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*" [13]. 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 [16], the first formally proven operating system kernel against overflows, memory leaks, non-termination, etc. using the interactive theorem prover Isabelle/HOL [27]. 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 [11]. 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 [5] (wp): starting with a postcondition, it works backwards and calculates what the weakest precondition is that guarantees both correctness and termination. A predicate P is considered to be weaker than Q, if $Q \implies P$ is valid. Conversely, Q is stronger than P. 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

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

precondition. Relaxing the commitment for termination, Dijkstra also proposed the weakest liberal precondition transformer [7] (wlp) which delivers preconditions so that the program either terminates correctly or never terminates, proving partial correctness. The name contains the word liberal, in the sense that the wlp transformer takes a more "liberal" view on non-termination, that divergence or execution forever is considered as acceptable in this case.

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 [9]. Hoare triple also finds applications in low-level programming languages [23], quantum programs [33], distributed real-time systems [12], and so on.

Likewise, Dijkstra's wp transformer also spawned various scientific work on probabilistic programs [15], quantum programs [2], continuous programs [19], and so much more. Especially inspiring for this thesis is the work by O'Hearn named incorrectness logic [28]. It studies an underapproximation (subset) of the strongest postcondition transformer [7] (sp) by Dijkstra, focusing on reachability of final states to prove the existence of bugs. He proposed the incorrectness triple starting from the implication

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

Here, 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

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

where 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 prove that the definition of the weakest precondition transformer of while-programs by Dijkstra [5] 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 in Figure 3.6.

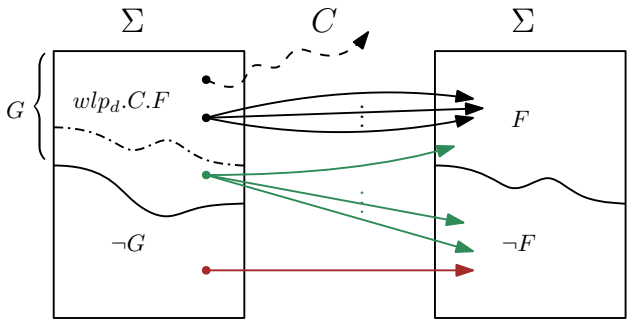


Figure 1.1: Necessarily Liberal Precondition G That Additionally Has Green Dots

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 ¬G will lead to termination in states satisfying ¬F. Following this, we demonstrate the usefulness of this overapproximation with an example.

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

6. We also notice a special group of initial states, starting from which the execution is possible to terminate satisfying both the desired postcondition and its opposite, like the green parts in Figure 1.1. We capture these preconditions by approaching from top and bottom, namely finding scenarios in which 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 2.9 and list some properties in Section 2.10.

In Chapter 4, we first relate wp, wlp, sp, slp together using various relations, then illustrate them in a graph in Chapter 3. 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 4.2. 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 4.3. Finally, we summarize our conclusions in Chapter 5 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
$\exists x.P : F$	There exists x that satisfies P, and also satisfies F
$\forall x.P : F$	All x that satisfy P, also satisfy 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 $F(\sigma) = \mathit{true}$ 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 [8, 22] 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* [11] 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 [14], 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.

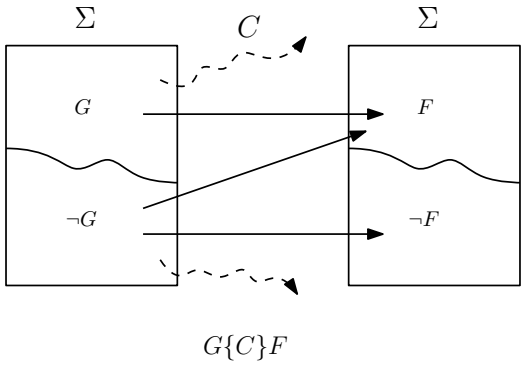


Figure 2.1: Valid Hoare Triple (Deterministic)

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 [21], 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 [5], among other things.

2.3 GUARDED COMMAND LANGUAGE

From now on we use Dijkstra's (non-deterministic) guarded command language (GCL) [5] 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 [5]; $\{C_1\}\square\{C_2\}$ 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.

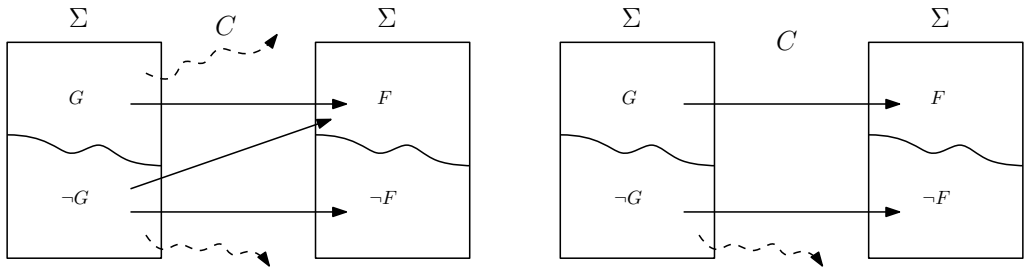
When skip is executed, the program state does not change and the consecutive part is executed. When diverge is executed, the execution never stops and the program can not reach a final state.

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 [7]. 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 [14]

A predicate P is weaker than Q, if and only if $Q \implies P$ is valid, and Q is then stronger than P. In this case we also say that P is an overapproximation of Q, and Q is an underapproximation of P.

$F[x/e]$ 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

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.

satisfying the precondition $wp.diverge.F$ 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 $false$.

The definition for the while-loop [14] 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 [5], 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

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

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 $IF = \underline{\text{if}}\ B_1 \to SL_1\ []\ \ldots\ []\ B_n \to SL_n\ \underline{\text{fi}}$ where divergence is possible in case none of the guards is true, we need $diverge$ instead of $skip$ in the $else$-branch.

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

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

IF is defined in such way that $wp.IF.X$ 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:

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

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

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

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, we know that the semantics of WHILE is equivalent to the semantics of $\mathtt{if}(\varphi)\{C'; WHILE\}\ else\ \{skip\}$ [20]. 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:

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

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 [7] 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]* [14]

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

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:

$$\begin{aligned}
\Phi(\mathit{false}) &= (\neg\varphi \wedge F) \vee (\varphi \wedge wp.C'.\mathit{false}) \\
&= (\neg\varphi \wedge F) \vee (\varphi \wedge \mathit{false}) && \mid (***) \\
&= \neg\varphi \wedge F && \mid \text{predicate calculus} \\
&= H_0(F)
\end{aligned}$$

Line (***) is supported by the Law of Excluded Miracle [6, p.18]: for all programs C, $wp.C.\mathit{false} = \mathit{false}$. It states that it is impossible for a program to terminate in a state satisfying no postcondition.

STEP CASE:

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

□

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

$$\mathrm{lfp}\ \Phi = wp.\mathrm{WHILE}.F$$

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 $\mathrm{lfp}\ \Phi$ is the disjunction of $\Phi^k(\mathit{false})$ for all non-negative k, whereas Lemma 2.2 tells us that $H_k(F) = \Phi^{k+1}(\mathit{false})$.

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 $\mathrm{lfp}\ \Phi$ is

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

or *W*HILE 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 $\mathrm{lfp}\ \Phi$ is the disjunction of all preconditions that guarantee the termination of *W*HILE 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 $p \implies wlp.\mathit{W}\mathrm{HILE}.F$, where p is a fixed point of Φ. Following the same reasoning as Equation 2.2, we can conclude that $wlp.C.F$ is a fixed point of Φ. Hence, by definition, $wlp.\mathit{W}\mathrm{HILE}.F = \mathrm{gfp}\ \Phi$.

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

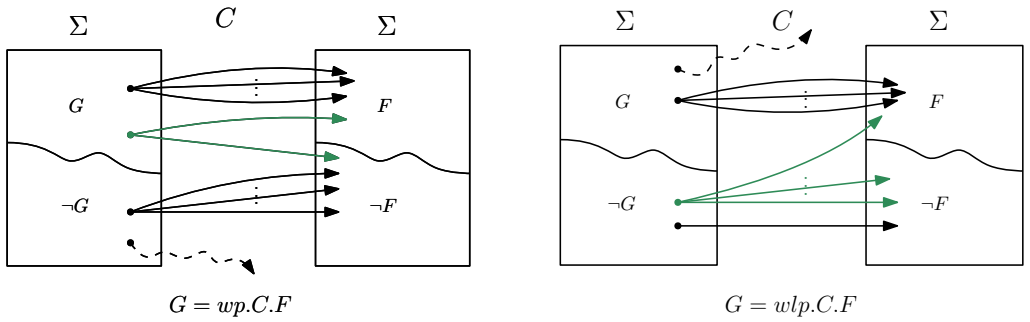
Now we bring the non-deterministic choice back into the picture and add its wp to Table 2.5. Here we assume a setting with angelic non-determinism, where we assume that whenever non-determinism occurs, it will be resolved in our favor. This means we are optimistic that the non-determinism is likely to resolve to our favor, meaning that we consider an initial state can lead the program execution to terminate in a final state satisfying F, we consider it a valid initial state to satisfy the weakest (liberal) precondition. Conversely, if we consider the non-deterministic choice to be demonic, then we require that all executions from an initial state must satisfy F for this state to be included in the set of states described by the weakest (liberal) precondition.

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 [5], 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 [31].

C	wp.C.F	wlp.C.F
skip	F	F
diverge	false	true
$x := e$	$F[x/e]$	$F[x/e]$
$C_1; C_2$	$wp.C_1.(wp.C_2.F)$	$wlp.C_1.(wlp.C_2.F)$
$\{C_1\}\square\{C_2\}$	$wp.C_1.F \vee wp.C_2.F$	$wlp.C_1.F \wedge wlp.C_2.F$
if (φ) $\{C_1\}$ else $\{C_2\}$	$(\varphi \wedge wp.C_1.F) \vee (\neg\varphi \wedge wp.C_2.F)$	$(\varphi \wedge wlp.C_1.F) \vee (\neg\varphi \wedge wlp.C_2.F)$
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 [14]

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).

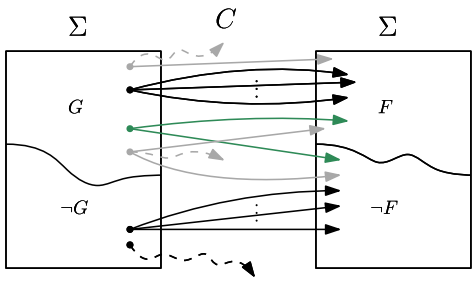


$G = wp.C.F$ (1) wp (angelic non-determinism)

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

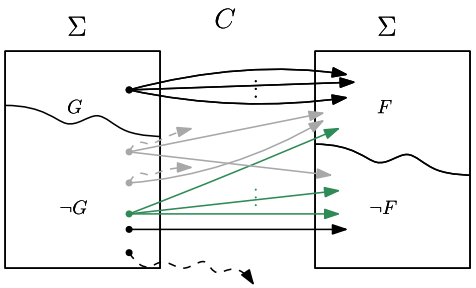
Figure 2.3: Weakest Precondition (Angelic Non-determinism) and Weakest Liberal Precondition (Demonic Non-determinism)

Note that wp with angelic resolution of non-determinism can be satisfied with initial states from which the execution of C may either diverge or terminate satisfying F, as shown with the gray arrows in Figure 2.4.1. Similar statements can be said about the negation of wlp with demonic resolution of non-determinism. However, to make the graphs clear, we omit drawing the gray arrows unless necessary, because we recon that the essence of the predicate transformers are already captured using graphs similar to Figure 2.3. For completeness' sake, we



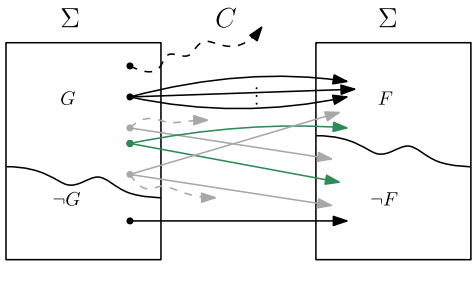
$G = wp_a.C.F$

(1) wp (angelic non-determinism)



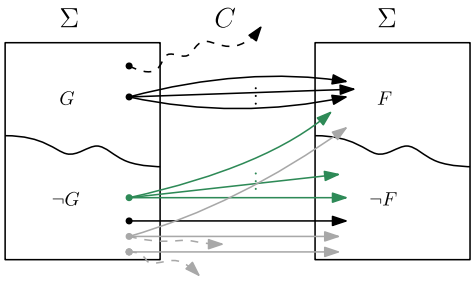
$G = wp_d.C.F$

(2) wp (demonic non-determinism)



$G = wlp_a.C.F$

(3) wlp (angelic non-determinism)



$G = wlp_d.C.F$

(4) wlp (demonic non-determinism)

Figure 2.4: Weakest (Liberal) Precondition Transformers with More Detail

show all the gray arrows in Figure 2.4, but for the remainder of this thesis, we keep the graphs compact.

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 post-condition in the wlp setting is guaranteed upon termination, because we regard the non-deterministic choice as demonic, again favoring to establish a Galois connection [31].

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 precon-dition, and $wlp.\mathrm{WHILE}.F$ should be a fixed point of its characteristic function, similar to Section 2.4.2.

Theorem 2.3 [14]*[gfp heuristics]* $\mathtt{gfp}\ \Phi = \sup_{n \in \mathbb{N}} \Phi^n(\mathtt{true})$

2.6 STRONGEST POSTCONDITIONS

Following the style to define wp and wlp, Zhang and Kaminski [31] (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
$x := e$	$\exists a.x = e[x/a] \wedge G[x/a]$
$C_1; C_2$	$sp.C_2.(sp.C_1.G)$
$\{C_1\}\square\{C_2\}$	$sp.C_1.G \vee sp.C_2.G$
$\mathit{if}\ (\varphi)\ \{C_1\}\ \mathit{else}\ \{C_2\}$	$sp.C_1.(\varphi \wedge G) \vee sp.C_2.(\neg\varphi \wedge G)$
$\mathit{while}\ (\varphi)\ \{C'\}$	$\neg\varphi \wedge \mathrm{lfp}\ X.G \vee sp.C.(\varphi \wedge X)$

Table 2.6: The Strongest Postcondition Transformer [31]

We can also illustrate the behavior of a program controlled by sp in Figure 2.5. Instead of discussing termination starting from a precondition, sp focuses on reachability of states satisfying postconditions. The dotted arrow points to postconditions describing unreachable final states after the execution of C. For example, no state would satisfy $x = 2$ after the execution of $x := 1$.

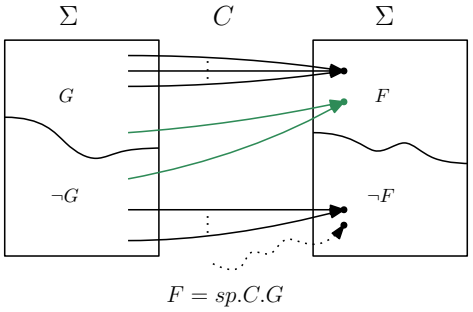


Figure 2.5: Strongest Postcondition (Angelic Non-determinism)

2.7 BIG STEP SEMANTICS

Big-step semantics are often seen while reasoning about the semantics of programming languages, it is known to generate simple proofs. To express the meaning of GCL programs, we can use big-step semantics to describe executions of a program. Taking inspiration from Nipkow and Klein's book [26] we define the big-step semantics in Table 2.7.

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

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

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

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

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

Table 2.7: Big Step Semantics

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 which would restrict 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, divergence is defined to lead to state $\perp$ that is not a part of Σ.

Let $\sigma \in \Sigma$ be a non-trivial state, where non-trivial means that $\sigma \vDash true$ is valid. To denote non-termination of a program C with initial state σ, a simple approach is to consider it as equivalent to the non-existence of a state $\tau \in \Sigma$ that $\sigma \xrightarrow{C} \tau$. This covers the case of divergence, since there is no rule given for $diverge$, there naturally does not exist a final state that program $diverge$ can take a big-step to. Note that divergence is expressed if all executions of a program diverges. If only some executions diverge, the terminating executions are captured. As an example, consider the program

$$D :=\ (x := y \square\ while\ (true)\ do\ \{x := z\})$$

First, there is no rule to prove the while-loop on the right hand of the choice. Since $\sigma \notin true$ is the same as with our notation, $\sigma \notin \{\sigma \in \Sigma \mid \sigma \vDash true\}$ can not be valid, since we are looking at a non-trivial σ. Consequently, we can not use the $while_0$ rule, hence the last premise of $while_n$ rule will never be fulfilled for this while-loop. As a result, we can only capture the left side of the non-deterministic choice:

$$\frac{\dfrac{}{\sigma \xrightarrow{x:=y} \sigma(x := \sigma.y)}\,assign}{\sigma \xrightarrow{x:=y \square while\ (true)\ do\ \{x:=z\}} \sigma(x := \sigma.y)}\,choice_1$$

As a result, we can conclude that $\sigma \xrightarrow{D} \tau$. This is not problematic for us to reason about wp with angelic non-determinism resolution and wlp with demonic resolution, but some still prefer to be able to explicitly reason about non-terminating

runs by extending the big-step semantics [17, 24] or turn to small-step semantics [24] and use paths to distinguish between executions. For this thesis, we choose another semantics to express soundness properties of predicate transformers, namely the collecting semantics, for their simplicity and elegance.

2.8 COLLECTING SEMANTICS

Zhang and Kaminski assign meaning to GCL programs with a collecting semantics [32]. Informally, we "collect" reachable states via program execution starting from a set of initial states. The definition is shown in Table 2.8.

Let $\mathit{Conf} = \mathcal{P}(\Sigma)$, the set of configurations in form of the power set of the set of states. An element of the Conf set is then a set of program states. For a configuration S, a filtering $[\![\varphi]\!]S := \{\sigma \mid \sigma \in S \land \sigma \vDash \varphi\}$ is the set of program states in S that satisfy the predicate φ. The circle $\circ$ is composition as usual: $([\![C_2]\!] \circ [\![C_1]\!])S = [\![C_2]\!]([\![C_1]\!]S)$.

C	$[\![\mathbf{C}]\!]\ \mathbf{S}$
skip	S
diverge	$\emptyset$
$x := e$	$\{\sigma[x/\sigma(e)] \mid \sigma \in S\}$
$C_1; C_2$	$([\![C_2]\!] \circ [\![C_1]\!])S$
$\{C_1\}\square\{C_2\}$	$[\![C_1]\!]S \cup [\![C_2]\!]S$
if (φ) $\{C_1\}$ else $\{C_2\}$	$([\![C_1]\!] \circ [\![\varphi]\!])S \cup ([\![C_2]\!] \circ [\![\neg\varphi]\!])S$
while (φ) $\{C'\}$	$[\![\neg\varphi]\!](\mathrm{lfp}\ X.\ S \cup ([\![C']\!] \circ [\![\varphi]\!])X)$

Table 2.8: Collecting Semantics for GCL [32]

Originally, the collecting semantics for diverge and skip was not explicitly given, but since $[\![\cdot]\!]\cdot$ "collects" reachable final states, $[\![\mathrm{skip}]\!]S$ should return S, and $[\![\mathrm{diverge}]\!]S$ should be the empty set for any S.

We see the difference between this collecting semantics and the big-step semantics for GCL Section 2.7 by looking at the same program D:

Remember from Section 2.1 that a predicate is a function that maps program states to truth values, and it can be written in a set form: $F = \{\sigma \in \Sigma \mid F(\sigma)\}$. This gives us two interesting insights:

First, a configuration corresponds to a class of equivalent predicates. Note that we can have predicates in different form that identify the same mapping from program states to truth values, for example

$$P \implies Q \quad \text{and} \quad \neg P \lor Q$$

Then the set form of multiple predicates can be identical to the set of program states described by the configuration. Nevertheless, we can see that a configuration is not much different from a predicate. Second, a filtering is a set intersection.

Rewriting the filtering, we get $[\![\varphi]\!]S = \{\sigma \mid \sigma \in S \wedge \sigma \in \varphi\} = \varphi \cap S$. With collecting semantics, we can define the soundness of the predicate transformers in the next section.

2.9 SOUNDNESS

Remember that a predicate is a function from program states to truth values (see Section 2.1). Then we can apply a predicate to a program state to get its truth value: $wlp.C.F.\sigma$, showing the advantage of using lambda-calculus style notation for function application.

Theorem 2.4 *[Soundness of wp]* [31]

$$wp.C.F.\sigma = \bigvee_{\tau \in [\![C]\!]\{\sigma\}} F.\tau$$

$wp.C.F$ is satisfied exactly by the initial states starting from which the execution of C terminates, and a final state satisfying F is always reachable. We can rewrite Theorem 2.4 as[5]:

$$wp.C.F.\sigma = \exists \tau \in [\![C]\!]\{\sigma\} : F.\tau$$

Recall from Section 2.1 that predicate F also defines a set $\{\sigma \in \Sigma \mid F.\sigma\}$, then we can rewrite the equation above as:

Corollary 2.5 *[Soundness of wp]*

$$wp.C.F = \{\sigma \in \Sigma \mid \exists \tau \in [\![C]\!]\{\sigma\} : F.\tau\}$$

Theorem 2.6 *[Soundness of wlp]* [7, 31]

$$wlp.C.F.\sigma = \bigwedge_{\tau \in [\![C]\!]\{\sigma\}} F.\tau$$

$wlp.C.F$ is satisfied exactly by initial states under whose control the execution of C either never terminates or terminates satisfying F

Similar as before, we can rewrite Theorem 2.6 as:

$$wlp.C.F.\sigma = \forall \tau \in [\![C]\!]\{\sigma\} : F.\tau$$

And consequently:

Corollary 2.7 *[Soundness of wlp]*

$$wlp.C.F = \{\sigma \in \Sigma \mid \forall \tau \in [\![C]\!]\{\sigma\} : F.\tau\}$$

From Theorem 2.4 and Theorem 2.6, we see again that the resolving the nondeterminism of wp as angelic and that of wlp is demonic is a good choice. It yields a simple and elegant way to define the soundness properties of wp and wlp using the collecting semantics, which the demonic resolution for wp and angelic resolution for wlp can not achieve.

To support the reasoning in Section 4.3, we also need the soundness property of sp:

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

Theorem 2.8 *[Soundness of sp]* [29, 31]

$$sp.C.G.\tau = \bigvee_{\sigma \text{ where } \tau \in [\![C]\!]\{\sigma\}} G.\sigma$$

sp.C.G is satisfied exactly by final states that are definitely reachable from some initial state satisfying G. Similarly, we can rewrite the soundness property of sp as:

$$sp.C.G.\tau = \exists \sigma.\tau \in [\![C]\!]\{\sigma\} : G.\sigma$$

And consequently:

Corollary 2.9 *[Soundness of sp]*

$$sp.C.G = \{\tau \in \Sigma \mid \exists \sigma.\tau \in [\![C]\!]\{\sigma\} : G.\sigma\}$$

For completeness' sake, we also state the soundness property of slp:

Theorem 2.10 *[Soundness of slp]* [31]

$$slp.C.G.\tau = \bigwedge_{\sigma \text{ where } \tau \in [\![C]\!]\{\sigma\}} G.\sigma$$

Corollary 2.11 *[Soundness of slp]*

$$slp.C.G = \{\tau \in \Sigma \mid \forall \sigma.\tau \in [\![C]\!]\{\sigma\} : \ G.\sigma\}$$

2.10 PROPERTIES OF WP AND WLP

We also include some properties of predicate transformers in this section that we will need later in Chapter 4 for our proofs.

Theorem 2.12 *[Conjugates]* [31] *wp and wlp are each other's conjugates:*

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

This property can be rewritten as $wp.C.F \vee \neg wlp.C.\neg F = true$. It tells us that wp and wlp complement each other, like in Figure 2.6.

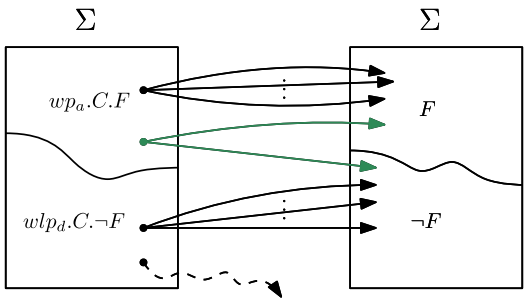


Figure 2.6: wp and wlp Are Conjugates

Theorem 2.13 *[Monotonicity]* [7] *For any* $C \in \mathcal{C}$, $wlp.C$ *is monotonic:*

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

It states that if we relax the postcondition into a weaker, more general condition, then we should be able to find more initial states that can lead to final states that satisfy this weaker postcondition upon termination.

Theorem 2.14 [7] $\forall C \in \mathcal{C} : \; wlp.C.true = true$

The left side of the equation describes the set of initial states that "either never terminates, or terminates satisfying $true$", which describes all initial states, hence it should equal to the set of all states, i.e. a predicate that maps all states to $true$.

With these theorems, we finish this chapter and the first part of this thesis. In the first part, we explain our motivation to study the predicate transformer, then proceed to discuss background information by first list the notations used throughout this thesis. Following this, we showcased the relevant formalism for this thesis, namely Hoare triples, the weakest (liberal) precondition transformer, and the strongest postcondition transformer.

During this process, we give intuition behind the definitions we use, and compare them to their original form. We also discuss the different perspectives to resolve non-determinism: angelic versus demonic, and give an excursion about the use of least and greatest fixed points in our definition. We also show our attempts to grasp the soundness properties of the predicate transformers using the big-step semantics, and conclude its ineptness for our purpose. In conclusion, we choose the collecting semantics to describe the soundness properties of our predicate transformers, and mention some other properties as well.

Concluding the first part, we proceed in the second part to further study the focus of this thesis. We call it the necessary liberal precondition. Necessary, because its violation will lead to violation of the given postcondition, the definition of a necessary condition in logical terms. Liberal, because we take a liberal view on non-termination. We accept initial states that can lead to non-termination as satisfying our precondition. Precondition, because we are concerned with the conditions before the execution of a program.

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

PRECONDITIONS, POSTCONDITIONS, AND THEIR APPROXIMATIONS

In this chapter, we give an overview of precondition and postcondition transformers presented in the previous chapter, establishing relations between all of them using termination, conjugation, reachability, and Galois connections. This reveals that the necessary liberal precondition triple that we investigate in this thesis is still an open question in the sense that it is not yet well-researched to our knowledge, and that it is not automatically related to another triple by the aforementioned relations. Furthermore, the methods mentioned in this chapter used to study other triples prove helpful while investigating necessary liberal preconditions.

3.1 PRECONDITION TRANSFORMERS

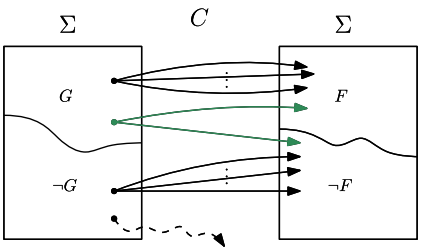
Figure 3.1 shows the impact of the interpretation of non-determinism on the semantics of the wp and wlp transformers. The subscripts $_a$ and $_d$ correspond to angelic and demonic resolution of non-determinism mentioned in Section 2.4.3, respectively. The notion of angelic and demonic is reflected by the placement of green dots and arrows in Figure 3.1.

In general, the rectangles denote the set of all states Σ. C is the program that we are investigating, all the arrows are executions of program C. When a rectangle is divided by a line into multiple parts, it means that the states are categorized into different types. For example, the section labelled with $wlp.C.F$ denotes all the states that satisfy $wlp.C.F$. The dashed arrow starting from inside a rectangle, ending outside any rectangle denotes executions that do not terminate. The arrows that share the same starting point denote different executions starting from the same initial state, and those that share an ending point denote executions that end in the same final state.

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,

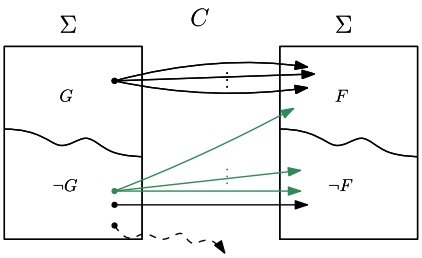
$$wp.C.(\{C_1\}\square\{C_2\}) = wp.C_1.F \vee wp.C_2.F$$

for a postcondition F in Table 2.5.



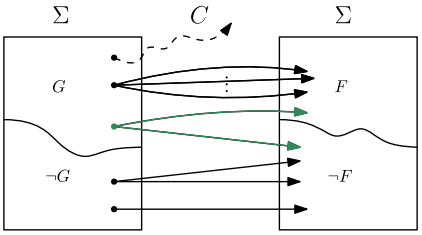
$G = wp_a.C.F$

(1) Weakest precondition (angelic non-determinism)



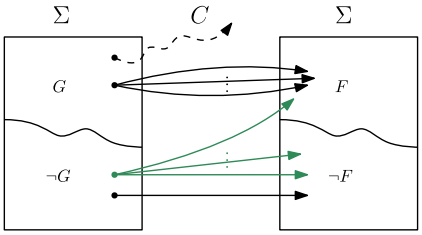
$G = wp_d.C.F$

(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

On the other hand, if the non-deterministic choice is resolved as demonic, we are pessimistic over the result of the non-deterministic choice: we assume it will not resolve to our favor, hence we are only confident to include a condition into the valid precondition set if all possible paths end in a final state satisfying our desired postcondition upon termination. Graphically, the green arrows are not included in wp_d and wlp_D in Figure 3.1.2 and Figure 3.1.4. The definition is then using conjuncts instead of disjuncts, like

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

for a postcondition F in Table 2.5.

Note that we keep the figures in Figure 3.1 clear and compact but still keep the essence of the predicate transformers. A more detailed version can be found in Figure 2.4.

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 [7]:

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

This means 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 3.1.4 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 [11] with rules to prove termination [21]) 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, for example angelic resolution for wp and demonic resolution for wlp, we end up with a different property that wp and wlp are each other's conjugates [6, 31]:

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

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 3.1.4 and vice versa, like in Figure 3.2.

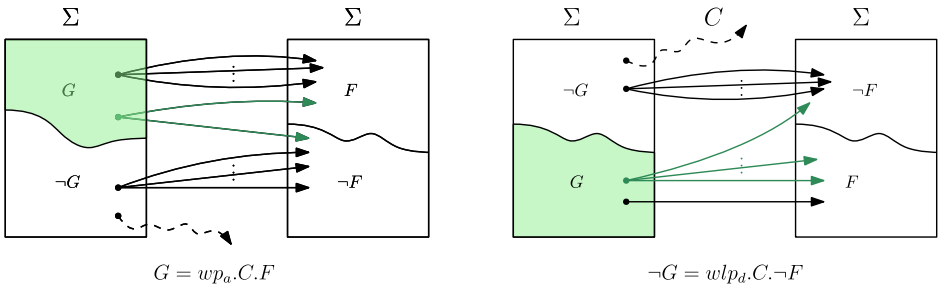


Figure 3.2: wp_a and wlp_d Are Conjugates

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.

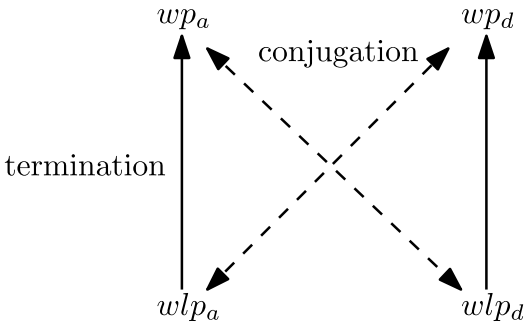


Figure 3.3: Relations in wp-Family

In this thesis, we prefer wp with angelic non-determinism and wlp with demonic non-determinism, because this choice gifts us Galois connection between the precondition transformers and the postcondition transformers [31], which we will illustrate in Section 3.3:

Theorem 3.15 *[Galois connection between wp and slp]*

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

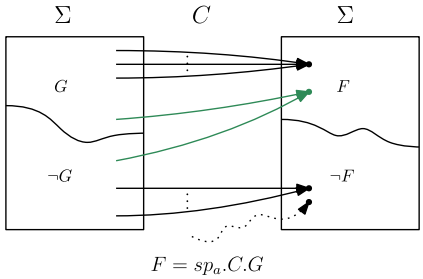
Theorem 3.16 *[Galois connection between wlp and sp]*

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

3.2 POSTCONDITION TRANSFORMERS

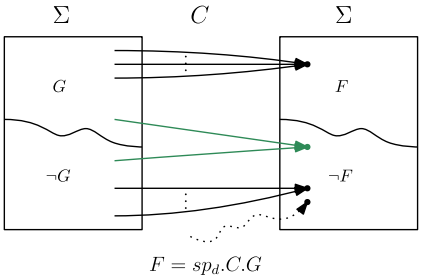
The strongest postcondition was originally discussed as a side-product of the wp and wlp transformers by Dijkstra and Scholten [7]. sp.S.Y is meant to describe "those final states for which there exists a computation under control of S that belongs to the class *initially Y*", but it has found more important utilities. For example, de Vries and Koutavas [29] use sp to reason with randomized algorithms, although they called it weakest postconditions. O'Hearn [28] 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.



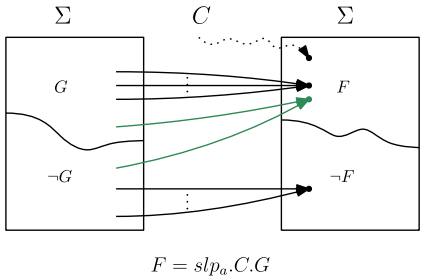
$F = sp_a.C.G$

(1) Strongest postcondition (angelic non-determinism)



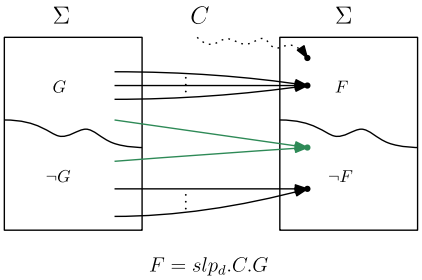
$F = sp_d.C.G$

(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

Note that while sp is well-researched, slp is less investigated to the best of our knowledge. For example, Wulandari and Plump [30] defined and proved

soundness for slp, but only for loop-free programs. While Li et al. [18] 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.

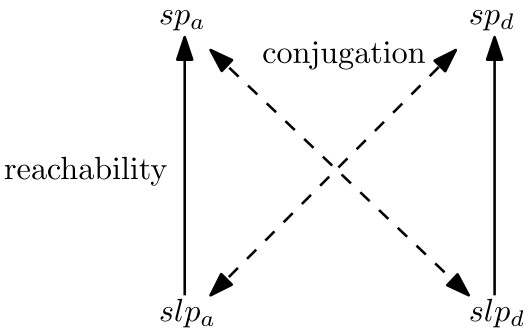


Figure 3.5: Relations in sp-Family

3.3 APPROXIMATION TRIPLES IN LITERATURE

Supplementing the Galois connection specified in Theorem 3.15 and Theorem 3.16, 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.

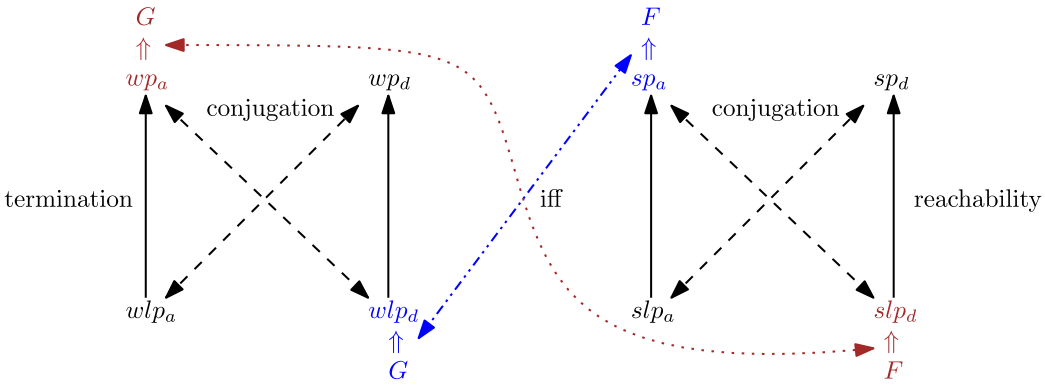


Figure 3.6: Connecting wp-Family and sp-Family

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 choosing the formalism that are colored in Figure 3.6, all the under- and over-approximations over the predicate transformers are related well together, as we depict in Figure 3.7.

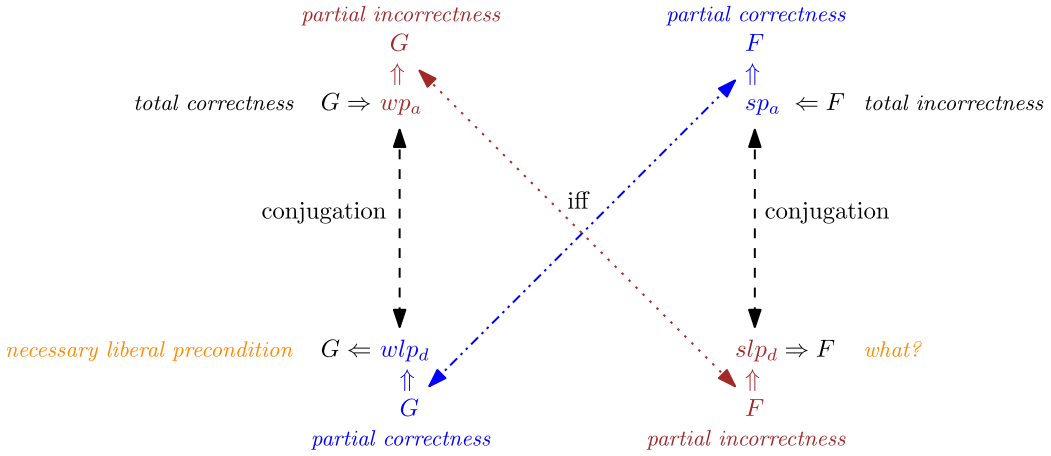


Figure 3.7: Connecting Overapproximations and Underapproximations

Some of the implications in Figure 3.6 are well-studied by now, as we will see in the upcoming paragraphs. Since the view on angelic or demonic nondeterminism varies depending on the authors' choice, and do not make fundamental differences on their results, we will mention it in the description, but drop the subscripts $_a$ and $_d$ in the remainder of the section.

PARITIAL CORRECTNESS For example, the blue implication

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

corresponds to a Hoare triple [11] with respect to partial correctness: $G\ \{C\}\ F$ is valid if and only if $G \implies wlp.C.F$ [9, 27]. This means that if the program C starts in a state satisfying G, then the execution must end in a final state satisfying F, if it terminates. In terms of approximation, we can say that the precondition in the valid Hoare triple is an underapproximation of the weakest precondition.

Hoare triple does not originally contain rules for non-deterministic choice, but it can be added without much difficulty, for example with demonic nondeterminism [25]:

$$\frac{G\ \{C_1\}\ F,\ G\ \{C_2\}\ F}{G\ \{C_1 \square C_2\}\ F}$$

As we can see in Figure 3.6, the blue implications satisfy the if-and-only-if relation. This means that an overapproximation of the strongest postcondition

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

also guarantees partial correctness, aka $G\ \{C\}\ F$ is a valid Hoare triple.

TOTAL CORRECTNESS Manna and Pnueli [21] supplimented Hoare triple with rules to prove termination, making the extended proof system valid for total correctness:

$$\frac{P \wedge \varphi \wedge v \in \mathbb{N} \wedge v = n \quad \{C\} \quad P \wedge \phi \wedge v \in \mathbb{N} \wedge v < n}{P \wedge v \in \mathbb{N} \quad \{\mathit{while}\ (\varphi)\ \mathit{do}\ C\} \quad \neg\varphi \wedge P \wedge t \in \mathbb{N}}$$

P is the so-called invariant, a condition that stays unchanged before and after the program execution. ν in this case is the variant, it is modified by the execution of the program. Specifically, ν reduces with each execution of the sub-program C inside the loop body. Since ν is a natural number, it is never negative, i.e. bound from the bottom. Naturally, execution of the loop has to stop at some point, the loop body C can be executed maximum ν times. This version of Hoare triple proves total correctness (termination and correctness), and it corresponds to the implication labelled with "total correctness" in Figure 3.6:

$$G \implies wp.C.F$$

TOTAL INCORRECTNESS In the correctness realm, we are concerned with termination and guaranteeing that only good things can happen in the end. Changing the perspective to focus on reachability of erroneous final states, we end up in the incorrectness realm. The implication marked with "total incorrectness" in Figure 3.6 was studied by O'Hearn [28] under the name "incorrectness logic", and by de Vries and Koutavas [29] with the name "reverse Hoare logic":

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

With this triple, we can prove the reachability of states described by F starting from the initial condition satisfying G. We can view it as an indication that the program C was not constructed correctly, and bugs described by F are reachable [28]; or use it to prove that the final states in F are reachable via randomized programs [29].

They all take an angelic view on the non-determinism, as well as Dijkstra and Scholten [7]. For example, O'Hearn defines the rule for non-deterministic choice as follows(ϵ is either ok or er, an indicator whether the final condition is reached normally or erroneously):

$$\frac{[P]C_i[\epsilon : Q]}{[P]C_1 + C_2[\epsilon : Q]} \text{ choice (where } i = 1 \text{ or } 2)$$

Zhang and Kaminski [31] supplemented the word "total" to the name of this triple, reason being that the strongest postcondition transformer only include final states that are reachable, dual to the total correctness triple that only allows states that lead to termination.

PARTIAL INCORRECTNESS As the name indicates, when there is "total incorrectness", there should be "partial incorrectness". The red implications correspond to the triples for partial incorrectness. As mentioned in the previous section, the slp transformer lacks attention, starting with a sound definition with loops [18, 30]. As far as our knowledge, the underapproximation triple of slp

$$F \implies slp.C.G$$

still needs proper investigation. However, the Galois connection relates it to the overapproximation triple of wp:

$$wp.C.F \implies G$$

Cousot et al. [4] studied an implication similar to this triple to help lighten the burden of providing specifications in design by contract programming schemes. They call it the "necessary preconditions", in the sense that these preconditions can not guarantee correctness, but once violated, definitely lead to erroneous final states. They achieve this by finding preconditions that will definitely lead to erroneous final states, then take its negation. Because they separated the issue of termination from their necessary precondition reasoning by assuming that all programs terminate, their necessary precondition is not exactly an overapproximation of wp. Nevertheless, it gives intuition on one of the ways to approach the necessary liberal precondition.

NECESSARY LIBERAL PRECONDITION The overapproximation of wlp is the main focus of this thesis, denoted with orange label in Figure 3.6: $wlp.C.F \implies G$. It reflexes an extremely relaxed view: in this triple, G can be satisfied by all possible initial states, under whose control the program can diverge, successfully terminate, erroneously terminate ... The only certainty is that G is the necessary condition, in the sense that once violated, the program can definitely reach erroneous final states. We will study this triple further in the upcoming sections.

REMAINING TRIPLE The remaining triple denoted with label "what?" still awaits investigation as far as we know:

$$slp.C.G \implies F$$

We think that one approach is to first establish a sound definition for slp, then find connection between slp and sp, either by conjugation or reachability depending on the choice how to resolve non-determinism. From here, one can start investigating this triple following similar methods for other triples.

With this overview of triples studied in literature, we know that the triple $wlp.C.F \implies G$ is still an open question, and not automatically solved by a Galois connection to another implication. As a result, we investigate the overapproximation we call the necessary liberal precondition following methods used with other triples by first discussing the possible scenarios of this overapproximation, then look into examples to shed light on the meaning of this implication.

4

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 be 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 relate together wp, wlp, sp, and slp with both angelic and demonic non-determinism, so we can distinguish the necessary liberal precondition from the existing triples that are already well-studied. In this chapter, we first discuss the general semantics of the necessary liberal precondition, then focus on a characteristic scenario and show properties that captures it.

4.1 RELATING OVERAPPROXIMATION OF WLP WITH SP

Remember in Section 3.3, we mentioned that there is an equivalence relation between the underapproximation of wlp and overapproximation of sp via Galois connection:

$$G \implies wlp.C.F \text{ iff } sp.C.G \implies F \tag{*}$$

One might wonder, can we also establish some similar relation between the overapproximation of wlp and underapproximation of sp:

$$wlp.C.F \implies G \quad \not\leftrightarrow \quad F \implies sp.C.G \tag{**}$$

This would imply that we merely need to study the underapproximation of sp to result in the overapproximation of wlp, the former of which is studied as the total incorrectness triple [28].

The answer is no. Without additional constraints, we can not directly establish any implication relations between $wlp.C.F \implies G$ and $F \implies sp.C.G$.

Remember from Chapter 3 that $wlp.C.F$ describes all initial states that can potentially lead to non-termination, and $sp.C.G$ is satisfied only by final states that are reachable. Termination and reachability are two different aspects of program execution. If we denote explicitly the states that can lead to non-termination with blue squares, and states that are unreachable with orange circles, then we can see on the left half of Figure 4.1 that F can also be satisfied with unreachable states, and from the right half that G can also be satisfied by states leading to non-termination. In general, without further constraints, F in Figure 4.1 is decorated with some of the orange circles, and G is decorated with some of the blue squares.

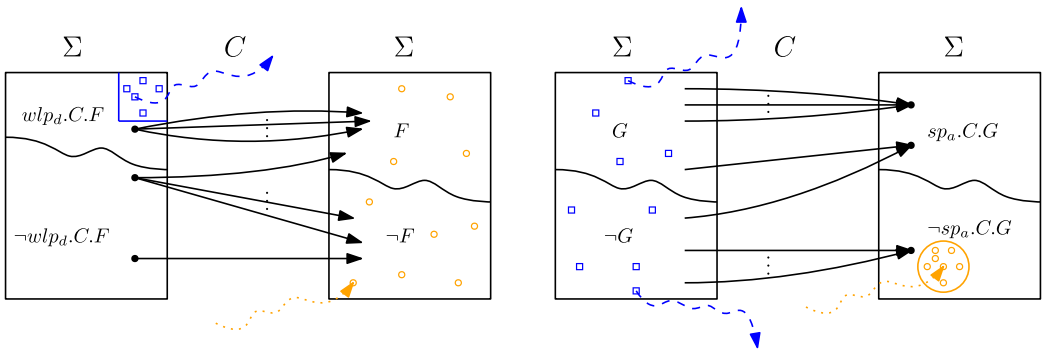


Figure 4.1: Non-terminating and unreachable States

As an example, consider the program

$$C = \text{if } (x = 2) \text{ then } \{x := x + 2\} \text{ else } \{x := x + 1\}$$

and a postcondition $F = \{x > 2\}$. We can deduce the weakest liberal precondition:

$$\begin{aligned} wlp.C.F &= \{x = 2\} \wedge (wlp.x := x + 2).F \ \vee \ \{x \neq 2\} \wedge wlp.(x := x + 1).F \\ &\Leftrightarrow \{x = 2\} \wedge \{x > 0\} \ \vee \ \{x \neq 2\} \wedge \{x > 1\} \end{aligned}$$

$$\Leftrightarrow \{x = 2\} \vee \{x > 2\}$$
$$\Leftrightarrow \{x \geqslant 2\}$$

As a result, as long as the program C starts in an initial state satisfying $x \geqslant 2$, it will terminate in a state satisfying $F = \{x > 2\}$. However, a state where x evaluates to 3 would satisfy F, but it is never reachable: assume the else-branch was discharged, then x must evaluate to 2 initially, so else-branch would not be discharged; otherwise assume the if-branch was discharged, then x mush initially evaluate to 1, but then the if-branch would not be discharged. Hence, any state where x evaluates to 3 is unreachable, but still satisfies the desired postcondition F. Such states are denoted as orange circles in Figure 4.1. Similarly, sp.C.F can also be satisfied by initial states that can lead to non-termination.

Attempting to establish a relation for line (**), we first start from the left side. Assume preconditions G satisfies $wlp.C.F \implies G$, we can end up in a situation like Figure 4.2.

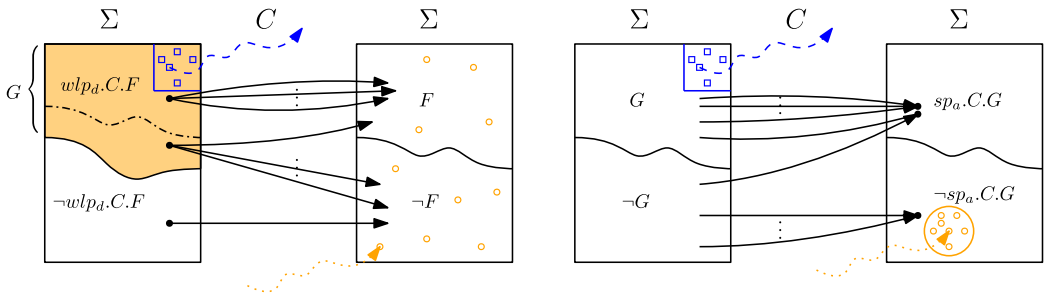


Figure 4.2: sp.C.G where G Is Overapproximation of wlp.C.F

We see on the left side of Figure 4.1 that G "includes" all the blue squares, and F only some of the orange circles. Now what do we know about sp.C.G? We see on the right side of Figure 4.1, sp.C.G is only satisfied by reachable states, i.e. none of the orange circles. This means that in general, without further constraints, we at least have that $F \not\Longrightarrow$ sp.C.G, since F may be satisfied by unreachable states, but sp.C.G is only satisfied by reachable states.

Can we then have a relation like $sp.C.G \implies F$? The answer is no. Assume we have both $wlp.C.F \implies G$, and $sp.C.G \implies F$, then with the Galois connection mentioned in line (*), we can conclude that $G \implies wlp.C.F$, which means that $G \Leftrightarrow wlp.C.F$, G must be exactly the weakest liberal precondition of F. Since we can only assume that G overapproximates wlp, we can not draw the conclusion that $sp.C.G \implies F$. As a result, from $wlp.C.F \implies G$ we can neither conclude $sp.C.G \implies F$ nor $F \implies sp.C.G$.

Analogously, assume $F \implies sp.C.G$, what can we state about wlp.C.F? From the left side of Figure 4.3, we can read that F does not "include" orange circles, i.e. F is only satisfied by reachable states. What do we know then about the relation between wlp.C.F and G? We see from Figure 4.3, wlp.C.F "includes" all the blue squares, but G only some of them. Consequently, we can not conclude that $wlp.C.F \implies G$. Also, we can not conclude that $G \implies wlp.C.F$, because this would require $sp.C.G \implies F$. Together with the assumption $F \implies sp.C.G$,

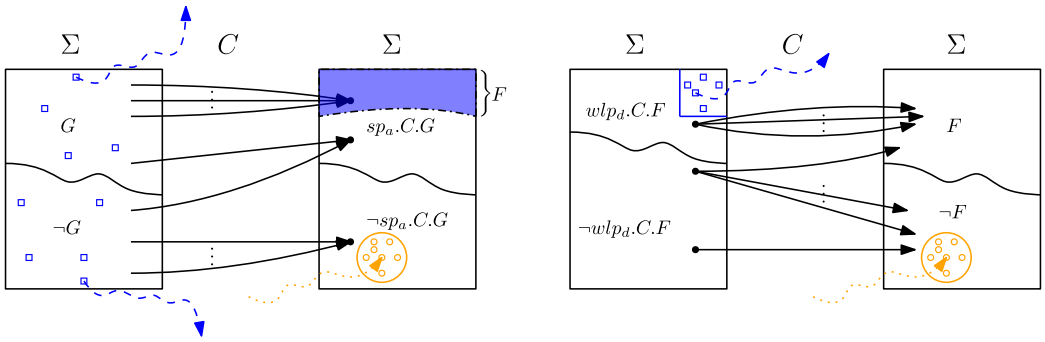


Figure 4.3: *wlp*.C.F where F Is Underapproximation of *sp*.C.G

this means that F must be exactly the strongest postcondition of program C w.r.t. precondition G, also a luxury we can not assume.

Summarizing the two conclusions above together, we can say that in general, without further constraints:

- From *wlp*.C.F $\implies$ G, we can neither conclude F $\implies$ *sp*.C.G nor *sp*.C.G $\implies$ F;
- Analogously, from F $\implies$ *sp*.C.G we can neither conclude *wlp*.C.F $\implies$ G nor G $\implies$ *wlp*.C.F.

This tells us that we can not overapproximate wlp by underapproximating sp, at least not directly.

4.2 THE GENERAL CASE

As hinted before, in our triple *wlp*.C.F $\implies$ G, G overapproximates wlp. It can take different forms: 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 be 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 ¬G will terminate, and the final state can satisfy ¬F:

$$\begin{aligned} wlp.C.F \implies G &\Leftrightarrow \neg G \implies \neg wlp.C.F \\ &\Leftrightarrow \neg G \implies wp.C.\neg F \qquad \text{| Theorem 2.12} \end{aligned}$$

These equivalences indicate that ¬G {C} ¬F is a valid Hoare triple with respect to total correctness, i.e. if C starts in an initial condition satisfying ¬G, then its execution must terminate, and it must be able to terminate satisfying ¬F.

4.2.1 *Overapproximation of wlp*

In Section 2.5 we define the weakest liberal precondition and state that it characterizes all the preconditions starting from which the program either diverges or will terminate in a state satisfying F. We are certain to use "will" instead of

"can" as the non-deterministic choice is viewed as demonic, so the behavior of wlp can be depicted by Figure 3.1.4.

The rectangle on the left (right) denote the set of all initial (final) states Σ. C is the program, and its executions are depicted by arrows. The rectangle is divided into halves, the part with label $wlp.C.F$ denotes all the states that satisfy $wlp.C.F$, and so on. The dashed arrow starting from inside a rectangle, ending outside neither rectangle denotes executions that do not terminate. The arrows that share the same point from the left rectangle denote different executions starting from the same initial state.

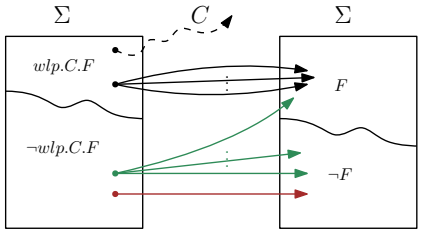
Starting from one initial state, the execution of the program can take various forms: it can be strictly deterministic, or it can have multiple non-deterministic choices; it can terminate in a final state, or it can execute forever. We categorize the executions of the program in four ways:

1. the dashed arrow means non-terminating executions;
2. the black arrows are executions starting from an initial state satisfying $wlp.C.F$ and if they terminate, then 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 are always possible to 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 $\neg F$.

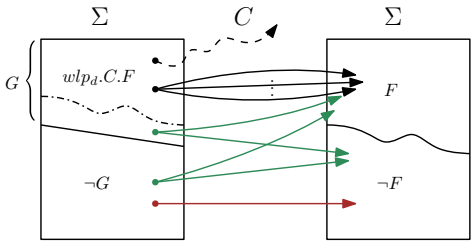
If we were to weaken the precondition, it can happen in various ways as shown in Figure 3.1.4-9. For example, Figure 3.1.4 shows the situation where G is exactly $wlp.C.F$. But in Figure 4.4.2, G is strictly weaker than $wlp.C.F$. Here, G is satisfied by all initial states that satisfy $wlp.C.F$, plus some of the initial states, starting from which the execution can end satisfying F or $\neg F$, aka some of the green arrows. Further, the G in Figure 4.4.3 is satisfied by all the above initial states, i.e. all the green arrows start from G. The same principle applies to the red arrows. Figure 4.4.4 shows a G that is satisfied additionally by some of the initial states that are starting points of the red arrows, i.e. executions that start in $\neg wlp.C.F$, and end in $\neg F$. In Figure 4.4.5, G contains all such initial states.

The situation can be trickier, for example in Figure 4.4.6, G is also the starting point of both green and red arrows. In other words, if C starts from some initial state satisfying this G, then maybe its execution diverges, maybe it terminates satisfying F, maybe not. The same can be said about the G in Figure 4.4.7 and Figure 4.4.8. Finally, G is weakened so much that it is simply $true$, hence satisfied by all initial states as seen in Figure 4.4.9. We see that there are a lot of possibilities on how to overapproximate wlp. Furthermore, the analysis shown in Figure 4.4 is not accounting for the gray arrows depicted in Figure 2.4.4, the addition of which would make the number of cases explode even more.

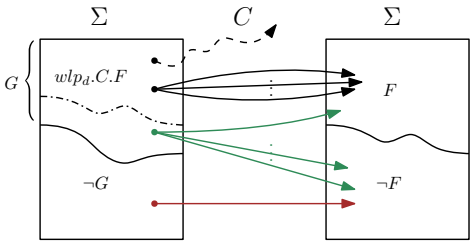
In general, when $wlp.C.F \implies G$, G can be satisfied by all possible initial states, which can be the starting points of black, green, red, or dashed arrows,



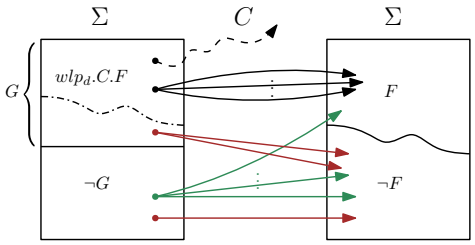
(1) Weakest liberal precondition (demonic non-determinism)



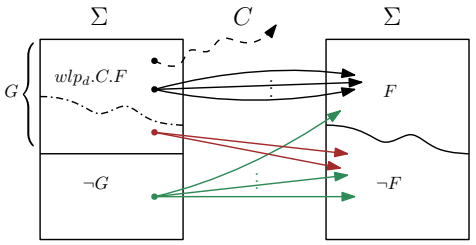
(2) Precondition G with wlp.C.F $\implies$ G and G contains some green arrows



(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 4.4: Case Distinction of Preconditions Weaker Than wlp

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.

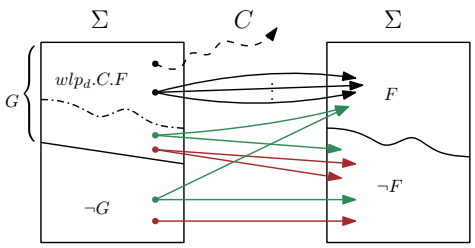
However, we can see from Figure 3.1.4-9 that when ¬G is not empty, there is always some arrow ending in ¬F. So there always exists some execution that starts from an initial state satisfying ¬G, and ends in a final state satisfying ¬F

Alternatively, we can say that 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

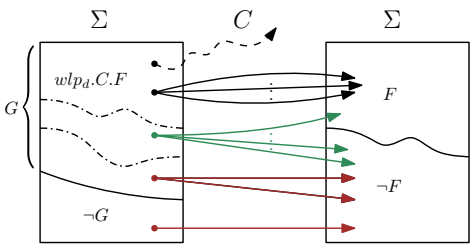
- its executions terminate, and
- there exists an execution that ends up in a final state that satisfies ¬F.

This again states 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 4.1.

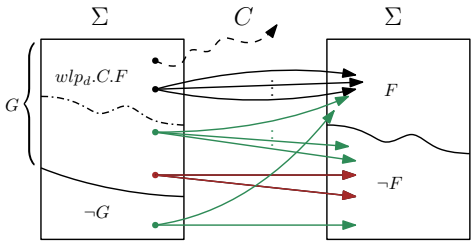
```
1        ... // starting device
```



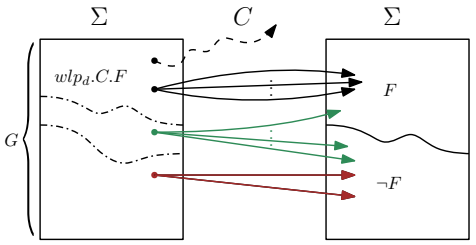
(6) Precondition G with $wlp.C.F \implies$ G and G contains some green arrows and some red arrows



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

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

Listing 4.1: Door with Sensors Counting Number of People Present

4.2.2 *Example: Door with Sensors*

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 4.1. 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 4.2: 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 4.5 We can see that with wlp, it is not enough to identify all preconditions

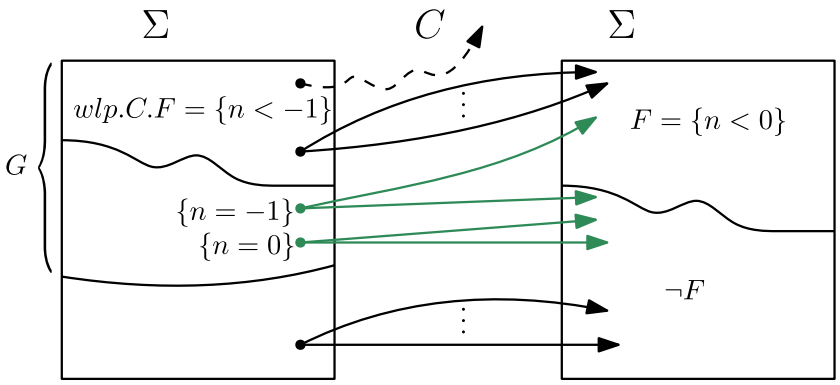


Figure 4.5: $wlp.C.F = \{n < -1\}$

that can result in errors. In other words, the green arrows in Figure 4.5 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 4.6, the hidden knowledge that seasoned users possess.

This demonstrates overapproximating wlp is useful to include the green and red arrows in Figure 4.6, aka when:

- some preconditions can but are not guaranteed to lead to the erroneous final states non-deterministically;

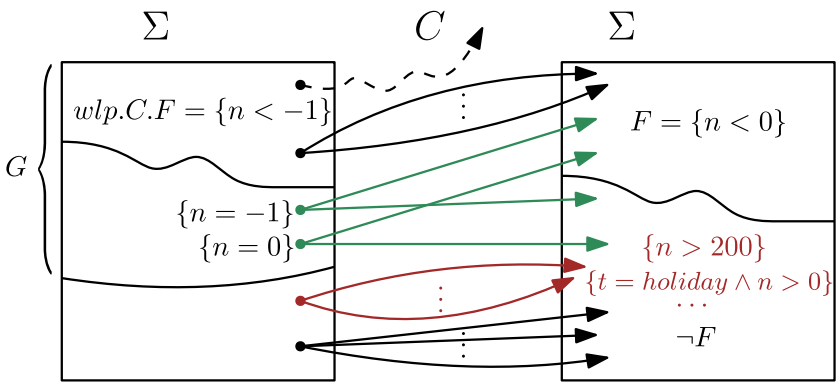


Figure 4.6: wlp.C.F $\implies$ G

- 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.

4.2.3 *Example: Including Good Runs*

Another reason one might wish to overapproximate the weakest liberal precondition is that wlp might be so strict that some good runs are excluded. Cousot et al. discovered patterns in System.dll that demonstrates this effect [4].

```
1       ...
2       for (int i = 0; i <= a.length; i++){
3               a[i] = ...f(a[i])...;
4               if (NonDet()) return;
5       }
6       ...
```

Listing 4.3: Good Runs Excluded

In this example in Listing 4.3, a[i] is an array, the function NonDet() is a nondeterministic function that returns boolean values. At run time, an out-of-bound error may or may not appear at line 3, depending on what happened before and after f(a[i]), the value of i may or may not be modified, or the array a may or may not be changed. To achieve out-of-bound-error-free, the weakest (liberal) precondition of this program snippet would be false: If the NonDet() function in line 4 returns false for some times, there is no guarantee that error will not appear at line 3.

We can see from Figure 4.7 that wlp.C.F empty, and it rules out some good runs like the green ones. However, G = a.length > 0 is an overapproximation over wlp, and it includes some good runs as indicated by the green arrows. These green arrows indicate runs where Nondet() returns false for some times,

so the program continues to run, but returning `false` and ending the program execution before an out-of-bound could happen. From the same initial state, we can be unlucky and NonDet() returns `false` for some more times, so that line 3 is executed for a couple of times more, and out-of-bound error happened, indicated by the red arrows in Figure 4.7.

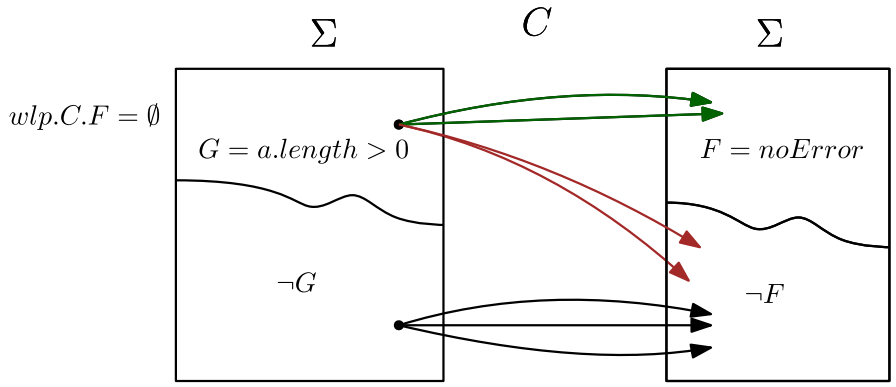


Figure 4.7: Overapproximating wlp to Include Good Runs

This example tells us that the weakest (liberal) precondition might be too strict at times, and it rules out good runs of program execution. By overapproximating wlp, we end up with a more relaxed precondition, starting from which even though bad runs are still possible, we still capture some good executions that would otherwise be excluded.

4.2.4 *Example: To Exclude Bugs*

The necessary liberal precondition can also be regarded as a method to rule out bugs

4.2.5 *Proof System*

We call the overapproximation of wlp necessary liberal precondition, "necessary" because it is required to ensure avoidance of erroneous final states, and "liberal", because it is satisfied by all initial states that can potentially lead to non-termination, just like wlp. We provide a proof system in Table 4.1. We use $\langle \mathsf{G} \rangle\ \mathsf{C}\ \langle \mathsf{F} \rangle$ to denote that G is a valid necessarily liberal precondition for program C given postcondition F.

The rules are mostly unsurprising. Remember that we would like to construct rules to prove that a precondition is the necessary liberal precondition, it should overapproximate the weakest liberal precondition, i.e. a triple $\langle \mathsf{G} \rangle\ \mathsf{C}\ \langle \mathsf{F} \rangle$ should satisfy:

$$\mathit{wlp}.\mathsf{C}.\mathsf{F} \implies \mathsf{G}$$

Consequently, we have the *diverge* axiom: because $\mathit{wlp}.\mathit{diverge}.\mathsf{F} = \mathit{true}$, and *true* is the only condition that overapproximates it, so the necessary liberal precondition is *true*.

Consequently, we have the skip axiom: because $wlp.skip.F = F$, then F overapproximates it. Alternatively, we can construct the rule in such way:

$$\frac{F \Longrightarrow G}{\langle G \rangle \text{ skip } \langle F \rangle}\ \text{skip}_{alt}$$

But this rule can be derived using the skip rule and the conseq in Table 4.1, so we choose the simpler skip rule. The same principle applies to the assign, seq, if, and the choice rule: we keep the rules simple by not discussing overapproximation directly, but let the conseq rule be concerned with relaxation.

The consequence rule says that we are allowed to weaken the precondition and strengthen the postcondition:

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

Intuitively, if a precondition P already overapproximates the weakest liberal precondition, then by overapproximating it with G, the result is still an overapproximation of wlp:

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

Conversely, we can strengthen the postcondition from Q to F, because $wlp.C$ is monotonic (2.13), then given $F \implies Q$ we can derive:

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

Hence, G is still an overapproximation of the stronger $wlp.C.F$.

Note that the conseq rule is the complete opposite direction from the "Rules of Consequence" in Hoare logic that we referred to in Table 2.2. Writing the consequence rule of Hoare logic in the same natural deduction style, we arrive at the following rule labeled with conseq_h:

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

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:

$$\begin{aligned}
& wlp.C.F \Longrightarrow G \\
\Leftrightarrow\ & \neg G \Longrightarrow \neg wlp.C.F \\
\Leftrightarrow\ & \neg G \implies wp.C.\neg F && \mid \text{Theorem 2.12} \\
\Leftrightarrow\ & \neg G\ \{C\}\ \neg F && (*)
\end{aligned}$$

Note that the last line is an implication in both directions, because we take the version of Hoare triple regarding total correctness, i.e. supplementing the rule to prove termination of while loops [21]:

$$\frac{P \wedge \varphi \wedge v \in \mathbb{N} \wedge v = n \quad \{C\} \quad P \wedge v \in \mathbb{N} \wedge v < n}{P \wedge v \in \mathbb{N} \quad \{\text{while } (\varphi) \text{ do } C\} \quad \neg\varphi \wedge P \wedge t \in \mathbb{N}}\ \text{while}_h$$

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

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

$$\frac{\langle \varphi \wedge G\rangle\ C_1\ \langle F\rangle,\ \langle \neg\varphi \wedge G\rangle\ C_2\ \langle F\rangle}{\langle G\rangle\ \mathrm{IF}\ \langle F\rangle}\,\mathrm{if}$$

$$\frac{\langle P\rangle\ C\ \langle \varphi \vee P\rangle}{\langle P\rangle\ \mathrm{WHILE}\ \langle \varphi \vee P\rangle}\,\mathrm{while}$$

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

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

Table 4.1: The Proof System

This is necessary to give intuition of the consequence rule in our proof system. Assuming that the triple $\langle\cdot\rangle\ \cdot\ \langle\cdot\rangle$ is sound (which we will prove later in this section), i.e. $\langle P\rangle\ C\ \langle Q\rangle \implies (\mathit{wlp}.C.F \Longrightarrow G)$, we can derive the conseq rule from consqe_h :

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

It is however not trivial to construct the while rule. Assume we would like to find some G to overapproximate $\mathit{wlp}.\mathrm{WHILE}.F$, unfolding according to the definition in Table 2.5:

$$\begin{aligned}\mathit{wlp}.\mathrm{WHILE}.F &= \mathrm{gfp}\ X.\Phi(X)\\ &= \mathrm{gfp}\ X.\ (\neg\varphi \wedge F) \vee (\varphi \wedge \mathit{wlp}.C'.X)\end{aligned}$$

Since G overapproximates $\mathit{wlp}.\mathrm{WHILE}.F$, the greatest fixed point of Φ, then G must also overapproximate any fixed point of Φ: For any fixed point P of Φ, it is valid that $P \implies \mathrm{gfp}\ \Phi$, add the fact that $\mathrm{gfp}\ \Phi \implies G$, we can conclude that $P \implies G$. In other words, the weakest liberal precondition G must overapproximate all fixed points of Φ:

$$\begin{aligned}&\forall P.P = \Phi(P) : P \overset{!}{\Longrightarrow} G\\ \Leftrightarrow\ &\forall P.P = (\neg\varphi \wedge F) \vee (\varphi \wedge \mathit{wlp}.C'.X) : P \overset{!}{\Longrightarrow} G\\ &= \mathrm{gfp}\ X.\ (\neg\varphi \wedge F) \vee (\varphi \wedge \mathit{wlp}.C'.X)\end{aligned}$$

Lemma 4.17 *[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 support taking these rules instead of others?

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

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

□

Lemma 4.18 *[The proof system is incomplete] here*

Proof. here □

4.3 A SPECIAL CASE

As promised before, we will look into a way to identify the green arrows in Figure 4.5, 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 4.4.3, 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 4.4.3, 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. This behavior describes exactly the behavior of wlp with angelic non-determinism, which is more obvious if we put together Figure 4.4.3 and Figure 3.1.3:

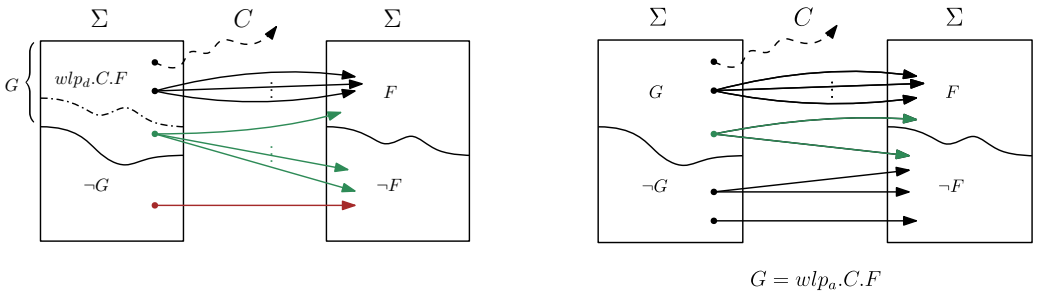


Figure 4.8: Comparing G With Angelic wlp

Unfortunately, we can not use either big-step semantics in Section 2.7 or collecting semantics in Section 2.8 to describe the intended semantics of wlp with angelic resolution of non-determinism. The reason is that both semantics can not

easily distinguish between executions of programs like $C_1 \square \mathit{diverge}$ and skip, but angelic wlp would require the distinction. Fixing an initial state σ, with big-step semantics, we can only state about the program $C_1 \square \mathit{diverge}$ that σ can take a big step to whichever state the program C_1 can take it to:

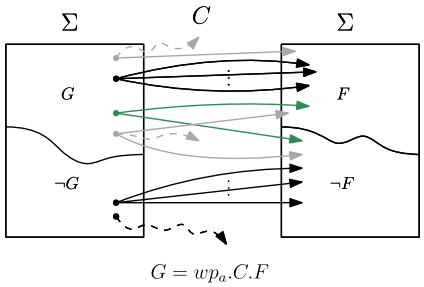
$$\frac{\sigma \xrightarrow{C_1} \tau}{\sigma \xrightarrow{C_1 \square \mathit{diverge}} \tau}\ \mathit{choice}_1$$

Hence, given a transition where $\sigma C \tau$, we do not know whether a big step towards τ is the only step σ can take with program C, or it is one of the steps, and another possible step leads to non-termination.

Similarly, with collecting semantics, we have that

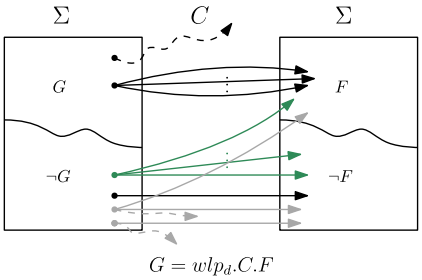
$$[\![C_1 \square \mathit{diverge}]\!]\{\sigma\} = [\![C_1]\!]\{\sigma\} \cup [\![\mathit{diverge}]\!]\{\sigma\} = [\![C_1]\!]\{\sigma\} \cup \emptyset = [\![C_1]\!]\{\sigma\}$$

Hence, the collecting semantics of $C_1 \square \mathit{diverge}$ and C_1 are the same. For the above reasons, we attempt to capture the intended behavior of wlp with angelic non-determinism (denoted by wlp_a) with case distinction of different groups of executions with help of graphical illustration. We show the graphs of wp, wlp, wlp_a, and sp together in Figure 4.9.



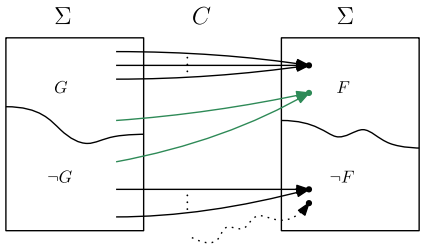
$G = wp_a.C.F$

(1) Weakest precondition (angelic non-determinism)



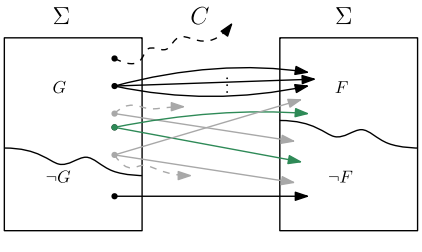
$G = wlp_d.C.F$

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



$F = sp_a.C.G$

(3) Strongest postconditoin (angelic non-determinism)



$G = wlp_a.C.F$

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

Figure 4.9: wp_a, wlp_d, sp_a, wlp_a Together

To help with reasoning, given program C and postcondition F, we consider its executions and categorize the initial states into the following disjoint sets:

- S_{FF}: the set of all initial states where starting from any state in this set, all executions terminate, and the final states satisfy F.

- $S_{\neg F \neg F}$:the set of all initial states, where starting from any state in this set, all executions terminate, and the final states satisfy $\neg F$.
- $S_{F \neg F}$: the set of all initial states, where starting from any state in this set, at least one execution diverges, at least one execution terminates and the final state satisfies F, and at least one execution terminates and the final state satisfies $\neg F$.
- S_{II}: the set of all initial states, where starting from any state in this set, all executions diverge. The subscript I stands for "infinite".
- S_{IF}: the set of all initial states, where starting from any state in this set, at least one execution diverges, and at least one execution terminates, and the final state satisfies F.
- $S_{I \neg F}$: the set of all initial states, where starting from any state in this set, at least one execution diverges, and at least one execution terminates, and the final state satisfies $\neg F$.
- $S_{IF \neg F}$: the set of all initial states, where starting from any state in this set, at least one execution diverges, at least one execution terminates and the final state satisfies F, and at least one execution terminates and the final state satisfies $\neg F$.

Similarly, given program C and precondition G, we can categorize the final states into the following disjoint sets:

- T_U: the set of all unreachable final states.
- T_{GG}: the set of all reachable final states, where all executions leading to it originally started in an initial state satisfying G.
- $T_{\neg G \neg G}$: the set of all reachable final states, where all executions leading to it originally started in an initial state satisfying $\neg G$.
- $T_{G \neg G}$: the set of all reachable final states, where looking at any of them, there exists at least one execution reaching this final state originated in an initial state satisfying G, and there exists at least one execution reaching this final state originated in an initial state satisfying $\neg G$.

Studying Figure 4.9, we see that the states satisfying wp with angelic resolution of non-determinism can be characterized as: Starting from this state, termination in a final state satisfying F is always possible. More concretely, starting from an initial state satisfying $wp_a.C.F$, there can exist executions where it diverges, or terminates in a final state satisfying $\neg F$, but there must exist at least one execution where it terminates, and the final state satisfies F. As a result, We can then characterize $wp_a.C.F$ as the following:

$$wp_a.C.F = S_{FF} \cup S_{F \neg F} \cup S_{IF} \cup S_{IF \neg F}$$

Similarly, the states satisfying $wlp_d.C.F$ can be characterized as: Starting from any of these states, either all executions lead to final states satisfying F upon termination, or non-termination is guaranteed. Consequently,

$$wlp_d.C.F = S_{FF} \cup S_{II} \cup S_{IF}$$

The states satisfying $sp_a.C.G$ can be characterized as: There must exist an execution from an initial state satisfying G that reaches this final state.

$$sp_a.C.G = T_{GG} \cup T_{GnG}$$

The states satisfying $wlp_a.C.F$ should behave like the following: Starting from any of these states, either non-termination is guaranteed, or there must be an execution with final state satisfying F, or some of the executions diverge, and others terminate in a final state satisfying $\neg F$. As a result,

$$wlp_a.C.F = S_{FF} \cup S_{FnF} \cup S_{IF} \cup S_{InF} \cup S_{IFnF} \cup S_{II} = \Sigma - S_{nFnF}$$

We propose that we can find wlp_a by using the necessary liberal precondition and the strongest postcondition. While trying to distinguish what is and is not in wlp_a, we approach it in two directions: first, we find restraints that make the necessary liberal precondition G where $wlp.C.F \implies G$ an overapproximation of angelic wlp ($wlp_a.C.F \implies G$); then we find restraints that makes G an underapproximation of angelic wlp. Finally, we combine both directions and conclude with statements that make G and $wlp_a.C.G$ equivalent. We also gain a side effect of expressing wlp_a without having to define it.

Lemma 4.19 *[Angelic wlp implies G]*

$$\textit{if} \quad (wlp_d.C.F \implies G) \wedge (sp_a.C.\neg G \implies \neg F) \quad \textit{then} \quad wlp_a.C.F \implies G$$

The second prerequisite $sp.C.\neg G \implies \neg F$ states that from $\neg G$ we are only allowed to reach $\neg F$, making sure that all green arrows as in Figure 4.4 are included in G.

Proof. Rewriting the statements above, we get:

$$\begin{aligned}
wlp_d.C.F \implies G &\Leftrightarrow S_{FF} \cup S_{II} \cup S_{IF} \implies G && \text{(a)}\\
sp_a.C.\neg G \implies \neg F &\Leftrightarrow T_{nGnG} \cup T_{GnG} \implies \neg F \\
&\Leftrightarrow F \implies \neg(T_{nGnG} \cup T_{GnG}) \\
&\Leftrightarrow F \implies \Sigma - (T_{nGnG} \cup T_{GnG}) \\
&\Leftrightarrow F \implies T_{GG} \cup T_U && \text{(d)}\\
wlp_a.C.F \implies G &\Leftrightarrow S_{FF} \cup S_{FnF} \cup S_{IF} \cup S_{InF} \cup S_{IFnF} \cup S_{II} \implies G && \text{(e)}
\end{aligned}$$

To prove line (e), comparing line (a) and line (e) we see that we only need to prove that

$$S_{FnF} \cup S_{InF} \cup S_{IFnF} \implies G$$

From line (d) we see that final states in F are either unreachable from any initial state, or can only be reached via executions starting from initial states satisfying $\neg G$. Since T_U and $T_n G n G$ are disjoint, there are two separate cases. The first case is that $F \implies T_U$, then for any state $\tau \in F$, we know that there exists no execution that has τ as final state, so all $S\_$ with "there exists at least one execution that terminates in a final state satisfying F" as part of their descriptions are empty, i.e. $S_{FnF}, S_{IF}, S_{IFnF}$. Hence, in this case, we have that $S_{FnF} \cup S_{InF} \cup S_{IFnF} \implies G$.

The second case is that $F \implies T_{GG}$, this means that for any state $\tau \in F$, there exists and only exists executions leading to τ that start from an initial state satisfying G. This means that any $S\_$ with "there exists at least one execution that terminates in a final state satisfying F" must be a subset of G. Consequently, $S_{FnF} \cup S_{InF} \cup S_{IFnF} \implies G$ is valid in this case as well. □

Now that we can use G to overapproximate wlp_a, the logcial next step is to underapproximate it, so that in the final step we can establish equivalences between G and angelic wlp:

Lemma 4.20 *[G implies angelic wlp]*

$$\textit{if} \quad (P \implies G) \implies \neg(sp.C.P \implies \neg F) \quad \textit{then} \quad G \implies wlp_a.C.F$$

Here, the prerequisite states that looking at any predicate P stronger than G (or any subset of G, if we view predicates as sets of all states that satisfy them), there are no executions reaching a final state satisfying $\neg F$. Remember that $sp.C.P$ is exactly satisfied by states that can be reached starting from an initial state in P, recalling Figure 3.4.1. The implication $sp.C.P \implies \neg F$ means that from P, we can only reach states satisfying $\neg F$. By negating this condition, we assert that starting from a state satisfying P, we should be able to reach final states other than those satisfying $\neg F$.

Since P can be singleton, i.e. a condition that is satisfied by only one state σ, then from σ we are not allowed to only have executions that end in a state satisfying $\neg F$. Consequently, 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 4.4.

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

$$\begin{aligned}
& P \implies G \implies \neg(sp.C.P \implies \neg F) \\
\Leftrightarrow\ & P \implies G \implies \neg(\forall \tau \in \Sigma : \tau \in sp.C.P \implies \tau \in \neg F) \\
\Leftrightarrow\ & P \implies G \implies \exists \tau \in \Sigma : \neg(\tau \in sp.C.P \implies \tau \in \neg F) \\
\Leftrightarrow\ & P \implies G \implies \exists \tau \in \Sigma : \neg(\neg \tau \in sp.C.P \ \vee\ \tau \in \neg F) \\
\Leftrightarrow\ & P \implies G \implies \exists \tau \in \Sigma : \tau \in sp.C.P \ \wedge\ \neg(\tau \in \neg F) \\
\Leftrightarrow\ & P \implies G \implies \exists \tau \in \Sigma : \tau \in sp.C.P \ \wedge\ \tau \in F \\
\Leftrightarrow\ & P \implies G \implies \exists \tau \in \Sigma : \tau \in (T_{PP} \cup T_{PnP}) \ \wedge\ \tau \in F \\
\Leftrightarrow\ & \sigma \in P \implies \sigma \in G \implies \exists \tau \in \Sigma : \tau \in (T_{PP} \cup T_{PnP}) \ \wedge\ \tau \in F \qquad \text{(f)}
\end{aligned}$$

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

$$\begin{aligned} & G \implies wlp_a.C.F \\ \Leftrightarrow\ & \sigma \in G \implies \sigma \in wlp_a.C.F \\ \Leftrightarrow\ & \sigma \in G \implies \sigma \in \Sigma - S_{nFnF} \\ \Leftrightarrow\ & \sigma \in G \implies \sigma \in (S_{FF} \cup S_{FnF} \cup S_{IF} \cup S_{InF} \cup S_{IFnF} \cup S_{II}) \end{aligned} \tag{g}$$

If G is the empty set, then our goal is automatically fulfilled. Now we can assume that G is not empty. Looking at an arbitrary $\sigma \in G$, we can always construct set $P = \{\sigma\}$, and consequently, the prerequisites in line (f) holds. As a result, the conclusion in line (f) holds: $\exists \tau \in \Sigma : \tau \in (T_{PP} \cup T_{PnP}) \ \wedge \ \tau \in F$

Now we can find witnesses τ such that

$$\tau \in (T_{PP} \cup T_{PnP}) \ \wedge \ \tau \in F$$

The first case is $\tau \in T_{PP}$. Remember that T_{PP} is the set of all reachable final states, where all executions leading to it originally started in an initial state satisfying P. Since $P = \{\sigma\}$, P is a singleton set, from $\tau \in T_{PP}$ we can then conclude that τ can only be reached from σ. This tells us that there exists an execution from σ to τ, where τ satisfies F. This tells us that σ can be an element of all $S_\_$ except from S_{II} and S_{nFnF}, which means that the conclusion of line (g) is true. Since we take an arbitrary $\sigma \in G$, the complete line (g) is then true.

The second case is $\tau \in T_{PnP}$. Remember that $\tau \in T_{PnP}$ means that there exists at least one execution from an initial state satisfying P, and at least one from $\neg P$. Again, we can draw the conclusion that there exists an execution from σ to τ, since P is a singleton set. From here, we follow exactly the same reasoning of the first case, and can draw the conclusion that line (g) is valid. □

Corollary 4.21 *[G equivalent to angelic wlp]*
if $wlp.C.F \implies G \ \wedge \ sp.C.\neg G \implies \neg F$ *and* $P \implies G \implies \neg(sp.C.P \implies \neg F)$
then $G = wlp_a.C.F$

With this corollary, we can identify all initial states that lead to executions terminating satisfying our given postcondition F, or its opposite. By setting the necessary liberal precondition G to be equal to $wlp_a.C.F$, we make sure that G is satisfied by all such states. If we think F to be the specification for erroneous termination, then his means that G is concerned with all problematic initial states, and it is necessary to avoid G to steer clear of those errors.

4.4 FROM NECESSARY PRECONDITION TO NECESSARY LIBERAL PRECONDITION

To reason about correctness, it is sometimes helpful to first reason about partial correctness, then supplement with termination proof to reach total correctness. For example, when Hoare triple was first proposed [11], it is in form of the sufficient precondition that guarantees partial correctness:

$$H\ \{C\}\ F \ \Leftrightarrow \ H \implies wlp.C.F$$

Then Manna and Pnueli [21] provided a termination proof for while-loops in Hoare triple, so that the extended Hoare Triple is a sufficient precondition that guarantees total correctness:

$$H' \{C\} F \Leftrightarrow H' \Longrightarrow wp.C.F$$

Explicitly, to arrive at $H \Longrightarrow wp.C.F$, we can first investigate $H \Longrightarrow wlp.C.F$, then supplement it with a termination proof.

It stands to reason that while attempting to reason about the necessary liberal precondition, we can take a similar approach but from the opposite direction: starting from necessary precondition $wp.C.F \Longrightarrow G$, supplementing it with a termination or non-termination proof, we try to reach the necessary liberal precondition $wlp.C.F \Longrightarrow G$.

First, we attempt to find a necessary precondition G where $wp.C.F \Longrightarrow G$. For this reason, we propose an overapproximation of wp, denoted by wp_d^+ in Table 4.2. For reference, we also include the definitions of wp_d and wlp_d, and we leave the cells empty to indicate that the definition is analogous to that of wp_d^+.

C	**wp_d.C.F**	**wp_d^+.C.F**	**wlp_d.C.F**
skip		F	
diverge	false	false	true
$x := e$		$F[x/e]$	
$C_1; C_2$		$wp_d^+.C_1.(wp_d^+.C_2.F)$	
$\{C_1\}\Box\{C_2\}$		$wp_d^+.C_1.F$ $\wedge wp_d^+.C_2.F$	
if (φ) $\{C_1\}$ else $\{C_2\}$		$(\varphi \wedge wp_d^+.C_1.F)$ $\vee(\neg\varphi \wedge wp_d^+.C_2.F)$	
while (φ) $\{C'\}$	lfp X. $(\neg\varphi \wedge F)$ $\vee(\varphi \wedge wp_d.C'.X)$	fp X. $(\neg\varphi \wedge F)$ $\vee(\varphi \wedge wp_d^+.C'.X)$	gfp X. $(\neg\varphi \wedge F)$ $\vee(\varphi \wedge wlp_d.C'.X)$

Table 4.2: wp_d^+, An Overapproximation of wp

We can see from Table 4.2 that the only difference wp_d^+ is defined very close to wp_d and wlp_d, aside from the different resolution for non-determinism and treatment for non-termination, the most significant difference is the definition for while-loops: with wp_d^+, we look for a fixed point of the characteristic function, instead of the least fixed point with wp and the greatest fixed point with wlp. The computation of some fixed point is simpler than the computation of the least or the greatest fixed points. There is a repertoire of fixed point combinators, with which we can calculate a fixed point of the given function [3]. The choice of which fixed point to take is left to the user. This also tells us that wp_d^+ represents rather a class of precondition transformers, depending on the choice of the fixed point, one chooses one of the precondition transformers in wp_d^+. We can see from Table 4.2 that wp_d^+ overapproximates wp_d, and underapproximates wlp_d:

Lemma 4.22

$$\forall C \in \mathcal{C}.\forall F \in \Sigma : wp_d.C.F \implies wp_d^+.C.F$$

Proof. By structural induction. □

Lemma 4.23

$$\forall C \in \mathcal{C}.\forall F \in \Sigma : wp_d^+.C.F \implies wlp_d.C.F$$

Proof. By structural induction. □

The semantics of wp_d^+ is captured in Figure 4.10. The area marked in dark yellow is wp_d, the areas with dark and medium yellow mark wp_d^+, and all three yellow areas together represent wlp_d. Remember from Theorem 2.12 that $wlp_d.C$ and $wp_a.C$ are each other's conjugates, i.e. $wp_a.C.F = wlp_d.C.\neg F$, hence we can label the rest of the left rectangle with $wp_a.C.\neg F$.

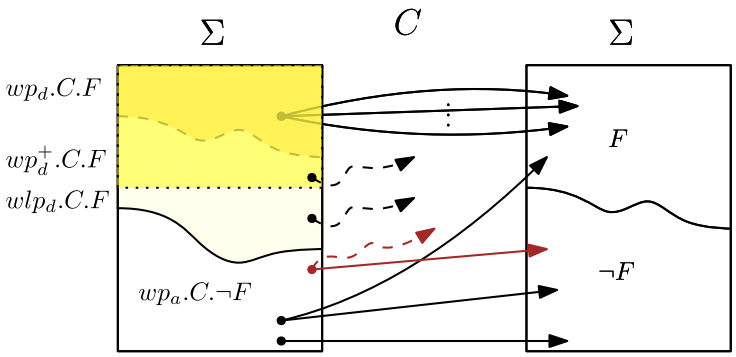


Figure 4.10: Graphical Illustration of wp_d^+

Remember that our goal is to overapproximate wlp_d, i.e. to find G such that $wlp.C.F \implies G$. Speaking in terms of Figure 4.10, it corresponds to finding an area larger than all three yellow areas, i.e. to find G where $wlp_d.C.F \implies G$, which is then equivalent to $\neg G \implies wp_a.C.\neg F$, as we see from Figure 4.10. This yields an alternative method to finding the necessary liberal precondition, given program C and postcondition F:

1. Calculate $wp_d^+.C.F$, where the discovery of any fixed point should be easier than the discovery of the least and the greatest one.
2. Negate the result: $P := \neg wp_d^+.C.F$.
3. Find $Q \implies P$ such that C always terminates starting from any initial states satisfying Q.
4. The negation of Q is then the necessary liberal precondition of program C with respect to postcondition F.

One might wonder, what is the difference between this method and directly underapproximating wp_a, i.e. proving the validity of a Hoare triple? The main

difference is that we are not concerned with the postcondition anymore. With this method, $\mathrm{wp}_{\mathrm{d}}^{+}$ is responsible for reasoning about the final states, and we are only concerned with termination proofs. By proving that all executions terminate, we identify the green area in Figure 4.11, which is strictly an underapproximation of wp_{a}, provided that the orange arrows in Figure 4.11 do exist, and that the negation of the green area is an overapproximation of $\mathrm{wlp}_{\mathrm{d}}$.

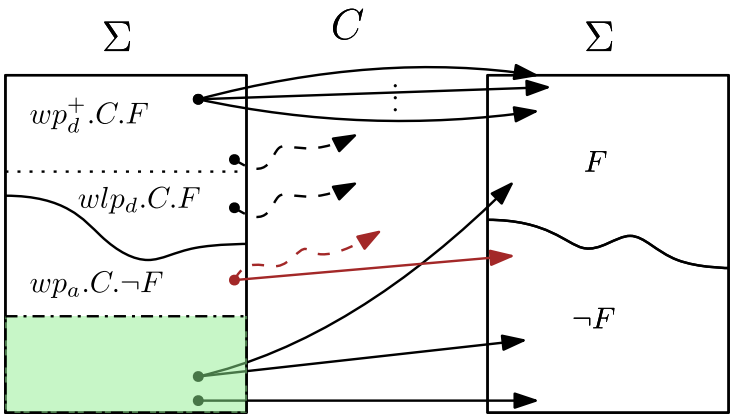


Figure 4.11: Overapproximating $\mathrm{wlp}_{\mathrm{d}}$ Using $\mathrm{wp}_{\mathrm{d}}^{+}$

Unfortunately, proving termination is not trivial. In fact, given a program C, and a condition P, finding a Q where $\mathrm{Q} \implies \mathrm{P}$, such that the execution of C starting in an initial state satisfying Q is in its essence the halting problem, which is undecidable. At this point, we can not provide an algorithm to always find such Q.

However, the use of $\mathrm{wp}_{\mathrm{d}}^{+}$ still helps us separate the reasoning about final states and termination: $\mathrm{wp}_{\mathrm{d}}^{+}$ lets us rule out executions not ending in a state satisfying F with less computational complexity, and we have freedom to choose a termination proof without having to be concerned with final states.

An interesting observation of $\mathrm{wp}_{\mathrm{d}}^{+}$ presents itself: remember from Figure 3.7 that partial correctness is indicated by $\mathrm{G} \implies \mathrm{wlp}_{\mathrm{d}}.\mathrm{C.F}$, and partial incorrectness by $\mathrm{wp}_{\mathrm{a}}.\mathrm{C.F} \implies \mathrm{G}$. We know from Lemma 4.23 that

$$\mathrm{wp}_{\mathrm{d}}^{+}.\mathrm{C.F} \implies \mathrm{wlp}_{\mathrm{d}}.\mathrm{C.F}$$

taking its contrapositive, we get $\neg \mathrm{wlp}_{\mathrm{d}}.\mathrm{C.F} \implies \neg \mathrm{wp}_{\mathrm{d}}^{+}.\mathrm{C.F}$. From Theorem 2.12 we know that $\mathrm{wp}_{\mathrm{a}}.\mathrm{C.F} = \neg \mathrm{wlp}_{\mathrm{d}}.\mathrm{C}.\neg \mathrm{F}$, hence

$$\neg \mathrm{wp}_{\mathrm{a}}.\mathrm{C}.\neg \mathrm{F} \implies \neg \mathrm{wp}_{\mathrm{d}}^{+}.\mathrm{C.F}$$

This tells us that $\mathrm{wp}_{\mathrm{d}}^{+}$ is valid for partial correctness, and $\neg \mathrm{wp}_{\mathrm{d}}^{+}$ is valid for partial incorrectness. This is not surprising, since partial correctness and incorrectness are each other's contrapositives, it still gives us a notion to represent the class of predicate transformers that are the "cut" between partial correctness and incorrectness triples, as shown in Figure 4.12.

this summary has to be changed To summarize this chapter, we started with the description of relations that connect the predicate transformers together, to

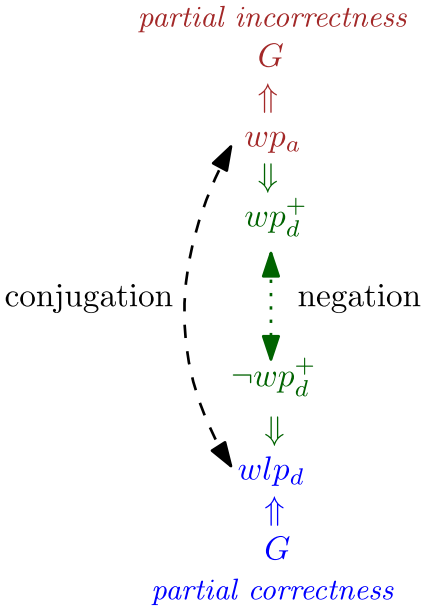


Figure 4.12: The Class of $\mathrm{wp}_{\mathrm{d}}^{+}$ Is the Cut Between Partial Correctness and Incorrectness

introduce the necessary liberal precondition. Then we discovered that G can be an indicator of "bad" precondition that is necessary to avoid, to successfully terminate. We can do it in two ways:

1. If we are sure that we captured all undesired final states in F, then we use the necessary liberal precondition that underapproximates (overapproximates) $\mathrm{wlp}_{\mathrm{a}}$, so that we catch as much "bad" initial states as possible.

2. If we only have limited knowledge of the erroneous final states, then we appeal to experts of the system for other potential errors in order to reach necessary liberal precondition.

5

CONCLUSIONS

5.1 CONCLUSIONS

In this thesis, we study the weakest liberal precondition transformer and its over-approximation: G such that $wlp.C.F \implies G$. We first discuss the definitions of while-loops in its original form [5] 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 constraints 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 from which 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 over-approximate 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 in Section 2.4.2, and give 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 in Chapter 3, which helps demonstrate the triples spawned by underapproximations and over-approximations of the predicate transformers.

3. We conclude in Section 4.2 that the necessary 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 also 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 in Section 4.2.5.

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 in Section 4.3 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.

5.2 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 $\mathtt{true}$ or $\mathtt{false}$. Albeit classic, it might be more interesting to examine the above results in a quantitative setting, where predicates evaluate to more than $\mathtt{true}$ or $\mathtt{false}$. 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. [10]

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] Felice Cardone and J Roger Hindley. "History of lambda-calculus and combinatory logic." In: *Handbook of the History of Logic* 5 (2006), pp. 723–817.

[4] Patrick Cousot, Radhia Cousot, Manuel Fähndrich, and Francesco Logozzo. "Automatic inference of necessary preconditions." In: *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer. 2013, pp. 128–148.

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

[6] 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.

[7] 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.

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

[9] 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).

[10] 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.

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

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

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

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

[15] 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.

[16] 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.

[17] Xavier Leroy and Hervé Grall. "Coinductive Big-Step Operational Semantics." In: *Information and Computation* 207.2 (Feb. 2009), pp. 284–304. ISSN: 08905401. DOI: 10.1016/j.ic.2007.12.004. (Visited on 04/08/2024).

[18] 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).

[19] 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.

[20] Johan J. Lukkien. "Operational Semantics and Generalized Weakest Preconditions." In: *Science of Computer Programming* 22.1-2 (Apr. 1994), pp. 137–155. ISSN: 01676423. DOI: 10.1016/0167-6423(94)90010-8. (Visited on 04/08/2024).

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

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

[23] 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.

[24] Keiko Nakata and Tarmo Uustalu. "Trace-Based Coinductive Operational Semantics for While." In: *Theorem Proving in Higher Order Logics*. Ed. by Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel. Vol. 5674. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 375–390. ISBN: 978-3-642-03358-2 978-3-642-03359-9. DOI: 10.1007/978-3-642-03359-9_26. (Visited on 04/08/2024).

[25] Tobias Nipkow. "Hoare Logics in Isabelle/HOL." In: *Proof and System-Reliability*. Ed. by Helmut Schwichtenberg and Ralf Steinbrüggen. Dordrecht: Springer Netherlands, 2002, pp. 341–367. ISBN: 978-1-4020-0608-1 978-94-010-0413-8. DOI: 10.1007/978-94-010-0413-8_11. (Visited on 04/02/2024).

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

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

[28] 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).

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

[30] 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).

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

[32] Linpeng Zhang and Benjamin Lucien Kaminski. *Quantitative Strongest Post*. 2022. arXiv: 2202.06765 [cs.LO].

[33] 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.