

# NECESSARY LIBERAL PRECONDITIONS: A PROOF SYSTEM

MASTER'S THESIS IN INFORMATICS

ANRAN WANG

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY - INFORMATICS  
TECHNICAL UNIVERSITY OF MUNICH





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

**MASTER'S THESIS IN INFORMATICS**

**ANRAN WANG, B.SC.**  
SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY - INFORMATICS  
TECHNICAL UNIVERSITY OF MUNICH

Examiner: Prof. Jan Křetínský  
Supervisors: Prof. Benjamin Lucien Kaminski  
Lena Verscht, M.Sc.  
Submission date: 15. September 2023





## 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, 15. September 2023*

---

Anran Wang

## ABSTRACT

---

This is where the abstract goes.

## ZUSAMMENFASSUNG

---

Kurze Zusammenfassung des Inhaltes in deutscher Sprache...

# CONTENTS

---

<b>I</b>	<b>HOARE TRIPLES, WEAKEST PRECONDITIONS, WEAKEST LIBERAL PRECONDITIONS</b>	<b>1</b>
1	BACKGROUND	2
2	PRELIMINARIES	3
2.1	Hoare Logic	3
2.2	Guarded Command Language	4
2.3	Weakest Precondition	5
2.3.1	The Deterministic Case	5
2.3.2	Defining Loops	6
2.3.3	The Non-deterministic Case	7
2.4	Weakest Liberal Precondition	7
<b>II</b>	<b>NECESSARY LIBERAL PRECONDITIONS</b>	<b>9</b>
3	A PROOF SYSTEM	10
3.1	A Proof System	10
4	CONCLUSIONS	11
4.1	Conclusions	11
4.2	Future Work	11
<b>III</b>	<b>APPENDIX</b>	<b>12</b>
	BIBLIOGRAPHY	13

## LIST OF FIGURES

---

Figure 1	Valid Hoare Triple (Deterministic)	4
Figure 2	Weakest Precondition (Deterministic)	5
Figure 3	Weakest Precondition (Non-deterministic)	7
Figure 4	Weakest Liberal Precondition (Deterministic)	8
Figure 5	Weakest Liberal Precondition (Non-deterministic)	8

## LIST OF TABLES

---

Table 1	Valid Hoare Triple	3
Table 2	Guarded Command Language	4
Table 3	The Weakest Precondition Transformer for Deterministic Programs [5]	6
Table 4	The Weakest Precondition Transformer for Non-deterministic Programs [5]	7
Table 5	The Weakest Liberal Precondition Transformer	8

## LISTINGS

---

## ACRONYMS

---



## Part I

### HOARE TRIPLES, WEAKEST PRECONDITIONS, WEAKEST LIBERAL PRECONDITIONS

Some text about this part.

## BACKGROUND

---

**TODO:** Make first letter big?

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

**TODO:** Rewrite; add chapter contents.

## PRELIMINARIES

### 2.1 HOARE LOGIC

Since the beginning of the 1960s, scholars have been researching the establishment of mathematics in computation [3, 7] 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* [4] 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 1](#).<sup>12</sup>  $\{F[x/e]\}$  is obtained by substituting occurrences of  $x$  by  $e$ .

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

Table 1: 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 1](#) 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  $x \leq 2 \{\text{while } x \leq 1 \text{ do } x := x * 2\} 1 < x \leq 2$ , it is provable in Hoare logic with the following proof tree. However, this while-loop will not terminate in case  $x \leq 0$  in the initial state.

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

<sup>2</sup> 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  $\perp$ .

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

$\frac{x \leq 1 \{x := x * 2\} x \leq 2}{x \leq 2 \{\text{while } x \leq 1 \text{ do } x := x * 2\} 1 < x \leq 2}$	
<b>Axiom of Assignment</b>	<b>Rule of Iteration</b>

Using style taken from Benjamin L. Kaminski's dissertation [5], Figure 1 illustrates a valid Hoare triple:  $\Sigma$  represents the set of all states, the section denoted with  $G$  includes the states that satisfy the predicate  $G$ . The arrow from left to right denotes the execution of the program  $C$ .

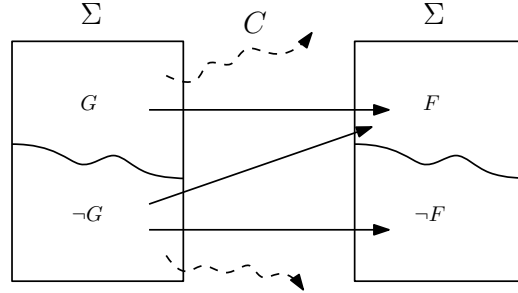


Figure 1: Valid Hoare Triple (Deterministic)

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 [6], 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 1, 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 1, which is what Edsger W. Dijkstra accomplished with his **weakest precondition** transformer in 1975 [2], among other things.

## 2.2 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 2.3.3). For better readability, we use an equivalent<sup>3</sup> form of GCL that is similar to modern pseudo-code as shown in Table 2.

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

Table 2: Guarded Command Language

<sup>3</sup> Specifically,  $\text{if } (\varphi) \{C_1\} \text{ else } \{C_2\}$  is equivalent to  $\text{if } \varphi \rightarrow C_1 \square \neg\varphi \rightarrow C_2 \text{ fi}$  in Dijkstra's original style [2];  $\{C_1\} \square \{C_2\}$  is equivalent to  $\text{if true} \rightarrow C_1 \square \text{true} \rightarrow C_2 \text{ fi}$ .

The **assignment**, **sequential composition**, **conditional choice**, **while-loop** commands conform to their usual meaning. The **non-deterministic choice**  $\{C_1\} \sqcap \{C_2\}$  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  $\perp$  symbolizing non-termination, and the execution stops.

## 2.3 WEAKEST PRECONDITION

### 2.3.1 The Deterministic Case

To better relate Hoare triples and Dijkstra's weakest precondition transformer, we first focus on deterministic programs. Again, 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 shows it graphically.

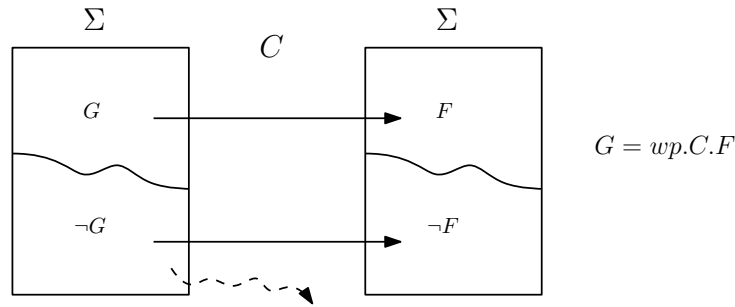


Figure 2: Weakest Precondition (Deterministic)

**TODO:** Discuss with Lena about this figure.

We define the **weakest precondition** transformer inductively over the program structure in lambda-calculus style<sup>4</sup> as in Table 3:

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

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

Let

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

be the characteristic function, then  $\text{wp}$  for while-loop can be defined as:

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

Most of the definitions in Table 3 are intuitive and correspond to their counterparts in Hoare logic, while those for **diverge** and **while** deserve special attention.

<sup>4</sup> For example,  $\text{wp}.C.F$  can be seen as  $\text{wp}(C, F)$  in “typical” style, where  $\text{wp}$  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  $\text{wp}.C.F.\sigma$  where  $\sigma$  means the initial state. Here  $\text{wp}$  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  $\text{wp}$ .

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

Table 3: The Weakest Precondition Transformer for Deterministic Programs [5]

Since wp aims for total correctness, a program starting in an initial state satisfying the precondition  $\text{wp}.\text{diverge}.F$  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 [5] is trickier, but we can verify its correctness by recalling Dijkstra's original definition in the following section.

**TODO:** Find out if there's earlier definition that used lfp.

### 2.3.2 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

$$\text{WHILE} = \text{while}(\varphi)\{C'\} \quad \text{and} \quad \text{IF} = \text{if } (\varphi)\{C'; \text{WHILE}\} \text{ else } \{\text{skip}\}.$$

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

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

Intuitively, when  $H_0(F)$  is satisfied before the execution of WHILE, the loop is exited with 0 iteration in a state that satisfies  $F \wedge \neg\varphi$  hence  $F$ . Then, we can understand  $H_k(F)$  as the weakest precondition such that the program terminates in a final state satisfying  $F$  after **at most**  $k$  iterations. **TODO:** Explain a bit more about how this relate to the

Then by definition:

$$\text{wp}.\text{WHILE}.F = (\exists k \geq 0 : H_k(F)) \tag{1}$$

We state that our definition in Table 3 coincides with this definition. Without going too deep into domain theory, we only use one of its theorem that yields a computation for least fixed points, when they exist.

**Theorem 1.** **TODO:** Insert theorem, then explain least point iteration from bottom.

Coincidentally,  $H_k(F)$  is the  $k$ -th iteration of the characteristic function  $\Phi$  from the bottom element, denoted by  $\Phi^k(\perp)$ . Thus by identifying the least fixed point, we've found a  $k$  that satisfies (1).

## 2.3.3 The Non-deterministic Case

Now we bring the non-deterministic choice back into the picture and add its wp to Table 4. To justify this definition for the non-deterministic choice, we must first clarify the intended semantics of the wp-transformer.

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

Table 4: The Weakest Precondition Transformer for Non-deterministic Programs [5]

Let  $\llbracket C \rrbracket$  denote the **execution** of program  $C$  and  $\llbracket C \rrbracket.\sigma$  denote the set of final states that **can** occur after the execution of  $C$ . A state is a function that maps a program variable to a value. The set of **states** is denoted by  $\Sigma = \{\sigma \mid \sigma : \text{Vars} \rightarrow \text{Vals}\}$ .

If  $C$  is deterministic, then  $\llbracket C \rrbracket.\sigma$  is a set of a single state, either a final state  $\sigma'$  or  $\perp$ , if the execution does not terminate. If  $C$  is non-deterministic,  $\llbracket C \rrbracket.\sigma$  can be a set with multiple elements, since multiple final states can be possible.

The weakest precondition  $\text{wp}.C.F$  is then

**Note for readers:** Up to here is readable.

Figure 3 shows wp with non-deterministic programs. Each arrow from left to right shows a **possible** execution of program  $C$ .

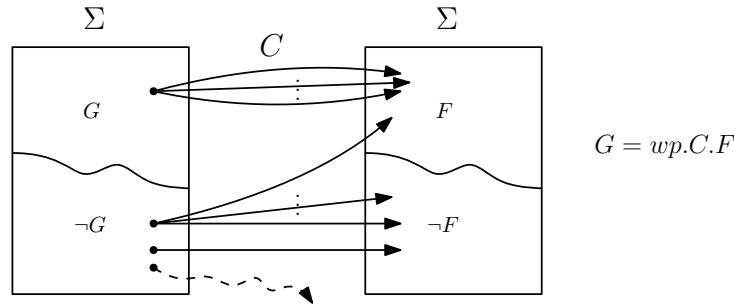


Figure 3: Weakest Precondition (Non-deterministic)

## 2.4 WEAKEST LIBERAL PRECONDITION

We define the weakest liberal precondition transformer in Table 5.

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

Table 5: The Weakest Liberal Precondition Transformer

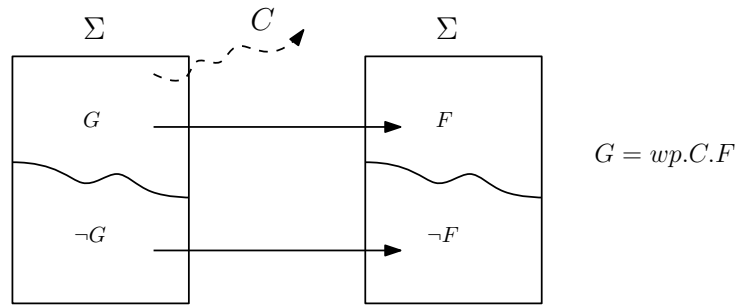


Figure 4: Weakest Liberal Precondition (Deterministic)

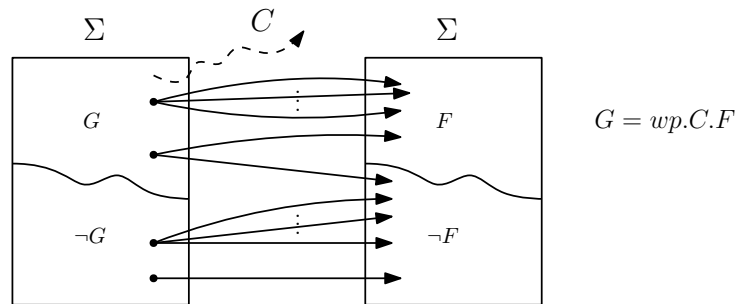


Figure 5: Weakest Liberal Precondition (Non-deterministic)



## Part II

### NECESSARY LIBERAL PRECONDITIONS

Some text about this part.

## A PROOF SYSTEM

---

### 3.1 A PROOF SYSTEM

In this section we study the necessary liberal precondition:

$$\text{wlp.C.F} \implies G$$

## CONCLUSIONS

---

### 4.1 CONCLUSIONS

### 4.2 FUTURE WORK

Part III

APPENDIX

## BIBLIOGRAPHY

---

- [1] Krzysztof R. Apt. “Ten Years of Hoare’s Logic: A Survey—Part I.” In: *ACM Trans. Program. Lang. Syst.* 3.4 (1981), 431–483. ISSN: 0164-0925. DOI: [10.1145/357146.357150](https://doi.org/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] Robert W Floyd. “Assigning meanings to programs.” In: *Program Verification: Fundamental Issues in Computer Science* (1993), pp. 65–81.
- [4] Charles Antony Richard Hoare. “An axiomatic basis for computer programming.” In: *Communications of the ACM* 12.10 (1969), pp. 576–580.
- [5] Benjamin Lucien Kaminski. “Advanced weakest precondition calculi for probabilistic programs.” PhD thesis. RWTH Aachen University, 2019.
- [6] Zohar Manna and Amir Pnueli. “Axiomatic approach to total correctness of programs.” In: *Acta Informatica* 3 (1974), pp. 243–263.
- [7] John McCarthy. “Towards a mathematical science of computation.” In: *Program Verification: Fundamental Issues in Computer Science* (1993), pp. 35–56.

## COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". `classicthesis` is available for both  $\text{\LaTeX}$  and  $\text{\LyX}$ :

<https://bitbucket.org/amiede/classicthesis/>

Happy users of `classicthesis` usually send a real postcard to the author, a collection of postcards received so far is featured here:

<http://postcards.miede.de/>

*Final Version* as of May 30, 2023 (`classicthesis` version 0.1).