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

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

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 [2] (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 [2] to present programs in Chapter 2. We also explain their connections and differences.

Then we proceed to Chapter 3 **[fill in content of this chapter]**
[1]

1 TODO: Decide on all the colors in the end.

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
$s : \mathcal{X} \to \mathcal{V}$	program states
Σ	set of program states
$\mathcal{C}$	set of programs
$\mathcal{P}$	set of predicates
$F : \Sigma \to \{\mathtt{true}, \mathtt{false}\}$ $F := \{\sigma \in \Sigma \mid F(\sigma)\}$ (*)	predicates the set described by a predicate
$F(\sigma)$ (**) $F(\sigma) = \mathtt{true}$ (**) $\sigma \models F$ $\sigma \in F$	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
$\sigma \xrightarrow{c} \perp$	from initial state σ, an execution of program c diverges
$\exists x.P : F$ $\forall x.P : F$	there exists/for all x such that P is true: F is true

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 $F(\sigma) = \mathtt{true}$ for convenience's sake as shown in Lines (**). We also omit the use of equivalence symbol "$\equiv$" since we always use "$:=$" while defining objects, so we simply use the equation symbol "$=$" for equivalences instead. 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] [2]

[3]

$\{F[x/e]\}$ 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

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.

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

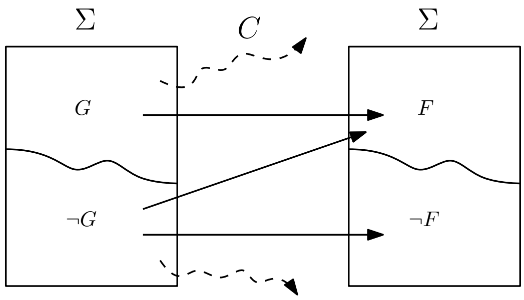
2 Non-determinism was not considered in the original paper, so we treat the programs here as deterministic. With deterministic programs, one initial state corresponds to one final state, and in case of non-termination we assign a final state $\perp$.

3 TODO: Think about whether to add liberally deterministic (Hesselink 1992, Programs, Recursion and Unbounded Choice).

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 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.



$G\{C\}F$

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 [2], among other things.

2.3 GUARDED COMMAND LANGUAGE

From now on we will use Dijkstra's (non-deterministic) guarded command language (GCL) [2] to represent programs and to include non-determinism (starting from Section 2.4.3). For better readability, we use an equivalent[4] form of GCL that is similar to modern pseudo-code as shown in Table 2.3.

4 Specifically, $\text{if } (\varphi) \ \{C_1\} \text{ else } \{C_2\}$ is equivalent to $\text{if } \varphi \rightarrow C_1 \ [] \ \neg\varphi \rightarrow C_2 \text{ fi}$ in Dijkstra's original style [2]; $\{C_1\}\square\{C_2\}$ is equivalent to $\text{if true} \rightarrow C_1 \ [] \ \text{true} \rightarrow C_2 \text{ fi}$.

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

Table 2.3: Guarded Command Language

The assignment, sequential composition, conditional choice, while-loop commands conform to their usual meaning. The non-deterministic choice $\{C_1\}\square\{C_2\}$ chooses from two programs randomly. 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 program goes to a state symbolizing non-termination, and the execution stops.

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 [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*

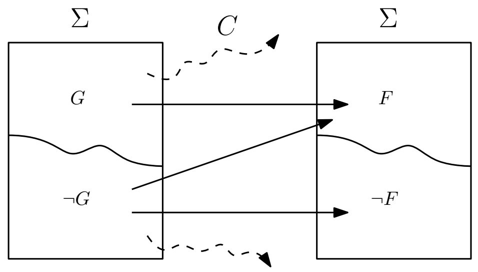
To better relate Hoare triples and Dijkstra's weakest precondition transformer, we first focus on deterministic programs. The goal is to find the necessary and sufficient precondition such that the program is guaranteed to terminate in a state that satisfies the postcondition. Figure 2.2 shows it graphically 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.

We define the weakest precondition transformer inductively over the program structure in lambda-calculus style[5] as in Table 2.4:

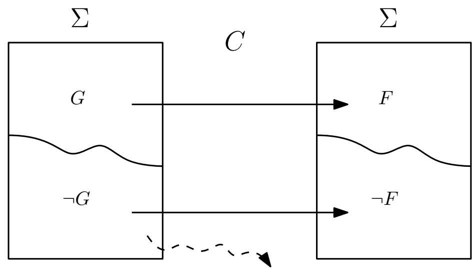
$F[x/e]$ is F where every occurrence of x is syntactically replaced by e.

$\mathrm{lfp}\ X.f$ is the least fixed point of function f with variable X.

5 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.



(1) Valid Hoare Triple



(2) Weakest precondition

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

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]

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 diverge and while deserve special attention. Since wp aims for total correctness, a program starting in an initial state 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 [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 [2], 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

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

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

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

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:

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

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 $\mathtt{if}(\varphi)\{C; WHILE\}else\{skip\}$, we can derive the need for fixed point:

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

The 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 Dijkstra's definition at the beginning of this chapter.[6] We borrow a theorem from domain theory that yields a computation for least fixed points, provided they exist:

Theorem 2.1 *hello test **[Insert theorem]***

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.

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.

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 [3, 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:

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

$\square$

Thus by identifying the least fixed point, we find a k that satisfies Equation 2.1. The advantage of using 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 $wp.WHILE$, which are weaker than the least fixed point. For the same reason, we will see that greatest fixed point is necessary to define weakest liberal precondition.

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

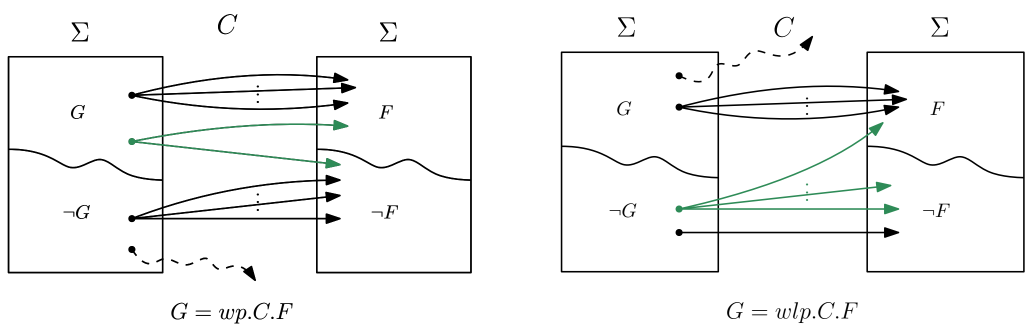
Now we bring the non-deterministic choice back into the picture and add its wp to Table 2.5. Here we assume a setting with angelic non-determinism, where we assume that whenever non-determinism occurs, it will be resolved in our favor. This results in the weakest precondition for our non-deterministic 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 [2], 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 [12].

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

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



(1) wp (angelic non-determinism) (2) wlp (demonic non-determinism)

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

2.5 WEAKEST LIBERAL PRECONDITIONS

While the wp-transformer excludes non-termination, the wlp-transformer takes a more liberal approach. The weakest precondition delivers a precondition so that the program terminates and a state satisfying the postcondition is reachable. The weakest liberal precondition, however, delivers a precondition so that the program either terminates satisfying the postcondition, or diverges. The postcondition in the wlp setting is guaranteed upon termination, because we regard the non-deterministic choice as demonic, again favoring to establish a Galois connection [12].

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.

2.6 STRONGEST POSTCONDITIONS

Following the style to define wp and wlp, Zhang and Kaminski [12] (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
$diverge$	$false$
$x := e$	$\exists a. x = e[x/a] \wedge G[x/a]$
$C_1; C_2$	$sp.C_2.(sp.C_1.G)$
$\{C_1\}\Box\{C_2\}$	$sp.C_1.G \vee sp.C_2.G$
$if\ (\varphi)\ \{C_1\}\ else\ \{C_2\}$	$sp.C_1.(\varphi \wedge G) \vee sp.C_2.(\neg\varphi \wedge G)$
$while\ (\varphi)\ \{C'\}$	$\neg\varphi \wedge lfp\ X.G \vee sp.C.(\varphi \wedge X)$

Table 2.6: The Strongest Postcondition Transformer [12]

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 $x := 1$.

2.7 SOUNDNESS

Theorem 2.3 *[Soundness of wp]* [12]

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

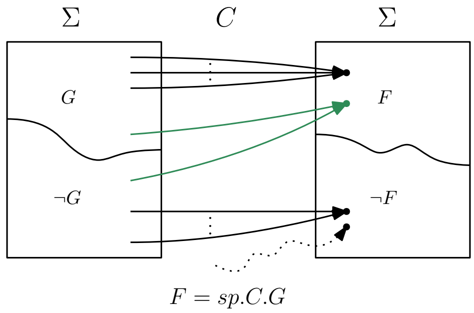


Figure 2.4: Strongest Postcondition (Angelic Non-determinism)

7

Theorem 2.4 *[Soundness of wlp]* [4]

$$wlp.C.F = \{\sigma \in \Sigma \mid \sigma \xrightarrow{C} \perp \;\vee\; \forall \tau \in \Sigma.\, \sigma \xrightarrow{C} \tau : \; \tau \models F\}$$

Theorem 2.5 *[Soundness of sp]* [11, 12]

$$sp.C.G = \{\tau \in \Sigma \mid \exists \sigma \in \Sigma.\, \sigma \xrightarrow{C} \tau : \; \sigma \models G\}$$

2.8 PROPERTIES OF WP AND WLP

wp and wlp are each other's conjugate:

$$wp.C.F = \neg wlp.C.\neg F$$

[TO BE CONTINUED]

7 TODO: make sure the theorem and its content is on the same page

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 $\neg F$:

$$\begin{aligned} wlp.C.F \implies G &= \neg G \implies \neg wlp.C.F \\ &= \neg G \implies wp.C.\neg F \end{aligned} \qquad |\ ^1$$

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. [2]

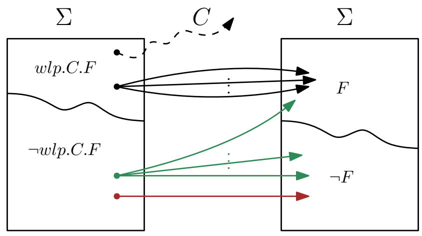
3.1 A PRECONDITION WEAKER THAN THE WEAKEST LIBERAL PRECONDITION

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:

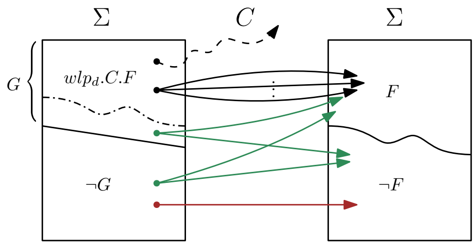
1. the dashed arrow means non-terminating executions;
2. the black arrows are executions starting from an initial state satisfying $wlp.C.F$ and only terminating in final states satisfying F;
3. the green arrows are the executions starting from an initial state satisfying $\neg wlp.C.F$ but can terminate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

2 TODO: replace with concrete example

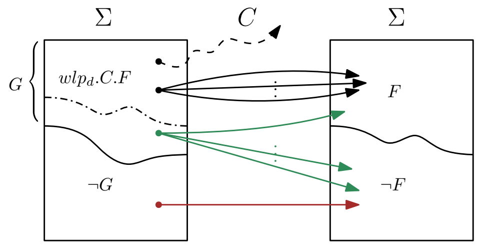
If we were to weaken the precondition, it can happen in various ways as shown in Figure 3.1.2-9. We argue that G is most characteristic, when it takes the form as in Figure 3.1.3, because under its control, the program always can reach a final state satisfying F if it terminates, while with an initial 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.



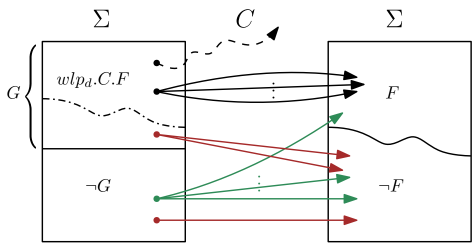
(1) Weakest liberal precondition (demonic nondeterminism)



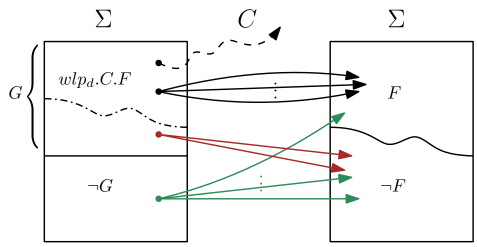
(2) Precondition G with $wlp.C.F \implies G$ and G contains some green arrows



(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

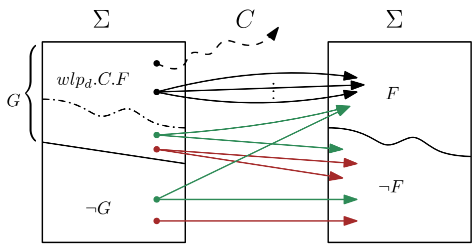
Figure 3.1: Case Distinction of Preconditions Weaker Than wlp (Part 1)

Dual to the semantics of wp and wlp as shown in Theorem 2.3 and Theorem 2.4, we can deduce the semantics of wlp with angelic non-determinism (denoted by wlp_a):

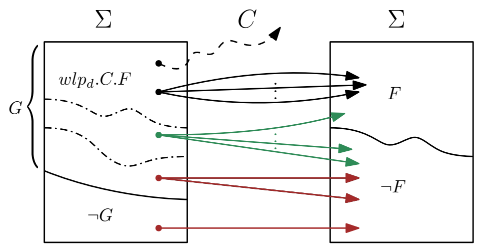
Statement 3.6 *[Semantics of wlp_a]*

$$wlp_a.C.F = \{\sigma \in \Sigma \mid \sigma \xrightarrow{C} \perp \ \vee\ \exists \tau.\sigma \xrightarrow{C} \tau : \ \tau \models F\}$$

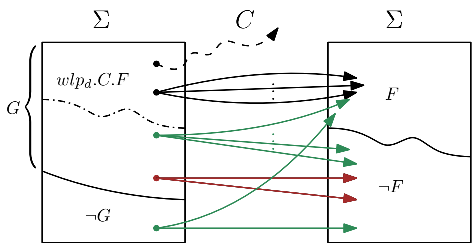
Luckily, we can find statements using wlp and sp that captures this specific G, hence giving us a way to express wlp_a without having to define it:



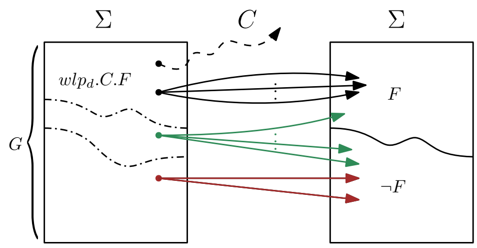
(6) Precondition G with wlp.C.F $\implies$ G and G contains some green arrows and some red arrows



(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 (Part 2)

Lemma 3.7 *[Angelic wlp implies G]*

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

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

Proof.

$$\begin{aligned}
wlp.C.F \implies G &= \sigma \in wlp.C.F \implies \sigma \in G \\
&= \sigma \xrightarrow{C} \bot \vee \forall \tau.\sigma \xrightarrow{C} \tau : \tau \in F \implies \sigma \in G \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid \text{Theorem 2.4} \\
&= (\sigma \xrightarrow{C} \bot \implies \sigma \in G) \qquad\qquad\quad \mid \text{(a)} \\
&\quad \vee (\forall \tau.\sigma \xrightarrow{C} \tau : \tau \in F \implies \sigma \in G) \quad \mid \text{(b)} \\
sp.C.\neg G \implies \neg F &= \tau \in sp.C.\neg G \implies \tau \in \neg F \\
&= \exists \sigma.\sigma \xrightarrow{C} \tau : \sigma \in \neg G \implies \tau \in \neg F \quad \mid \text{Theorem 2.5} \\
&= \neg(\tau \in \neg F) \implies \neg(\exists \sigma.\sigma \xrightarrow{C} \tau : \sigma \in \neg G) \\
&= \tau \in F \implies \forall \sigma.\sigma \xrightarrow{C} \tau : \neg(\sigma \in \neg G) \\
&= \tau \in F \implies \forall \sigma.\sigma \xrightarrow{C} \tau : \sigma \in G \qquad \mid \text{(c)} \\
wlp_a.C.F \implies G &= \sigma \in wlp_a.C.F \implies \sigma \in G
\end{aligned}$$

$$
\begin{aligned}
&= \sigma \xrightarrow{C} \bot \vee \exists \tau . \sigma \xrightarrow{C} \tau : \tau \in F \implies \sigma \in G \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid \text{Statement 3.6} \\
&= (\sigma \xrightarrow{C} \bot \implies \sigma \in G) \qquad\qquad \mid \text{(d)} \\
&\quad \vee (\exists \tau . \sigma \xrightarrow{C} \tau : \tau \in F \implies \sigma \in G) \quad \mid \text{(e)}
\end{aligned}
$$

Doing case distinction of the first assumption (Line (a) and Line (b)) gives us two options: if the Line (a) is true, then Line (d) is obviously also true; if merely Line (b) is true, then we need to use Line (c) as well. This requires case distinction whether F is empty.

If F is empty, then the prerequisite of Line (e) is trivially false, hence Line (e) is true. If F is non-empty, discharging the prerequisite of Line (e), we can find a witness $t \in F$ such that $\sigma \xrightarrow{C} t$. Because of Line (c) we know that all initial states that lead to final state t via program C are in G, hence $\sigma \in G$ and we proved our goal. □

Lemma 3.8 *[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.

$$
\begin{aligned}
&(P \implies G) \implies \neg(sp.C.P \implies \neg F) \\
&\qquad = (\sigma \in P \implies \sigma \in G) \implies \neg(\forall \tau . \tau \in sp.C.P : \tau \in \neg F) \\
&\qquad = (\sigma \in P \implies \sigma \in G) \implies \exists \tau . \tau \in sp.C.P : \tau \in F \\
&\qquad = (\sigma \in P \implies \sigma \in G) \implies \exists \tau . (\exists \sigma . \sigma \xrightarrow{C} \tau : \ \sigma \in P) : \tau \in F \qquad \mid \text{(f)} \\
&G \implies wlp_a.C.F \\
&\qquad = \sigma \in G \implies \sigma \in wlp_a.C.F \\
&\qquad = \sigma \in G \implies (\sigma \xrightarrow{C} \bot \vee \exists \tau . \sigma \xrightarrow{C} \tau : \tau \in F) \qquad\qquad \mid \text{(g)}
\end{aligned}
$$

Assume $\sigma \in G$, then we can construct set $P = \{\sigma\}$ such that the prerequisits in Line (f) holds. Now we can find witnesses t and s such that $s \xrightarrow{C} t$ and $s \in P$, $t \in F$.

Since P is a singleton set, s can only be σ. Then we have found a witness t such that $\sigma \xrightarrow{C} t$ and $t \in F$, satisfying the postrequisite of Line (g). □

Corollary 3.9 *[G equivalent to angelic wlp]*

$$
\begin{aligned}
&\textit{if } wlp.C.F \implies G \textit{ and } sp.C.\neg G \implies \neg F \textit{ and } (P \implies G) \implies \neg(sp.C.P \implies \neg F) \\
&\textit{then } G = wlp_a.C.F
\end{aligned}
$$

3.2 A PROOF SYSTEM

4 CONCLUSIONS

4.1 CONCLUSIONS

4.2 FUTURE WORK

Part III

APPENDIX

A

SYMBOLS AND ACRONYMS

SYMBOL	MEANING
fastidii ea ius	germano
suscipit instructior	titulo

ACRONYM	MEANING
quaestio philosophia	facto

B

GRAPHICAL ILLUSTRATION OF PREDICATE TRANSFORMERS

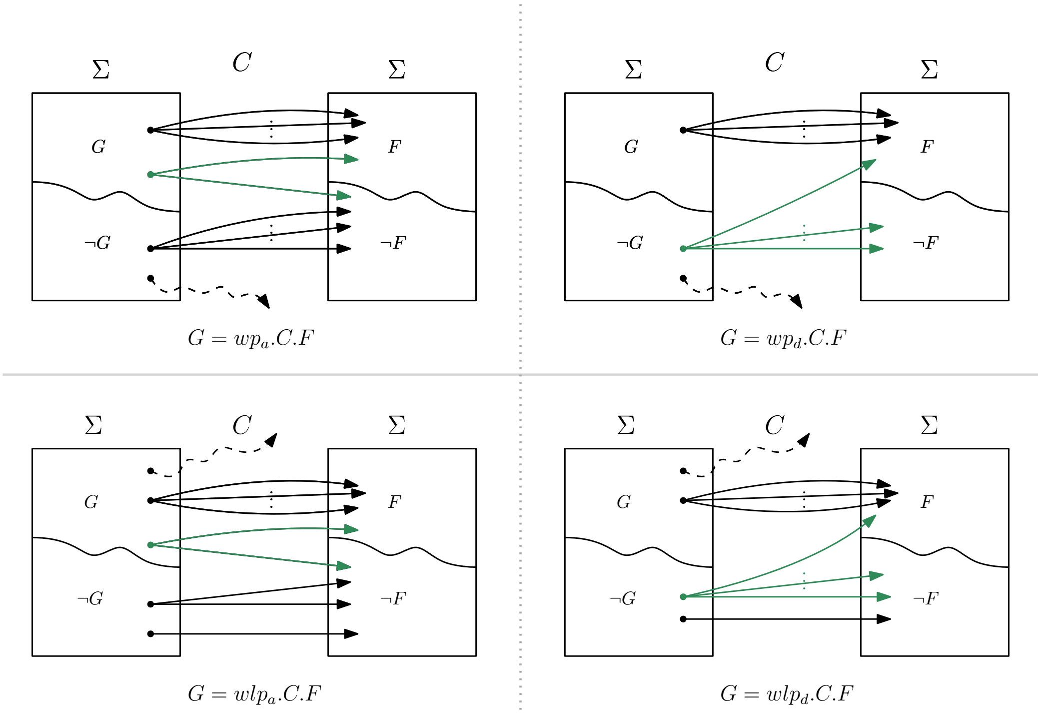


Figure B.1: Angelic and Demonic Nondeterminism

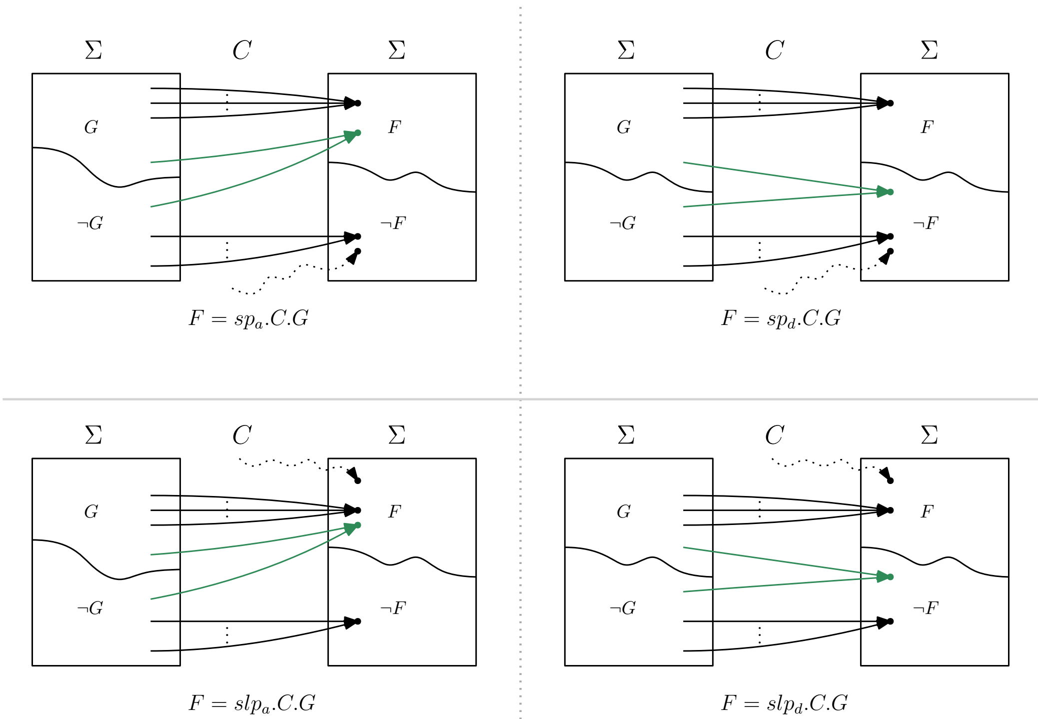


Figure B.2: sp and slp

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), 431–483. ISSN: 0164-0925. DOI: 10.1145/357146.357150. URL: https://doi.org/10.1145/357146.357150.

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

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

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

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