBN: 9781450393515. DOI: 10.1145/3531130.3533327. URL: https://doi.org/10.1145/3531130.3533327.

[17] Zohar Manna and Amir Pnueli. "Axiomatic approach to total correctness of programs." In: *Acta Informatica* 3 (1974), pp. 243–263.

[18] John McCarthy. "Towards a mathematical science of computation." In: *Program Verification: Fundamental Issues in Computer Science* (1993), pp. 35–56.

[19] Magnus O. Myreen and Michael J. C. Gordon. "Hoare Logic for Realistically Modelled Machine Code." In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Orna Grumberg and Michael Huth. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 568–582. ISBN: 978-3-540-71209-1.

[20] Tobias Nipkow and Gerwin Klein. *Concrete semantics: with Isabelle/HOL*. Springer, 2014.

[21] Tobias Nipkow, Markus Wenzel, and Lawrence C Paulson. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer, 2002.

[22] Peter W. O'Hearn. "Incorrectness Logic." In: *Proceedings of the ACM on Programming Languages* 4.POPL (Jan. 2020), pp. 1–32. ISSN: 2475-1421. DOI: 10.1145/3371078. (Visited on 03/10/2024).

[23] Edsko de Vries and Vasileios Koutavas. "Reverse hoare logic." In: *International Conference on Software Engineering and Formal Methods*. Springer. 2011, pp. 155–171.

[24] Gia Wulandari and Detlef Plump. *Verifying Graph Programs with First-Order Logic (Extended Version)*. Nov. 2020. arXiv: 2010.14549 [cs]. (Visited on 04/01/2024).

[25] Linpeng Zhang and Benjamin Kaminski. "Quantitative Strongest Post." In: *arXiv preprint arXiv:2202.06765* (2022).

[26] Li Zhou, Nengkun Yu, and Mingsheng Ying. "An applied quantum Hoare logic." In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2019. Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 1149–1162. ISBN: 9781450367127. DOI: 10.1145/3314221.3314584. URL: https://doi.org/10.1145/3314221.3314584.