NECESSARY LIBERAL PRECONDITIONS:
A PROOF SYSTEM

Mster'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

Mster'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 is where the abstract goes.

ZUSAMMENFASSUNG

Kurze Zusammenfassung des Inhaltes in deutscher Sprache. . .

摘 要

这里是中文摘要 hi

CONTENTS

Part I

HOARE TRIPLES, WEAKEST PRECONDITIONS, WEAKEST LIBERAL PRECONDITIONS

Some text about this part.

1 BACKGROUND

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*" [7]. 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. Imagine being driven by an autonomous car. 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. Once we know that a program is totally correct, then we are sure that the sun rises tomorrow.

To know "for sure", we could verify programs using formal methods. One famous method is Hoare triples [6]. A Hoare Triple contains three parts: a precondition, a program, and a postcondition. They are written as such: $G \, \{C\} \, F$. It states that if the system starts in a state that satisfies the precondition, then the state after the execution of the program will satisfy the postcondition, provided that the program terminates. Hoare triples are elegant in that once we have appropriate preconditions, we can follow their reference rules on sequential programs with ease. But with Hoare triples in their original form, we know the program is correct, but we are not sure of its termination. This is called partial correctness.

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

Since then, a plethora of research projects blossomed and yielded fruitful results. This thesis aims to follow the steps of the predecessors and investigate the weakest liberal precondition transformer [4] (wlp), which gives preconditions that are necessary and sufficient so that the program either terminates correctly or never terminates, proving partial correctness.

We first introduce Hoare triples, the wp transformer, and the wlp transformer using the Guarded Command Language [3] to present programs in Chapter 2. We also explain their connections and differences.

Then we proceed to Chapter 3, analyzing various cases of $wlp.C.F \Longrightarrow G$, first focusing on a special case where G is equivalent to wlp with angelic nondeterminism, before proceeding to G in general. Finally, we summarize our conclusions in Chapter 4 and propose possible future work.

2 PRELIMINARIES

2.1 NOTATIONS

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

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

Table 2.1: Symbols and Notations

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

By default, we take $F(\sigma)$ to mean the same as $F(\sigma) = \mathrm{true}$ 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 [5, 10] 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* [6] 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 Benjamin L. Kaminski's dissertation [8], 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.

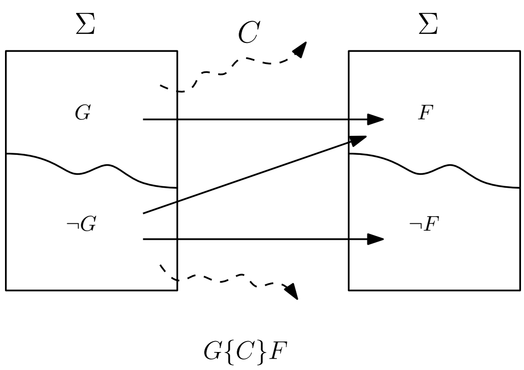


Figure 2.1: Valid Hoare Triple (Deterministic)

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 [9], where they introduced what we call 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 $\neg G$ to F in Figure 2.1, which is what Edsger W. Dijkstra accomplished with his weakest precondition transformer in 1975 [3], among other things.

2.3 GUARDED COMMAND LANGUAGE

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

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

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

$C ::=$	$x := e$	$\mid C; C$	$\mid \{C\}\Box\{C\}$	
	assignment	sequential composition	non-deterministic choice	
	$\mid \mathit{if}\ (\varphi)\ \{C\}\ \mathit{else}\ \{C\}$	$\mid \mathit{while}\ (\varphi)\ \{C\}$	$\mid \mathit{skip}$	$\mid \mathit{diverge}$
	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.

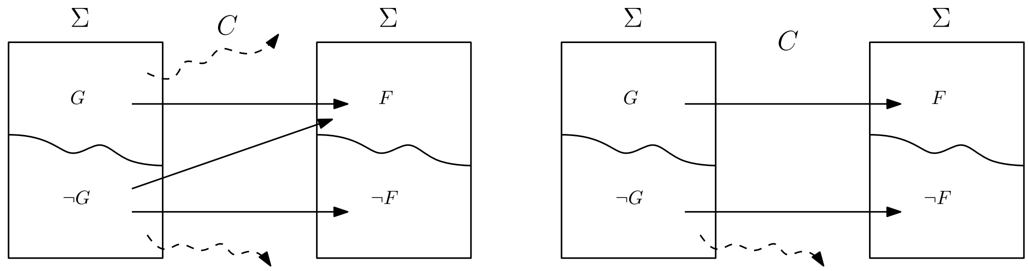
When skip is executed, the program state does not change and the consecutive part is executed. When diverge is executed, the execution never stops and the program can not reach a final state.

In our representation of GCL, non-determinism is explicitly constructed via the infix operator $\Box$, whereas in its original definition, non-determinism occurs when the guards within the if and while commands are not mutually exclusive [4]. 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)

We define the weakest precondition transformer inductively over the program structure in lambda-calculus style[4] as in Table 2.4:

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

Table 2.4: The Weakest Precondition Transformer for Deterministic Programs [8]

$F[x/e]$ 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:

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

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

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

The definition for the while-loop [8] is trickier, but we can verify its correctness by recalling Dijkstra's original definition in the following section.

2.4.2 *Defining Loops*

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

Let

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

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 IF discharges 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:

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

The definition in Table 2.4, however, uses the least fixed point of the characteristic function that is not obvious. We understand the use of fixed point in two ways. First, a precondition G being a fixed point of the characteristic function $G = \Phi(G) = (\neg\varphi \wedge F) \vee (\varphi \wedge wp.C'.G)$ means that under control of G, termination is possible (left side of the disjunction) and repeated execution of C' is possible(right side of the disjunction), since G is invariant before and after the execution of C'. Second, if we were to believe that the semantics of WHILE should be equivalent to the semantics of $\mathit{if}(\varphi)\{C;\mathit{WHILE}\}\ \mathit{else}\ \{\mathit{skip}\}$ [5], we can derive the need for fixed point:

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

The question then arises: can we define wp with any fixed point? The answer is no and we show it by verifying that the definition in Table 2.4 coincides with

5 The program in the else-branch is skip instead $\mathit{diverge}$, 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.

Dijkstra's definition at the beginning of this chapter.[6] Thanks to domain theory, we have a heuristic to calculate the least fixed point of Φ.

Theorem 2.1 [8] $\mathrm{lfp}\ \Phi = \sup_{n \in \mathbb{N}} \Phi^n(\mathit{false})$

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

Lemma 2.2 $\forall k \geqslant 0 : H_k(F) = \Phi^{k+1}(\mathit{false})$

Proof. Proof by induction.

BASE CASE:

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

Line (***) is supported by the Law of Excluded Miracle [2, 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:

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

$\square$

Thus, by identifying the least fixed point, we find a k that satisfies Equation 2.1. 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. Essentially, they express the same predicate, i.e. the "weakest" precondition for while-loops which is unique. Consequently, it means that we can not use other fixed points to define $wp.\mathrm{WHILE}$, which are weaker than the least fixed point. For the same reason, we will see that greatest fixed point is necessary to define the weakest liberal precondition.

6 In fact, Dijkstra and Scholten [4] 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.

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

Now we bring the non-deterministic choice back into the picture and add its wp to Table 2.5. Here we assume a setting with angelic non-determinism, where we assume that whenever non-determinism occurs, it will be resolved in our favor. This results in the weakest precondition for our non-deterministic choice being a disjunction of the wp for its subprograms. We are hopeful that a precondition satisfying the wp of one of the subprograms can also lead to termination in our desired postcondition. This is a design choice that is different from Dijkstra's [3], where the wp for non-deterministic choice is a conjunction, hinting at a demonic setting. Both choices are justifiable, we choose to follow Zhang and Kaminski's work, favoring the resulting Galois connection between the weakest (liberal) precondition transformers and the strongest (liberal) postcondition transformers [14].

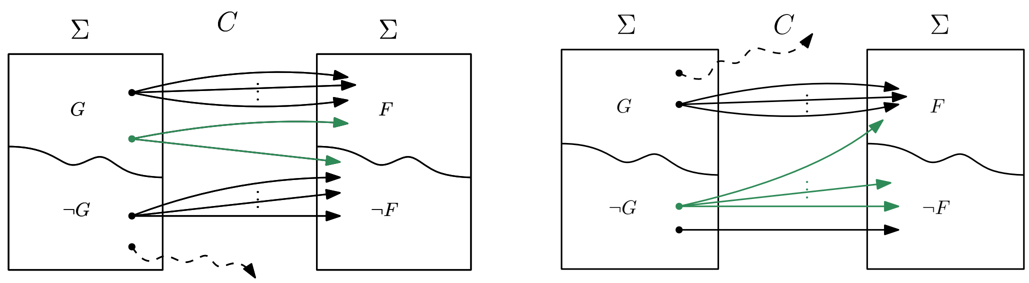
C	**wp.C.F**	**wlp.C.F**
skip	F	F
diverge	false	true
$x := e$	$F[x/e]$	$F[x/e]$
$C_1; C_2$	$wp.C_1.(wp.C_2.F)$	$wp.C_1.(wp.C_2.F)$
$\{C_1\}\square\{C_2\}$	$wp.C_1.F \vee wp.C_2.F$	$wlp.C_1.F \wedge wlp.C_2.F$
if (φ) $\{C_1\}$ else $\{C_2\}$	$(\varphi \wedge wp.C_1.F) \vee (\neg\varphi \wedge wp.C_2.F)$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

Table 2.5: The Weakest (Liberal) Precondition Transformer for Non-deterministic Programs [8]

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).

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-



$G = wp.C.F$

(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)

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 [14].

We define the weakest liberal precondition transformer in Table 2.5. A graphical representation can be found on Figure 2.3.2.

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

Theorem 2.3 [8] $\text{gfp } \Phi = \sup_{n \in \mathbb{N}} \Phi^n(\text{true})$

2.6 STRONGEST POSTCONDITIONS

Following the style to define wp and wlp, Zhang and Kaminski [14] (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 is reachable from G. The definition of the predicate transformer sp is shown in Table 2.6.

C	**sp.C.G**
skip	G
diverge	false
$x := e$	$\exists a. x = e[x/a] \land G[x/a]$
$C_1; C_2$	$sp.C_2.(sp.C_1.G)$
$\{C_1\}\Box\{C_2\}$	$sp.C_1.G \lor sp.C_2.G$
if (φ) $\{C_1\}$ else $\{C_2\}$	$sp.C_1.(\varphi \land G) \lor sp.C_2.(\neg\varphi \land G)$
while (φ) $\{C'\}$	$\neg\varphi \land \text{lfp } X. G \lor sp.C.(\varphi \land X)$

Table 2.6: The Strongest Postcondition Transformer [14]

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

$$\frac{\sigma \xrightarrow{C_1} \mu,\ \mu \xrightarrow{C_2} \tau}{\sigma \xrightarrow{C_1;C_2} \tau}\,\mathtt{seq} \qquad \frac{\sigma \xrightarrow{C_i} \tau,\ i \in \{1,2\}}{\sigma \xrightarrow{C_1 \Box C_2} \tau}\,\mathtt{par}_i$$

$$\frac{\sigma \in \varphi,\ \sigma \xrightarrow{C_1} \tau}{\sigma \xrightarrow{\mathrm{IF}} \tau}\,\mathtt{if}_1 \qquad \frac{\sigma \notin \varphi,\ \sigma \xrightarrow{C_2} \tau}{\sigma \xrightarrow{\mathrm{IF}} \tau}\,\mathtt{if}_0$$

$$\frac{\sigma \notin \varphi}{\sigma \xrightarrow{\mathrm{WHILE}} \sigma}\,\mathtt{while}_0 \qquad \frac{\sigma \in \varphi,\ \sigma \xrightarrow{C} \mu,\ \mu \xrightarrow{\mathrm{WHILE}} \tau}{\sigma \xrightarrow{\mathrm{WHILE}} \sigma}\,\mathtt{while}_n$$

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

Table 2.7: Big Step Semantics

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

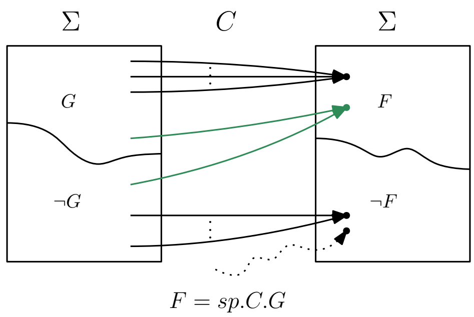


Figure 2.4: Strongest Postcondition (Angelic Non-determinism)

2.7 BIG STEP SEMANTICS

To express the meaning of programs, we choose big-step semantics to describe executions of a program. We take inspiration from Nipkow and Klein's book [11] and define our big-step semantics in Table 2.7.

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

We also use symbols $\sigma(x := v)$ to denote a state where the value of variable x is v, and all other variables have the same values as in σ. In our big-step semantics, non-termination of C is equivalent to the nonexistence of state τ such that $\sigma \xrightarrow{C} \tau$. With the definition of big-step semantics, we can precisely express the soundness of our predicate transformers in the following section.

2.8 SOUNDNESS

Theorem 2.4 *[Soundness of wp]* [14]

$$wp.C.F = \{\sigma \in \Sigma \mid \exists\tau \in \Sigma : \ \sigma \xrightarrow{C} \tau \ \wedge \ \tau \vDash F\}^{7}$$

Theorem 2.5 *[Soundness of wlp]* [4]

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

Theorem 2.6 *[Soundness of sp]* [13, 14]

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

2.9 PROPERTIES OF WP AND WLP

Theorem 2.7 *wp and wlp are each other's conjugate:*

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

Theorem 2.8

$$\forall C \in \mathcal{P} : \ wlp.C.true = true$$

8

7 $\exists\tau \in \Sigma : P$ is short for $\exists\tau.\tau \in \Sigma : P$

8 TODO: add references.

Part II

NECESSARY LIBERAL PRECONDITIONS

Some text about this part.

3 A PROOF SYSTEM

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

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

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

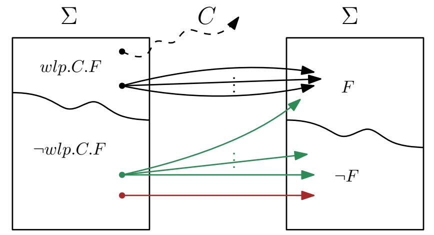
$$\begin{aligned} wlp.C.F \implies G &= \neg G \implies \neg wlp.C.F \\ &= \neg G \implies wp.C.\neg F && \text{Theorem 2.7} \end{aligned}$$

In the upcoming sections, we first discuss various forms that the necessary liberal precondition can take and try to identify a G that is most characteristic. We proceed then to propose a proof system stemming from the necessary liberal precondition and show its usefulness using an example.

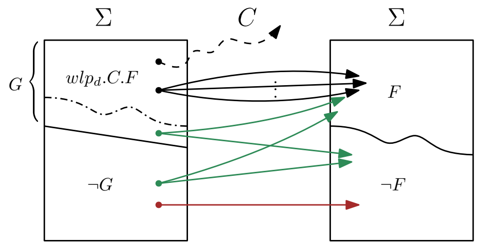
3.1 A PRECONDITION WEAKER THAN THE WEAKEST LIBERAL PRECONDITION

In Section 2.5 we defined the weakest liberal precondition and state that it characterizes all the preconditions under whose control the program either diverges or will terminate in a state satisfying F. We are certain to use "will" instead of "can", because we view the non-determinism as demonic, so the behavior of wlp can be depicted by Figure 3.1.1. We can categorize the executions of the program in four ways:

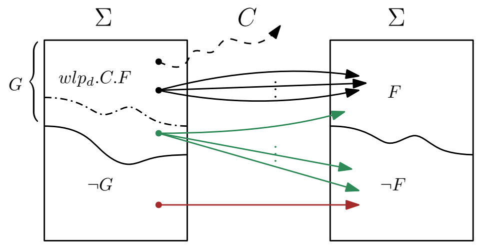
1. the dashed arrow means non-terminating executions;
2. the black arrows are executions starting from an initial state satisfying $wlp.C.F$ and only terminating in final states satisfying F;
3. the green arrows are the executions starting from an initial state satisfying $\neg wlp.C.F$ but can terminate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



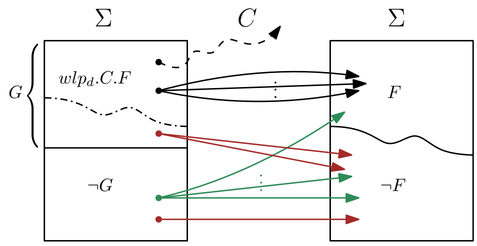
(1) Weakest liberal precondition (demonic non-determinism)



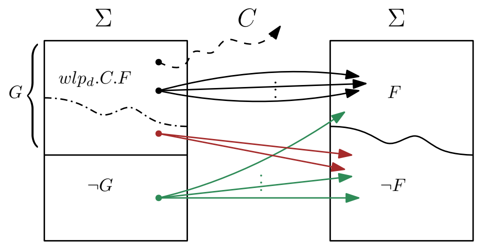
(2) Precondition G with $wlp.C.F \implies G$ and G contains some green arrows



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



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



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

Figure 3.1: Case Distinction of Preconditions Weaker Than wlp

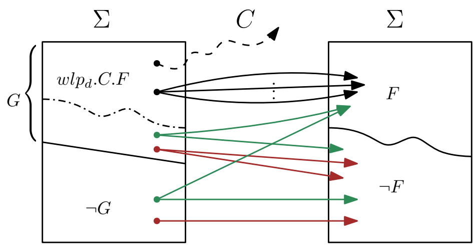
If we were to weaken the precondition, it can happen in various ways as shown in Figure 3.1.2-9. However, G spikes our interest when it takes the form as in Figure 3.1.3, because under its control, the program always can reach a final state satisfying F if it terminates, while with an initial state satisfying $\neg G$, the program is will terminate satisfying $\neg F$. This behavior is exactly the behavior of wlp, if we were to regard the non-deterministic choice as angelic, as hinted by the similarities between Figure 3.1.3 and Figure 2.3.1. We thus first investigate this special case, before proceeding with G in general.

3.1.1 *A Special Case*

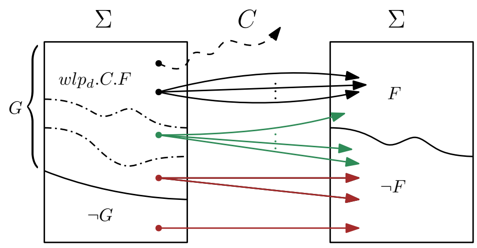
Dual to the semantics of wp and wlp as shown in Theorem 2.4 and Theorem 2.5, we can deduce the semantics of wlp with angelic non-determinism (denoted by wlp_a) recalling the representation for non-termination mentioned in Section 2.7:

Statement 3.9 *[Semantics of* wlp_a*]*

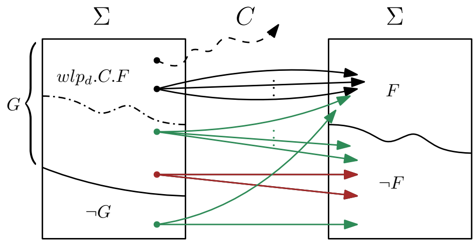
$$wlp_a.C.F = \{\sigma \in \Sigma \mid \neg(\exists \tau \in \Sigma : \sigma \xrightarrow{C} \tau) \;\vee\; (\exists \tau \in \Sigma : \sigma \xrightarrow{C} \tau \wedge \tau \vDash F)\}$$



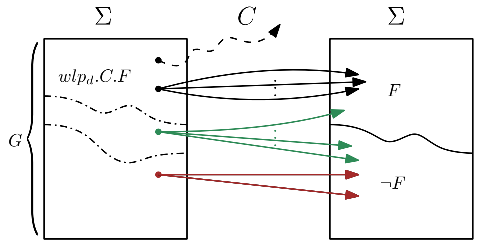
(6) Precondition G with $wlp.C.F \implies G$ and G contains some green arrows and some red arrows



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



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



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

Figure 3.1: Case Distinction of Preconditions Weaker Than wlp (Cont.)

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

Lemma 3.10 *[Angelic wlp implies G]*

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

The second prerequisite $sp.C.\neg G \implies \neg F$ states that from ¬G we are only allowed to reach ¬F, making sure that all green arrows as in Figure 3.1 are included in G.

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

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

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

$$sp.C.\neg G \implies \neg F \Leftrightarrow \tau \in sp.C.\neg G \implies \tau \in \neg F$$

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

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

$$\begin{aligned}
wlp_a.C.F \implies G &\Leftrightarrow \sigma \in wlp_a.C.F \implies \sigma \in G \\
&\Leftrightarrow \neg(\exists \tau \in \Sigma : \sigma \xrightarrow{C} \tau) \ \vee\ (\exists \tau \in \Sigma : \sigma \xrightarrow{C} \tau \wedge \tau \in F) \\
&\qquad \implies \sigma \in G \qquad |\ \text{Statement 3.9} \\
&\Leftrightarrow (\neg(\exists \tau \in \Sigma : \sigma \xrightarrow{C} \tau) \implies \sigma \in G) \qquad \text{(c)} \\
&\quad \wedge\ ((\exists \tau \in \Sigma : \sigma \xrightarrow{C} \tau \wedge \tau \in F) \implies \sigma \in G) \qquad \text{(d)}
\end{aligned}$$

We can prove Lemma 3.10 by proving that Line (a) implies Line (c) and that Line (b) implies Line (d). For any state $\sigma \in \Sigma$, we first prove (a) $\implies$ (c):

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

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

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

Then we conclude from Line (b) that

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

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

Lemma 3.11 *[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 we do not allow executions starting from G that only finish in $\neg F$, making sure that G does not include the red arrows as in Figure 3.1.

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

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

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

$$\begin{aligned}
& G \implies wlp_a.C.F \\
\Leftrightarrow\ & \sigma \in G \implies \sigma \in wlp_a.C.F \\
\Leftrightarrow\ & \sigma \in G \implies \neg(\exists \tau \in \Sigma : \sigma \xrightarrow{C} \tau) \ \vee \ (\exists \tau \in \Sigma : \sigma \xrightarrow{C} \tau \wedge \tau \in F) \\
& \hfill \mid \text{Statement 3.9}
\end{aligned}$$

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

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

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

Corollary 3.12 *[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 $G = wlp_a.C.F$

3.1.2 *The General Case*

While having restrictions on G yields interesting results, without the restrictions we can still find useful characteristics. As shown in Figure 3.1, G can contain all possible initial states, which can be the starting points of black, green, red, or dashed arrows, representing executions terminating in states satisfying F, F or $\neg F$, $\neg F$, or non-terminating, respectively. As a result, we can not make much statements without adding extra restrictions to G. However, we can see from Figure 3.1 that $\neg G$ does not contain any black or dashed arrows in all cases. In other words, if program C starts in any initial state satisfying $\neg G$, then either G is empty, or

- its executions terminate, and

- there exists an execution that ends up in a final state that satisfies $\neg F$.

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

```
1       ... // leave non-critical section
2       turn := B;
3       while (turn != A) do
4               turn := A □ turn := B
5               // modeling the behavior of other threads
6       critA := true;
7       ... // enter critical section
```

Listing 3.1: Thread A Hoping to Access Critical Section

The pseudocode is modified from Peterson's mutual exclusion algorithm [12], but we are now only concerned with one of possibly many threads. Our thread A is trying to enter some critical section but only have limited knowledge as to what other threads might also want to access the same critical section: A is only aware of thread B that also wants to enter critical section.

To have a fair system, thread A gives B the turn after leaving non-critical section as written in line 2 of Listing 3.1. Otherwise, thread A might never enter the while-loop and directly skip ahead to line 6 to enter the critical section, without giving other threads a chance. Line 4 models the behavior of other threads in the system: while A is waiting for its turn, thread B might also have just left the non-critical section and gave the turn to A; or some other thread than A or B might have given the turn to B. [1]

Mutual exclusion requires that no threads are simultaneously in the critical section. In this case, a state we definitely want to avoid is where the values of critA and critB are both true. In other words, the postcondition

$$F = \{\sigma \in \Sigma \mid \sigma.\mathrm{critB} = \mathrm{true}\}$$

after line 6 in Listing 3.1 is an undesirable postcondition.

Filling in this F after line 6 and calculating the weakest liberal precondition backwards according to Table 2.5, we arrive at the conditions as shown in Listing 3.2. For better readability, we shorten predicates like F to $\{\mathrm{critB}\}$.

```
        ... // leave non-critical section
        {critB}
        turn := B;
        {critB}
        while (turn != A) do
                turn := A □ turn := B
                // modeling the behavior of other threads
        {critB}
```

1 We will see with the calculation for wlp of this while-loop that it makes no difference whether we include more non-deterministic choices like $\mathrm{turn} := C \;\square\; \mathrm{turn} := D \;\square\; \ldots$.

```
9          critA := true;
10         {critB}
11         ... // enter critical section
```

Listing 3.2: Weakest Liberal Precondition w.r.t Postcondition $F = \{\sigma \in \Sigma \mid \sigma.\mathrm{critB} = \mathrm{true}\}$

The only non-obvious step in Listing 3.2 is from line 8 to line 4. Remember from Table 2.5 that the wlp for while-loops is defined with the greatest fixed point operator:

$$wlp.(\mathrm{while}\ (\varphi)\ \{C'\}).F = \mathrm{gfp}\ X.\ (\neg\varphi \wedge F) \vee (\varphi \wedge wlp.C'.X)$$

We simply follow the iteration to find the greatest fixed point of a function until we see a pattern, then prove by natural induction that the solution we found is indeed the greatest fixed point (see Theorem 2.3).

Lemma 3.13 $wlp.\mathrm{WHILE}.\{\mathrm{critB}\} = \{\mathrm{critB}\}$ *where* $\mathrm{WHILE} := \mathrm{while}\ (\varphi)\ \mathrm{do}\ C'$, $\varphi := \{\mathrm{turn!} = A\}$, *and* $C' := (\mathrm{turn} := A \,\square\, \mathrm{turn} := B)$.

Proof.

$$\begin{aligned}
\text{Let } \Phi(X) &:= \neg\varphi \wedge F \quad\vee\quad \varphi \wedge wlp.C'.X \\
&= \{\mathrm{turn} = A\} \wedge \{\mathrm{critB}\} \quad\vee\quad \{\mathrm{turn!} = A\} \wedge wlp.C'.X \\
\text{Then } \Phi(\mathrm{true}) &= \{\mathrm{turn} = A\} \wedge \{\mathrm{critB}\} \quad\vee\quad \{\mathrm{turn!} = A\} \wedge wlp.C'.\mathrm{true} \\
&\overset{\text{Theorem 2.8}}{=} \{\mathrm{turn} = A\} \wedge \{\mathrm{critB}\} \quad\vee\quad \{\mathrm{turn!} = A\} \wedge \mathrm{true} \\
&= \{\mathrm{turn!} = A\} \vee \{\mathrm{critB}\}
\end{aligned}$$

$$\begin{aligned}
\text{Also, } wlp.C'.\Phi(\mathrm{True}) &= wlp.(\mathrm{turn} := A \,\square\, \mathrm{turn} := B).\Phi(\mathrm{true}) \\
&\overset{\text{Table 2.5}}{=} wlp.(\mathrm{turn} := A).(\{\mathrm{turn!} = A\} \vee \{\mathrm{critB}\}) \\
&\qquad \wedge wlp.(\mathrm{turn} := B).(\{\mathrm{turn!} = A\} \vee \{\mathrm{critB}\}) \\
&\overset{\text{Table 2.5}}{=} (\{A! = A\} \vee \{\mathrm{critB}\}) \wedge \{B! = A\} \vee \{\mathrm{critB}\} \\
&= \{\mathrm{critB}\} \wedge \mathrm{true} \\
&= \{\mathrm{critB}\}
\end{aligned}$$

$$\begin{aligned}
\text{Hence, } \Phi^2(\mathrm{true}) &= \{\mathrm{turn} := A\} \wedge \{\mathrm{critB}\} \\
&\qquad \vee \quad \{\mathrm{turn!} = A\} \wedge wlp.C'.\Phi(\mathrm{true}) \\
&= \{\mathrm{turn} := A\} \wedge \{\mathrm{critB}\} \quad\vee\quad \{\mathrm{turn!} = A\} \wedge \{\mathrm{critB}\} \\
&= \{\mathrm{critB}\} \wedge (\{\mathrm{turn} := A\} \vee \{\mathrm{turn!} = A\}) \\
&= \{\mathrm{critB}\} \\
\text{And } wlp.C'.\Phi^2(\mathrm{True}) &= wlp.(\mathrm{turn} := A \,\square\, \mathrm{turn} := B).\Phi^2(\mathrm{true}) \\
&\overset{\text{Table 2.5}}{=} wlp.(\mathrm{turn} := A).\{\mathrm{critB}\} \\
&\qquad \wedge wlp.(\mathrm{turn} := B).\{\mathrm{critB}\}
\end{aligned}$$

$$\begin{aligned} &\overset{\text{Table 2.5}}{=} \{\text{critB}\} \wedge \{\text{critB}\} \\ &= \{\text{critB}\} \\ &= \Phi(\text{true}) \end{aligned}$$

Consequently, $\Phi^3(\text{true}) = \Phi^2(\text{true}) = \{\text{critB}\}$

From the above results we can form the hypothesis

$$\forall i \in \mathbb{N} \wedge i \geqslant 2 : \Phi^i(\text{true}) = \{\text{critB}\}$$

and prove by natural induction:

1. Base case: $\Phi^2(\text{true}) = \{\text{critB}\}$. ✓
2. Step case: (IH stands for induction hypothesis $\Phi^i(\text{true}) = \{\text{critB}\}$)

$$\begin{aligned} \Phi^{i+1}(\text{true}) &= \{\text{turn} = A\} \wedge \{\text{critB}\} \;\vee\; \{\text{turn!} = A\} \wedge wlp.C'.\Phi^i(\text{true}) \\ &\overset{\text{IH}}{=} \{\text{turn} = A\} \wedge \{\text{critB}\} \;\vee\; \{\text{turn!} = A\} \wedge \{\text{critB}\} \\ &= \{\text{critB}\} \checkmark \end{aligned}$$

Combine this proven hypothesis with Theorem 2.3 and the definition of $wlp.\text{WHILE}.F$ in Table 2.5, we can conclude that

$$wlp.\text{WHILE}.\{\text{critB}\} = \text{gfp}\ \Phi = \sup_{n \in \mathbb{N}} \Phi^n(\text{true}) = \{\text{critB}\}$$

□

We have proven that $\{\text{critB}\}$ is the weakest liberal precondition of the program in Listing 3.1 w.r.t. postcondition $\{\text{critB}\}$, meaning that once $\{\text{critB}\}$ is satisfied as a precondition, the program is doomed to either end up in deadlock, or not even terminate, as illustrated in Figure 3.2. Hence, by asserting that $\{\text{critB}\}$ is not satisfied in the precondition, we can avoid that A and B are deadlocked.

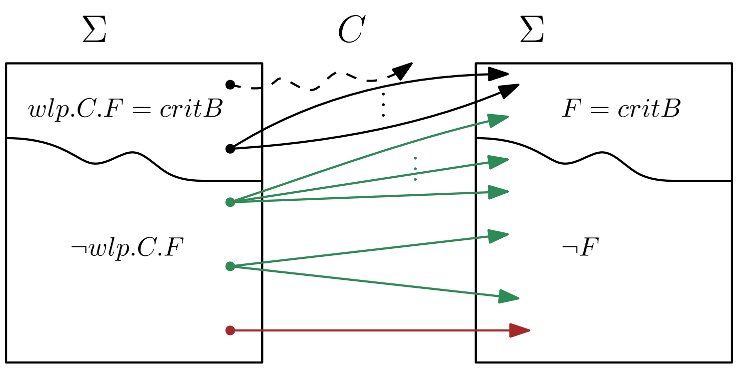


Figure 3.2: $wlp.C.F = \text{critB}$

However, B may not be the only process or thread that wishes to enter the same critical section as A. For example, a seasoned programmer with extra knowledge

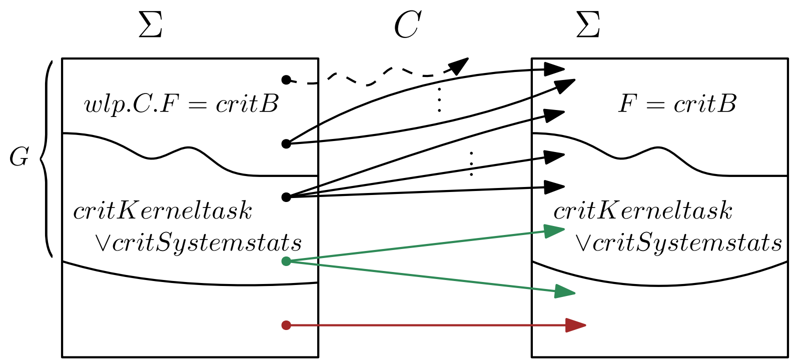


Figure 3.3: Relaxing the precondition with extra knowledge

may guess that processes like $\mathsf{kerneltask}$, $\mathsf{systemstats}$ are some of them, and relax the precondition that we should avoid to

$$wlp.C.F = \{critB\} \implies \{critB \vee critKerneltask \vee critSystemstats\} = G$$

as drawn in Figure 3.3. This demonstrates the use of the triple $wlp.C.F \implies G$: with insufficient knowledge we have over the system, 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 CONCLUSIONS

4.1 CONCLUSIONS

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

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

However, without extra constraints, G can be a precondition where all executions are possible. The only certainty is that $\neg G\ \{C\}\ \neg F$ is a valid Hoare Triple. Regardless, G still finds its usefulness while trying to identify preconditions that lead to erroneous final states, but without sufficient knowledge of all the undesired final states. 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.

4.2 FUTURE WORK

This thesis is only concerned with binary predicates, i.e. a predicate that evaluates to either true or false. Albeit classic, it might be more interesting to examine the above results in a quantitative setting, where predicates evaluate to more than true or false. In a quantitative setting, the notion of angelic or demonic non-determinism might be too absolute. 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? [1]

1 TODO: How about parallel composition?

Part III

APPENDIX

A

GRAPHICAL ILLUSTRATION OF PREDICATE TRANSFORMERS

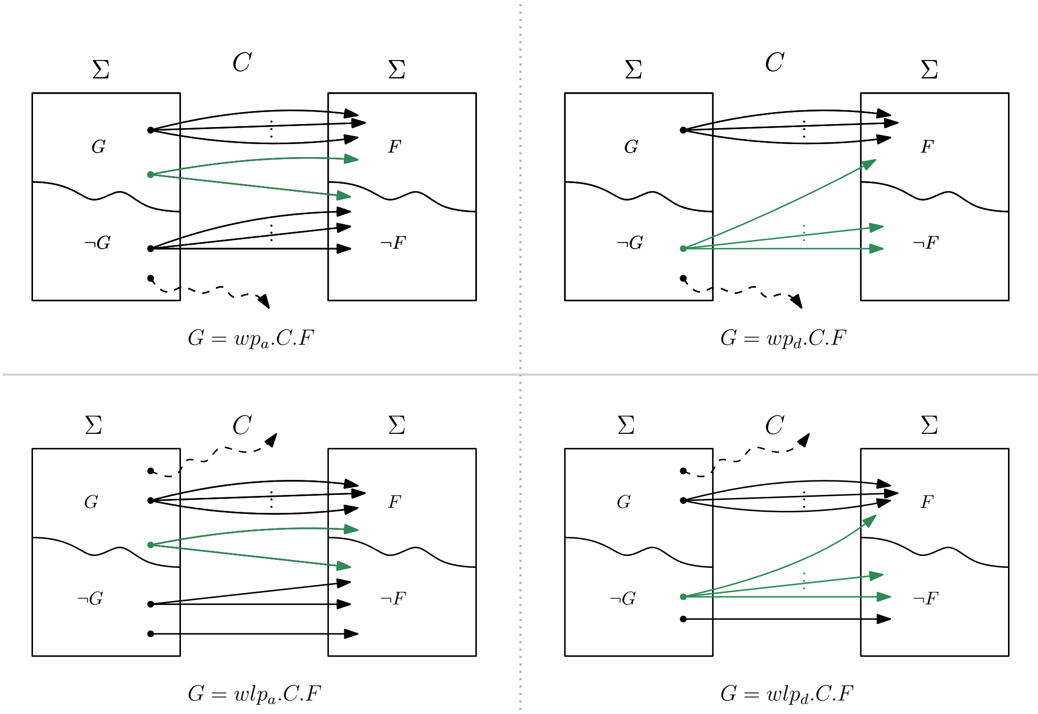


Figure A.1: Angelic and Demonic Nondeterminism

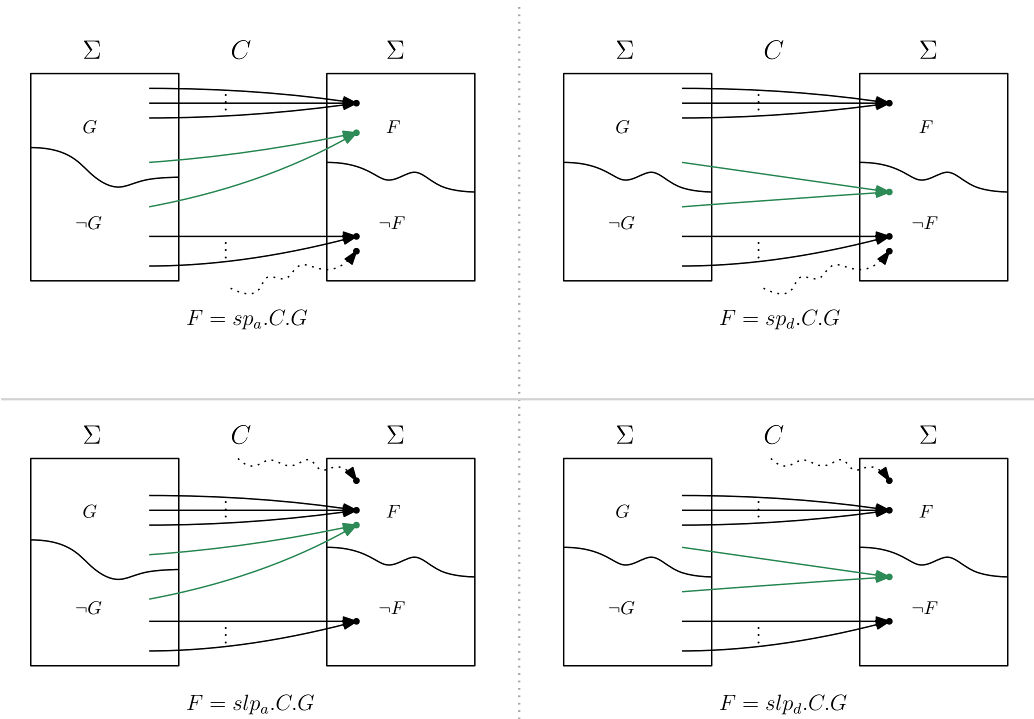


Figure A.2: sp and slp

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] Edsger W Dijkstra, Edsger Wybe Dijkstra, Edsger Wybe Dijkstra, and Edsger Wybe Dijkstra. *A discipline of programming*. Vol. 613924118. prentice-hall Englewood Cliffs, 1976.

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

[4] Edsger W. Dijkstra and Carel S. Scholten. "On substitution and replacement." In: *Predicate Calculus and Program Semantics*. New York, NY: Springer New York, 1990. ISBN: 978-1-4612-3228-5. URL: https://doi.org/10.1007/978-1-4612-3228-5_2.

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

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

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

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

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

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

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

[12] Gary L. Peterson. "Myths about the mutual exclusion problem." In: *Information Processing Letters* 12 (1981), pp. 115–116.

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

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