WEAKEST PRECONDITION REASONING 2

Weakest precondition reasoning [Dij75; Dij76] is a technique developed by Edsger Wybe Dijkstra for formal reasoning about the correctness of nonprobabilistic programs. In this chapter, we will introduce this technique and show how it can be generalized to allow for quantitative reasoning. Later, in Chapter 4, we will see how those quantitative techniques can be extended to reasoning about probabilistic programs.

This chapter is organized as follows: We first present the programming language GCL. This language (as well as probabilistic and recursive extensions of it) will be used throughout this entire thesis. We then recap Dijkstra's original weakest precondition calculus which allows for reasoning at the level of predicates. Finally, we show how to extend this style of reasoning beyond predicates to quantities, e.g. values of program variables.

2.1 THE GUARDED COMMAND LANGUAGE (GCL)

Dijkstra's Guarded Command Language (GCL) [Dij76] is a very simple, yet Turing–complete [Wikm], imperative model programming language that still provides enough structure to produce readable programs. In that sense the GCL formalism lies between elementary models of computation like e.g. Turing–machines or the Lambda calculus (very simple formalism, less readable programs) and higher programming languages like e.g. C++ or Java (more complex formalism, much more readable programs).

We use GCL to describe ordinary, i.e. nonprobabilistic, programs. There is thus no access to any source of randomness within GCL. On the other hand, we will later see that GCL does include features for modeling uncertainty, namely *nondeterministic choices*. Later in Chapter 3, we will see how this language can be extended further to express probabilistic computations as well. For now, though, let us consider only ordinary, deterministic programs:

Definition 2.1 (The Guarded Command Language [Dij76]):

a. *Let* Vars *be a countable set of program variables and let* Vals *be a countable set of values. If not explicitly stated otherwise, we let* $\mathsf{Vals} = \mathbb{Q}$*, where* $\mathbb{Q}$ *is the set of rational numbers. For later use, let* $\aleph : \mathbb{N} \to \mathsf{Vals}$*. be a* bijective *canonical enumeration of* Vals*.*

b. *The set of program states is given by*

$$\Sigma \;=\; \{\sigma \mid \sigma : \mathsf{Vars} \to \mathsf{Vals}\}\ .$$

c. *The set of programs in guarded command language, denoted GCL, is given by the grammar*

$$\begin{aligned} C \;\longrightarrow\; & \texttt{skip} && \text{(effectless program)}\\ \mid\; & \texttt{diverge} && \text{(freeze)}\\ \mid\; & x := E && \text{(assignment)}\\ \mid\; & C \,;\, C && \text{(sequential composition)}\\ \mid\; & \texttt{if}\,(\varphi)\,\{C\}\,\texttt{else}\,\{C\} && \text{(conditional choice)}\\ \mid\; & \texttt{while}(\varphi)\{C\}\,, && \text{(while loop)} \end{aligned}$$

where $x \in \mathsf{Vars}$ is a program variable, E is an arithmetic expression over program variables, and φ is a boolean expression over program variables guarding a choice or a loop.

d. *A program containing no* `diverge` *or while loops is loop–free.*

e. *Given a program state σ, we denote by $\sigma(E)$ the evaluation of expression E in σ, i.e. the value obtained by evaluating E after replacing any occurrence of any program variable x in E by the value $\sigma(x)$. Analogously, we denote by $\sigma(\varphi)$ the evaluation of φ in σ to either true or false. Furthermore, for a value $v \in \mathsf{Vals}$ we write $\sigma\,[x \mapsto v]$ to indicate that in σ we set x to v, i.e.*[1]

$$\sigma\,[x \mapsto v] \;=\; \lambda y.\, \begin{cases} v, & \text{if } y = x \\ \sigma(y), & \text{if } y \neq x\,. \end{cases}$$

f. *We use the Iverson bracket notation [Wikg] to associate with each guard its according indicator function. The Iverson bracket $[\varphi]$ of guard φ is thus defined as the function*

$$[\varphi]\colon \quad \Sigma \to \{0,1\}, \quad [\varphi](\sigma) \;=\; \begin{cases} 1, & \text{if } \sigma(\varphi) = \text{true} \\ 0, & \text{if } \sigma(\varphi) = \text{false}. \end{cases}$$

Let us examine the computational effects of all GCL constructs. We start with the atomic programs: The effectless program `skip` does nothing, meaning that it terminates immediately in an unaltered program state. Starting in an initial state σ, the program `skip` will terminate in the same final state σ.

The freezing program `diverge` is a program that immediately enters an endless busy loop and therefore it does not terminate (diverges) regardless of the initial state it is started in. It can be thought of as a shorthand notation for the endless loop `while(true){skip}` (for the effects of loops, see below).

1 We use λ–expressions to construct functions: $\lambda\xi.\,\epsilon$ stands for the function that, when applied to an argument α, evaluates to ϵ in which every occurrence of ξ is replaced by α.

The assignment $x := E$ is the (only) statement that directly alters the program state. It evaluates E in the current program state and sets program variable x to the thusly obtained value. When executed on an initial state σ, the assignment $x := E$ thus terminates in final state $\sigma[x \mapsto \sigma(E)]$.

We proceed with the composed statements: The sequentially composed program $C_1 \fatsemi C_2$ has exactly the effect one would expect: First C_1 is executed and after its termination C_2 is executed. So if C_1 transforms state σ into σ' and C_2 transforms σ' into σ'', then $C_1 \fatsemi C_2$ transforms σ into σ''. Notice that if C_1 diverges on σ (e.g. if $C_1 = \texttt{diverge}$), then so does $C_1 \fatsemi C_2$.

The construct $\texttt{if}\,(\varphi)\,\{C_1\}\,\texttt{else}\,\{C_2\}$ is a conditional choice. Guard φ is a boolean expression over program variables, thus e.g. of the form $x > 0$ or $(x + y \leq z + 17) \wedge (z \neq 0)$. If φ evaluates to true in the current program state, then C_1 is executed and if φ evaluates to false, then C_2 is executed.

EXAMPLE 2.2 (Deterministic GCL Programs):
Consider the following loop–free GCL program:

$$
\begin{aligned}
C_{2.2} \rhd \quad & \texttt{if}\,(y > 0)\,\{x := 5\}\,\texttt{else}\,\{x := 2\} \fatsemi \\
& y := x - 3 \fatsemi \\
& \texttt{skip}
\end{aligned}
$$

First, this program checks whether y is strictly larger than 0. If this is the case, it sets x to 5. Otherwise (i.e. if $y \leq 0$) it sets x to 2. After that, the program sets y to the value of x decreased by 3. Finally, the program does an effectless operation by performing a `skip` statement.

The last construct $\texttt{while}(\varphi)\{C\}$ is a guarded while loop, that is executed as follows: If in the current program state σ the guard φ evaluates to false, then the whole loop immediately terminates without any effect. If on the other hand φ evaluates to true, then the loop body C is executed. After C has terminated in some state σ' (if C in fact terminates), the whole loop construct is invoked all over again but now starting from initial state σ': If φ evaluates to false in state σ', the loop terminates, otherwise C is executed to obtain a next state σ'', and so on. In principle the loop $\texttt{while}(\varphi)\{C\}$ is thus equivalent to the *infinitely long* (thus not well–formed) program

$$\texttt{if}\,(\varphi)\,\{C \fatsemi \texttt{if}\,(\varphi)\,\{C \fatsemi \cdots\}\,\texttt{else}\,\{\texttt{skip}\}\}\,\texttt{else}\,\{\texttt{skip}\}$$

which is an infinite nesting of simple conditional choices.

Notice that loops need not terminate on all initial states: Consider

$$\texttt{while}(x \neq 0)\{x := x - 1\},$$

which is a program that terminates only from those initial states σ in which $\sigma(x)$ is positive and moreover an integer.

EXAMPLE 2.3 (Deterministic GCL Programs):
Consider the following GCL program:

$$
\begin{array}{ll}
C_{2.3} \rhd & z := y \fatsemi \\
& \texttt{while}(x > 0)\{ \\
& \quad z := z + 1 \fatsemi \\
& \quad x := x - 1 \\
& \}
\end{array}
$$

$C_{2.3}$ first sets z to y and then, as long as x is strictly larger than 0, it repeats the following two steps: It adds 1 to z and subtracts 1 from x. These two steps (i.e. the effect of the loop body) will be repeated in total $\lceil x_0 \rceil$ times, where x_0 is the initial value of x.

Effectively this program therefore adds to y the rounded up value that x initially had and stores that result in variable z. As a (possibly unwanted) side–effect, the program also „forgets" about the value of x by effectively setting it to a value between 0 and -1.

2.2 REASONING ABOUT PREDICATES

We now develop formal reasoning about correctness of GCL programs at the level of predicates. For us, a predicate represents simply an arbitrary subset of program states, i.e. we can think of a predicate F as

$$F \in \mathcal{P}(\Sigma) ,$$

where for a set S we denote by $\mathcal{P}(S)$ its powerset. The predicate false stands for the empty set $\emptyset$ and dually true stands for the entire set Σ. Furthermore, $\neg F$ stands for the set $\Sigma \setminus F$, and $F \wedge G$ and $F \vee G$ stand respectively for the intersection and the union of the sets represented by F and G. We write

$$\sigma \models F$$

and say „σ satisfies F" to indicate that state σ is in the set represented by predicate F and $\sigma \not\models F$ to indicate that σ is not in that set. We write

$$F_1 \implies F_2$$

and say „F_1 implies F_2" to indicate that the set represented by predicate F_1 is a subset of the set represented by predicate F_2.

Our goal will be to associate to each program C and each predicate F (interpreted as a set of *final* states) a predicate G (interpreted as a set of *initial* states) such that if and only if the program C is started in any state $\sigma \models G$, then C terminates in a state $\tau \models F$. In the following we will gradually develop a calculus for achieving this.

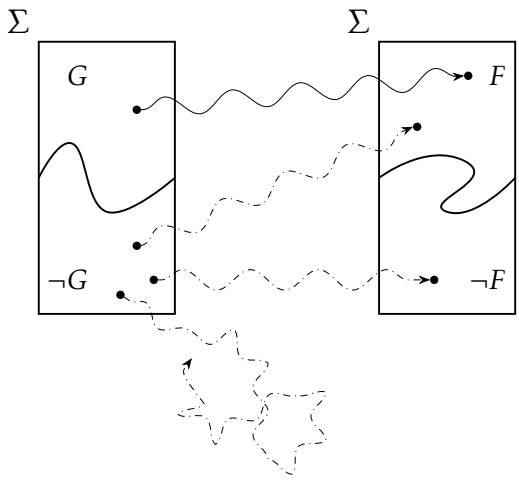

Figure 2.1: Hoare triple $\langle G \rangle\ C\ \langle F \rangle$ is valid: Starting in G, C will terminate in F. Starting in $\neg G$, we do not know whether C diverges or terminates in F or $\neg F$.

2.2.1 *Hoare Triples*

Hoare logic is a formal verification method seeded by the works of Robert (Bob) W Floyd[2] [Flo67a] and later developed further by Charles Antony (Tony) Richard Hoare [Hoa69]. It is therefore also called Floyd–Hoare logic.

The crucial concept of this technique are *Hoare triples*: Given two predicates F and G and a program C, a Hoare triple $\langle G \rangle\ C\ \langle F \rangle$ is said to be *valid* iff the following holds:

> If program C is started in some initial state $\sigma \models G$,
> then C *terminates* in a final state $\tau \models F$.

Notice our notion of validity means „valid for total correctness“[3] as presented by Manna & Pnueli [MP74]. In a more diagrammatic style, the situation is depicted in Figure 2.1. We call F a *postcondition* since we interpret it as a predicate over final states, i.e. F shall hold after (post) the execution of C. Dually, we call G a *precondition* since we interpret it as a predicate over initial states, i.e. G shall hold before (pre) the execution of C.

There are two things we would like to emphasize here: First, the validity of $\langle G \rangle\ C\ \langle F \rangle$ still does not tell us *anything* about what happens when C is

2 Floyd's middle name is in fact just W: „[Floyd] was indeed born with another middle name, but he had it legally changed to ‚W'—just as President Truman's middle name was simply ‚S'. Bob liked to point out that ‚W.' is a valid abbreviation for ‚W'.“ [Hai04]

3 As opposed to „valid for partial correctness“.

executed on some initial state $\sigma \not\models G$. In particular, it might be the case that C will terminate in a final state $\tau \models F$ nonetheless. This will be different for the notion of weakest preconditions.

Secondly, if $\langle G \rangle\ C\ \langle F \rangle$ is valid for some F, then it is *guaranteed that C terminates* from any state $\sigma \models G$. This means in particular that the validity of $\langle G \rangle\ C\ \langle \mathsf{true} \rangle$ simply states that C terminates from every state $\sigma \models G$, but it does not tell us anything about the final state τ, since $\forall \tau\colon\ \tau \models \mathsf{true}$.

2.2.2 *Weakest Preconditions*

To circumvent the dissatisfactory situation that validity of $\langle G \rangle\ C\ \langle F \rangle$ gives no information about the states satisfying $\neg G$ we now introduce the notion of weakest preconditions [Dij75; Dij76]. Imagine for that a Hoare triple

$$\langle \ldots\ldots \rangle\ C\ \langle F \rangle\ ,$$

where the precondition is left open just like in a cloze — so to speak: a Hoare triple with a *blank*. We would like to fill this blank in a very general way, namely with the *weakest possible predicate* G in the following sense: Any predicate G' for which $\langle G' \rangle\ C\ \langle F \rangle$ is valid should imply the more general (weaker) predicate G. Put more formally:

$$\forall G'\colon \qquad G' \implies G \quad \text{iff} \quad \langle G' \rangle\, C\, \langle F \rangle \text{ is valid} \tag{2.1}$$

Given program C and postcondition F, we call the (unique) predicate G that satisfies Condition (2.1) the

weakest precondition of C with respect to postcondition F,

denoted as $\mathsf{wp}\,[\![C]\!]\,(F)$. As a diagram, the situation is depicted in Figure 2.2: From any state $\sigma \models G$ the program C terminates in some state $\tau \models F$. Moreover, if C is started in a state $\sigma \not\models G$, then

- either C terminates in a state $\tau \not\models F$,
- or C does not terminate at all.

It is easy to see that the Hoare triple $\langle \mathsf{wp}\,[\![C]\!]\,(F) \rangle\ C\ \langle F \rangle$ is always valid and moreover that the following holds:

$$\langle G' \rangle\, C\, \langle F \rangle \text{ is valid} \qquad \text{iff} \qquad G' \implies \mathsf{wp}\,[\![C]\!]\,(F)$$

We can also see that the situation has now changed in comparison to Hoare triples in the sense that executing C on initial state $\sigma \not\models \mathsf{wp}\,[\![C]\!]\,(F)$ will *definitely not* terminate in a state $\tau \models F$.

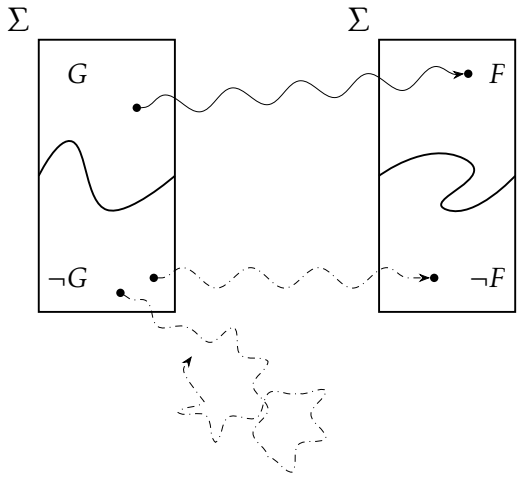

Figure 2.2: G is the weakest precondition of C with respect to postcondition F: Starting in G, C terminates in F. Starting in $\neg G$, C either diverges or terminates in $\neg F$.

2.2.3 *The Weakest Precondition Calculus*

Though we have defined what weakest preconditions are, given a program and a postcondition we yet have no *method* of finding out what the respective weakest precondition is. In the following we will therefore show how to obtain weakest preconditions in a systematic way, namely with the aid of a *backward moving continuation–passing style weakest precondition transformer*.

2.2.3.1 *Continuation–passing*

The principle of a continuation–passing style transformer is depicted in Figure 2.3: Assume we want to know the weakest precondition of the composed program $C_1 \fatsemi C_2$ with respect to postcondition F. Then we start from the end of $C_1 \fatsemi C_2$ with continuation F and move *backward* to the position between C_1 and C_2. While moving that position, we also transition from F to the weakest precondition of C_2 with respect to F, i.e. to $\mathrm{wp}\,[\![C_2]\!]\,(F)$.

Let us denote by G the intermediate predicate $\mathrm{wp}\,[\![C_2]\!]\,(F)$. Then G represents by definition exactly those states from which execution of C_2 will terminate in F. Therefore, we want precisely G to be *the* postcondition that the execution of C_1 should terminate in, so that the execution of the entire program $C_1 \fatsemi C_2$ terminates in F.

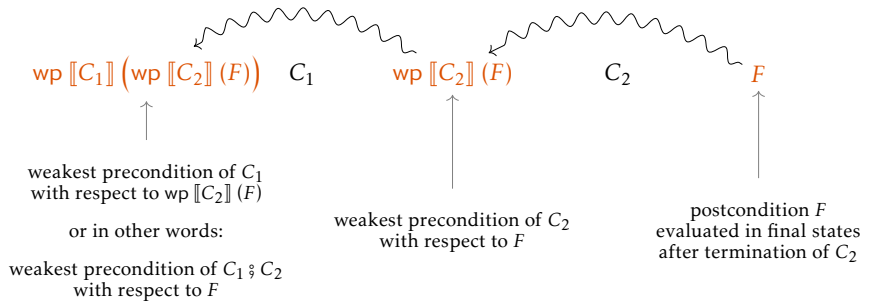

Figure 2.3: Continuation–passing style weakest precondition transformer.

To the program C_1 the predicate G is of course just an arbitrary continuation as any other. We can therefore determine the weakest precondition of C_1 with respect to postcondition G by moving to the beginning of program C_1 while transitioning from G to $\mathsf{wp}\,[\![C_1]\!]\,(G)$. But since $G = \mathsf{wp}\,[\![C_2]\!]\,(F)$, we have $\mathsf{wp}\,[\![C_1]\!]\,(G) = \mathsf{wp}\,[\![C_1]\!]\,(\mathsf{wp}\,[\![C_2]\!]\,(F))$, and therefore we have effectively moved from the end of program $C_1 \fatsemi C_2$ to its beginning while transitioning from F to the weakest precondition of $C_1 \fatsemi C_2$ with respect to F.

Now that we have some familiarity with the basic concept of continuation–passing, we will use it to give precise rules on how to obtain weakest preconditions for any program with respect to any postcondition. It turns out that this can be done in a very structured way, namely by induction on the structure of the programs: For every program C, we will show how to construct a continuation–passing style transformer $\mathsf{wp}\,[\![C]\!]$ of type

$$\mathsf{wp}\,[\![C]\!]\colon \quad \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)\ ,$$

that takes as input a postcondition F and returns the weakest precondition of C with respect to postcondition F. We first show how to construct these transformers for loop–free programs and then proceed with loops.

2.2.3.2 *Weakest Preconditions of Loop–free Programs*

The rules for constructing the wp transformer are given in Table 2.1. Let us ignore the definition for while loops for the time being and let us go over the other definitions one by one: Since the program `skip` has no effect, the postcondition F has to be transformed to the very same precondition F, i.e.

$$\mathsf{wp}\,[\![\texttt{skip}]\!]\,(F) \;=\; F\ .$$

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

Table 2.1: The weakest precondition transformer acting on predicates.

For the assignment $x := E$, we essentially *replace every occurrence of x in F by E*. However, since E does not actually *occur* in F, we define

$$\mathsf{wp}\,[\![x := E]\!]\,(F) \;=\; F\,[x/E]\,,$$

where $F\,[x/E]$ is a predicate whose indicator function is given by

$$\lambda\sigma.\;[F](\sigma\,[x \mapsto \sigma(E)])\,.$$

Notice that we have again used the Iverson bracket notation $[F]$ above.

Example 2.4 (Weakest Preconditions of Assignments):

A. $\mathsf{wp}\,[\![x := 5]\!]\,(x \leq 0) \;=\; (5 \leq 0) \;=\; \mathsf{false}$

B. $\mathsf{wp}\,[\![z := 18]\!]\,(x = 0) \;=\; (x = 0)$

C. $$\begin{aligned}\mathsf{wp}\,[\![x := 5]\!]\,((x > 2) \to (y = 7)) \;&=\; (5 > 2) \to (y = 7)\\ &=\; \mathsf{true} \to (y = 7)\\ &=\; (y = 7)\end{aligned}$$

In the predicates above, the symbol $\to$ (logical implication) is the usual abbreviation for $\neg A \vee B$. Notice that $\to$ is syntactic construct while $\Longrightarrow$ is a semantic one: $A \to B$ is *one predicate* while $A \implies B$ is a *statement concerning the two predicates A and B*.

Next, we turn to sequential composition: We have already seen the principle of continuation–passing and this very principle is implemented in the def-

inition of the transformer for sequential composition: Given the two transformers $\mathsf{wp}\,[\![C_1]\!]$ and $\mathsf{wp}\,[\![C_2]\!]$, we define

$$\mathsf{wp}\,[\![C_1 \,;\, C_2]\!]\,(F) \;=\; \mathsf{wp}\,[\![C_1]\!]\,(\mathsf{wp}\,[\![C_2]\!]\,(F))\;.$$

The intuition for the conditional choice $\texttt{if}\,(\varphi)\,\{C_1\}\,\texttt{else}\,\{C_2\}$ is the following: If in the initial state σ, the guard φ is satisfied, then C_1 will be executed. We thus need to associate with that case the weakest precondition of C_1 with respect to F. As a predicate, this reads as[4]

$$\varphi \to \mathsf{wp}\,[\![C_1]\!]\,(F)\;.$$

Dually, if we have $\sigma \not\models \varphi$, then C_2 is executed and we thus need to associate with that case the weakest precondition of C_2. As a predicate, this reads as

$$\neg\varphi \to \mathsf{wp}\,[\![C_2]\!]\,(F)\;.$$

We can now express that both cases (the case for $\sigma \models G$ and the case for $\sigma \not\models G$) must hold true in one predicate, namely by

$$(\varphi \to \mathsf{wp}\,[\![C_1]\!]\,(F)) \;\wedge\; (\neg\varphi \to \mathsf{wp}\,[\![C_2]\!]\,(F))\;,$$

which is logically equivalent to

$$(\varphi \wedge \mathsf{wp}\,[\![C_1]\!]\,(F)) \;\vee\; (\neg\varphi \wedge \mathsf{wp}\,[\![C_2]\!]\,(F))\;.$$

We choose the latter over the former representation in Table 2.1 because we will later associate $\wedge$ with $\cdot$ and $\vee$ with $+$, whereas an arithmetic representation of $\to$ is more cluttered and inconvenient.[5]

Before we turn our attention to the definition of wp for while loops, let us take a look at how we can formally reason about the program from Example 2.2 on page 25 by using the wp transformer:

EXAMPLE 2.5 (Weakest Preconditions of Loop–Free Programs):
We will reconsider the program $C_{2.2}$ from Example 2.2 and reason about the set of initial states from which the execution of $C_{2.2}$ terminates in a state satisfying postcondition $y^2 > 2$.

Throughout this thesis, we will use the notation

$$\begin{array}{l} /\!/\!/\ \boldsymbol{G'} \\ /\!/\!/\ \boldsymbol{G} \\ C \\ /\!/\!/\ \boldsymbol{F} \end{array}$$

4 As usually, $\neg$ binds stronger than $\to$.
5 Namely $\alpha \to \beta$ would need to be associated with $(1-\alpha)+\beta$.

to express the fact that $G = \mathsf{wp}\,[\![C]\!]\,(F)$ and moreover that G' is logically equivalent to G. It is thus more intuitive to read annotated programs from bottom to top, just like the wp transformer moves from the back to the front. Using this notation, we can annotate the program $C_{2.2}$ simply by applying the wp rules from Table 2.1 as shown in Figure 2.4.

By these annotations, we have established $\mathsf{wp}\,[\![C_{2.2}]\!]\,\big(y^2 > 0\big) = (y > 0)$. This tells us that from any initial state in which y is larger than 0 the execution of $C_{2.2}$ terminates in some final state τ in which y^2 is larger than 2.

Notice that $y > 0$ and $y^2 > 2$ are evaluated in different states, namely in initial and final states, respectively.

2.2.3.3 *Weakest Preconditions of Loops*

We now study weakest preconditions of loops. For the freezing program `diverge`, notice that for any postcondition F there is no initial state σ from which `diverge` terminates in some final state $\tau \models F$ (because `diverge` does not terminate at all). So whatever precondition we assign to `diverge` with respect to F, it may not be satisfiable by any σ. Therefore, the weakest precondition of `diverge` with respect to any postcondition F must be defined as

$$\mathsf{wp}\,[\![\texttt{diverge}]\!]\,(F) \;=\; \mathsf{false}\,,$$

since $\forall \sigma\colon \sigma \not\models \mathsf{false}$.

If we take a look at the definition of wp for while loops in Table 2.1, we see that it is defined using a *least fixed point operator* (lfp), namely as

$$\mathsf{lfp}\; X.\; \underbrace{(\neg\varphi \wedge F) \vee (\varphi \wedge \mathsf{wp}\,[\![C']\!]\,(X))}_{=:\,\Phi(X)}\,,$$

by which we mean the least fixed point of the characteristic function $\Phi(X)$. This function is of type $\mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$, thus mapping predicates to predicates. A fixed point of Φ is a predicate G such that $\Phi(G) = G$. But in what sense can the fixed point be the *least* one? For that, we need to introduce some notion of order on the set of predicates, i.e. on $\mathcal{P}(\Sigma)$. More concretely, $\Longrightarrow$ induces a complete lattice (see Definition A.1) on $\mathcal{P}(\Sigma)$, i.e.

$$F_1 \text{ „is smaller than or equal to“ } F_2 \qquad \text{iff} \qquad F_1 \Longrightarrow F_2\,.$$

The least element of the *complete lattice* $(\mathcal{P}(\Sigma), \Longrightarrow)$ is false. The supremum of a chain $S \subseteq \mathcal{P}(\Sigma)$ is given by

$$\sup S \;=\; \bigvee_{F \in S} F\,,$$

```
/// y > 0
/// (y > 0 ∧ true) ∨ (y ≤ 0 ∧ false)
if (y > 0) {
    /// true
    /// (5 − 3)^2 > 2
    x := 5
    /// (x − 3)^2 > 2
}else{
    /// false
    /// (2 − 3)^2 > 2
    x := 2
    /// (x − 3)^2 > 2
};
/// (x − 3)^2 > 2
y := x − 3;
/// y^2 > 2
skip
/// y^2 > 2
```

Figure 2.4: Weakest precondition annotations for Example 2.5.

which is the predicate that corresponds to the union of all sets corresponding to the predicates in S.

One can now show that Φ is a continuous function (Definition A.2) und we thus know by the Kleene fixed point theorem (Theorem A.5) that Φ has a least fixed point, given by

$$\mathsf{lfp}\,\Phi \;=\; \sup_{n\in\mathbb{N}} \Phi^n(\mathsf{false})\,.$$

By the existence of the least fixed point, we have ensured that wp for while loops is well–defined.

The evaluation of $\Phi^0(\mathsf{false})$, $\Phi^1(\mathsf{false})$, $\Phi^2(\mathsf{false})$, ... is called a *fixed point iteration* and we call $\Phi^i(\mathsf{false})$ the i-th iteration or i-th step of that fixed point iteration. A very important fact about the fixed point iteration of continuous functions is that, if started from the least element of the underlying complete lattice, it converges *monotonically* to the least fixed point, meaning that in our particular case we have an ascending chain

$$\Phi^0(\mathsf{false}) \implies \Phi^1(\mathsf{false}) \implies \Phi^2(\mathsf{false}) \implies \Phi^3(\mathsf{false}) \implies \ldots\,.$$

This follows by induction from continuity of Φ which implies monotonicity of Φ (see Definition A.3 and A.4). For the base case, we have

$$\Phi^0(\mathsf{false}) \;=\; \mathsf{false} \implies \Phi(\mathsf{false})$$

trivially, since false implies anything. Then, by monotonicity, we can perform the induction step. Assuming $\Phi^n(\mathsf{false}) \implies \Phi^{n+1}(0)$, we get

$$\Phi^{n+1}(\mathsf{false}) \implies \Phi^{n+2}(0)$$

by monotonicity of Φ. Let us revisit Example 2.3, and reason about a possibly unwanted side–effect of that program (setting x to 0) using the wp calculus.

Example 2.6 (Weakest Preconditions of Loops):
Reconsider the program $C_{2.3}$ from Example 2.3:

$C_{2.3} \rhd$
```
z := y;
while(x > 0){
    z := z + 1;
    x := x − 1
}
```

We would like to reason about whether the program sets x exactly to 0, i.e. about postcondition $x = 0$. The characteristic function of the while loop with respect to postcondition $x = 0$ is given by

$$\begin{aligned}\Phi(X) &= (x \leq 0 \land x = 0) \lor (x > 0 \land \mathsf{wp}\,[\![z := z+1 \fatsemi x := x-1]\!]\,(X)) \\ &= (x = 0) \lor (x > 0 \land \mathsf{wp}\,[\![z := z+1 \fatsemi x := x-1]\!]\,(X))\,.\end{aligned}$$

Let us do the first three iterations of the fixed point iteration for Φ:

$$\begin{aligned}\Phi^0(\mathsf{false}) &= \mathsf{false} \\ \Phi^1(\mathsf{false}) &= (x = 0) \\ \Phi^2(\mathsf{false}) &= (x = 0) \lor (x = 1) \\ \Phi^3(\mathsf{false}) &= (x = 0) \lor (x = 1) \lor (x = 2)\end{aligned}$$

Detailed calculations can be found in Appendix C.1 on page 323.

After three iterations, we can already start seeing a pattern for $n > 1$:

$$\Phi^n(\mathsf{false}) = (x = 0) \lor (x = 1) \lor \ldots \lor (x = n-1) = \bigvee_{i=0}^{n-1} (x = i)$$

We could prove this pattern correct by induction on n, which we however omit here. The above fixed point iteration will converge to the precondition

$$\sup_{n \in \mathbb{N}} \Phi^n(\mathsf{false}) = \bigvee_{i=0}^{\omega} (x = i) = (x \in \mathbb{N})\,,$$

and thus

$$\mathsf{wp}\,[\![\mathtt{while}\,(x > 0)\,\{z := z+1 \fatsemi x := x-1\}]\!]\,(x = 0) = (x \in \mathbb{N})\,. \tag{2.2}$$

For the whole program, we can finally make these annotations:

$$\begin{aligned}&/\!/\!/\ x \in \mathbb{N} \\ &z := y \fatsemi \\ &/\!/\!/\ x \in \mathbb{N} \qquad \text{(by Equation 2.2)} \\ &\mathtt{while}\,(x > 0)\,\{ \\ &\quad z := z+1 \fatsemi \\ &\quad x := x-1\ \} \\ &/\!/\!/\ x = 0\end{aligned}$$

We have thus proven that from all initial states where x is a natural number, the program sets x to 0.

While the above reasoning about the while loop was more or less ad–hoc, formal reasoning about such fixed points in a systematic way is one of the most difficult tasks in program verification. In general, this is not automatable, as

this would contradict Rice's Theorem [Ric53] and therefore ultimately contradict the undecidability of the Halting Problem [Chu36; Tur37]. We show how to reason about loops in a possibly more automatable way in Chapter 5.

2.2.4 *Reasoning about Nondeterminism*

So far, all GCL constructs were of deterministic nature: Given an initial state, the behavior of the program was completely determined. We will now introduce some notion of uncertainty into our GCL programming language: the *nondeterministic choice* construct

$$\{C_1\} \square \{C_2\}\,.$$

We call programs that contain such nondeterministic choices *nondeterministic programs*. Analogously, we call programs that do not contain any nondeterministic choice constructs *deterministic programs*. The concept of programs containing nondeterministic choices was already present in Dijkstra's original weakest precondition calculus [Dij75], although the idea of „nondeterministic algorithms" is due to Floyd and dates back further [Flo67b]. Even earlier, as a precursor to nondeterministic programs, Rabin & Scott introduced nondeterministic finite automata [RS59].

As for the semantics of nondeterministic choice, the program $\{C_1\} \square \{C_2\}$ executes *either* C_1 *or* C_2. Both scenarios are possible and we simply have no information on which branch is going to be executed. In particular, we would like to stress that it is *not meaningful to associate a probability* to either executing C_1 or C_2. Especially assigning the probability of $1/2$ to either possibility is only seemingly self–evident, but not meaningful. Nondeterministic choice is thus a *possibilistic*, not a probabilistic construct. For semantics of nondeterministic and possibilistic programs based on *possibility theory* instead of probability theory, see [CW08; WC12; WC11].

Example 2.7 (A Nondeterministic Loop–Free GCL Program):
Consider the following program that extends Example 2.2 on page 25:

$$\begin{aligned} C_{2.7} \rhd \quad & \{y := 1\} \square \{y := y - 1\}\fatsemi \\ & \texttt{if}\,(y > 0)\,\{x := 5\}\,\texttt{else}\,\{x := 2\} \\ & y := x - 3\fatsemi \\ & \texttt{skip} \end{aligned}$$

This program first nondeterministically either sets y to 1 or decreases y by 1. Then it performs the same steps as the program from Example 2.2, starting with the check for $y > 0$. Notice that this check can either evaluate to true, i.e. in case that the left branch of the nondeterministic choice was executed

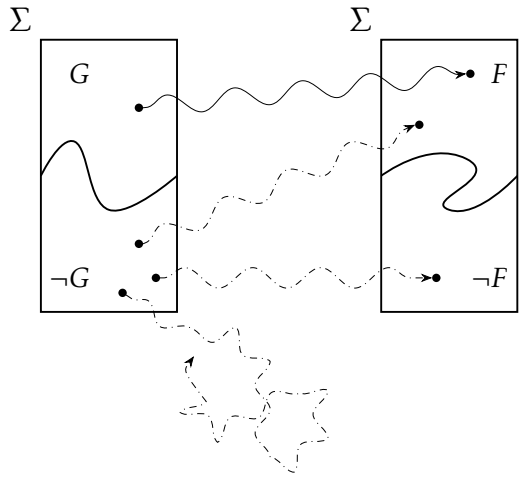

Figure 2.5: G is the weakest precondition of nondeterministic program C with respect to postcondition F: Starting in G, C will terminate in F. Starting in $\neg G$, we cannot guarantee anything about the computation of C.

and y was set to 1, or it can depend on the initial value of y, i.e. in case that the right branch was executed.

Let us now look at weakest preconditions of nondeterministic choices: Recall that we are interested in a precondition that *guarantees* both termination and establishment of the postcondition F. In order to guarantee this regardless whether C_1 or C_2 is executed, the weakest precondition of $\{C_1\} \square \{C_2\}$ with respect to F must be a weakest precondition of *both* C_1 *and* C_2 with respect to F. The weakest precondition transformer for $\{C_1\} \square \{C_2\}$ is thus given by

$$\mathsf{wp}\,[\![\{C_1\} \square \{C_2\}]\!]\,(F) \;=\; \mathsf{wp}\,[\![C_1]\!]\,(F) \;\wedge\; \mathsf{wp}\,[\![C_2]\!]\,(F)\;.$$

We might recall at this point that for a deterministic program C we could make the following statement:

> If $\sigma \not\models \mathsf{wp}\,[\![C]\!]\,(F)$ then we know that executing C on state σ will *definitely not* terminate in a state $\tau \models F$.

For a nondeterministic C, however, the statement must be:

> If $\sigma \not\models \mathsf{wp}\,[\![C]\!]\,(F)$ then it is *not guaranteed* that executing C on state σ will terminate in a state $\tau \models F$.

The situation is depicted in Figure 2.5. Notice that this diagram is *the same* as the one in Figure 2.1. There is a hidden difference in the possible compu-

tation starting in $\neg G$ and terminating in F, though: For Hoare triples with deterministic program C, this possible path stems from the fact that validity of the Hoare triple $\langle G \rangle\ C\ \langle F \rangle$ is too weak a statement to exclude this path. For weakest preconditions of nondeterministic programs on the other hand, the path from $\neg G$ to F is instead due to the nondeterminism of C: The path from $\neg G$ to F might actually be a possible computation of C. However: computations starting from $\neg G$ are *not guaranteed* to terminate in F.

Example 2.8 (Weakest Preconditions and Nondeterminism):
We will reconsider the program $C_{2.7}$ from Example 2.7 and again, as in Example 2.2, reason about postcondition $y^2 > 2$. Using the annotation style from earlier, we annotate $C_{2.7}$ as shown in Figure 2.6. By these annotations, we establish $\text{wp}\,[\![C_{2.7}]\!]\left(y^2 > 2\right) = (y > 1)$. This means that from any initial state in which y is larger than 1, it is guaranteed that execution of $C_{2.7}$ will terminate in a state in which y^2 is larger than 2.

Notice that even from a state in which $y \leq 1$ it is still possible that the program terminates in a state satisfying $y^2 > 2$, namely if in the nondeterministic choice the left branch $y := 1$ is executed. However, this is *not guaranteed*. This situation is reflected exactly by the path from $\neg G$ to F in Figure 2.5, when instantiating G with $y > 1$ and F with $y^2 > 2$.

2.2.5 *Weakest Liberal Preconditions*

We have already encountered the phenomenon that certain programs do not terminate from certain initial states. For instance, the program

$$\texttt{while}(x \neq 0)\{x := x - 1\}$$

terminates only on initial states where $x \in \mathbb{N}$. Our notion of weakest preconditions, however, captures only the fact that a program terminates in a state satisfying a given postcondition. Sometimes (e.g. later in this thesis), it is necessary, though, to reason about partial correctness, namely that a postcondition has to be satisfied only in case that the program terminates (but termination itself is not guaranteed).

2.2.5.1 *The Notion of Weakest Liberal Preconditions*

The type of reasoning we require here can be carried out using the notion of *weakest liberal preconditions*: Given a program C and a postcondition F, we call the (unique) predicate G the

weakest liberal precondition of C with respect to F,

```
/// y > 1
/// true ∧ y > 1
{
    /// true
    /// 1 > 0
    y := 1
    /// y > 0
} □ {
    /// y > 1
    /// y − 1 > 0
    y := y − 1
    /// y > 0
};
/// y > 0                                          (see Example 2.5)
if (y > 0) {x := 5} else {x := 2};
y := x − 3;
skip
/// y² > 2
```

Figure 2.6: Weakest precondition annotations for Example 2.8.

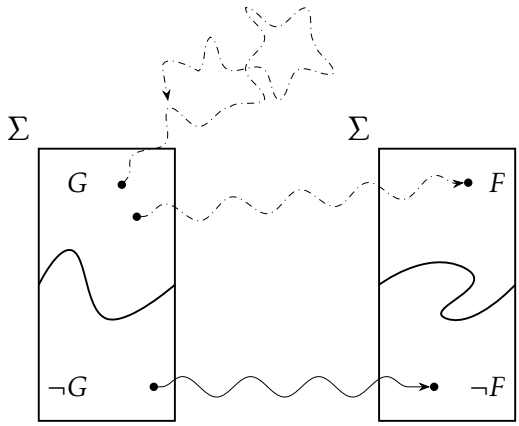

Figure 2.7: G is the weakest liberal precondition of deterministic program C with respect to postcondition F: Starting in G, C cannot terminate in $\neg F$, i.e. it will either diverge or terminate in F. Starting in $\neg G$, C will terminate in $\neg F$.

denoted $\mathsf{wlp}\,[\![C]\!]\,(F)$, if it satisfies the following: From any state $\sigma \models G$

- either C terminates in a state $\tau \models F$,
- or C does not terminate at all.

Moreover, if C is started in a state $\sigma \not\models G$, then C terminates in a state $\tau \not\models F$. As a diagram, the situation is depicted Figure 2.7 for deterministic programs and in Figure 2.8 for nondeterminstic programs. The difference is only in those computations starting from $\neg G$: For the possible computation path from $\neg G$ to F, recall the explanations on page 38. The possible diverging path emanating from $\neg G$ is also caused by the nondeterminism of the program: It is possible that C diverges from $\neg G$ but it is not guaranteed.

If F is some correctness property and we can prove the above, then we say that C is *partially correct*, whereas if we additionally require termination (as it was the case with weakest preconditions) we say that C is *totally correct*. In that terminology, weakest preconditions are suited for reasoning about total correctness whereas weakest liberal preconditions are suited for reasoning about partial correctness.

2.2.5.2 *The Weakest Liberal Precondition Calculus*

We now show how to obtain weakest liberal preconditions in a way similar to the weakest precondition transformer, namely by a backward moving

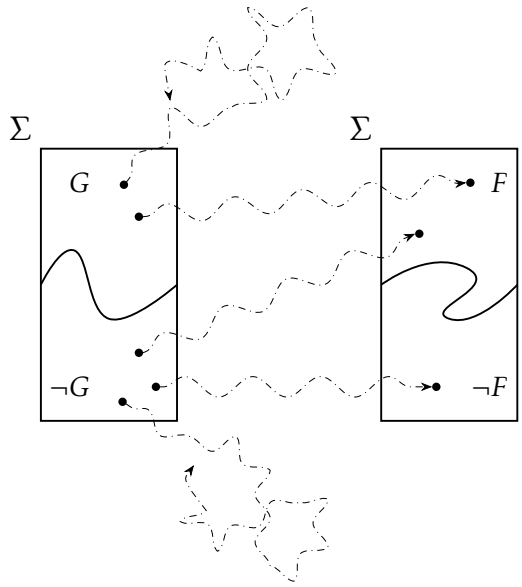

Figure 2.8: G is the weakest liberal precondition of nondeterministic program C with respect to postcondition F: Starting in G, C will not terminate in $\neg F$.

continuation–passing style weakest liberal precondition transformer.

Weakest Liberal Preconditions of Loop–Free Programs. The rules for constructing the wlp transformer are given in Table 2.2. Let us again ignore the definition for while loops for the time being and inspect the remaining rules: For the atomic programs `skip` and $x := E$ the rules are exactly the same.

For the remaining loop–free programs, the definitions differ only in the fact that the right hand sides use wlp instead of wp on subprograms. From that observation, we can easily conclude that wlp and wp coincide for any loop–free program, i.e.

$$\forall \text{ loop–free } C \ \forall F\colon \quad \mathsf{wlp}\,[\![C]\!]\,(F) \;=\; \mathsf{wp}\,[\![C]\!]\,(F)\ .$$

This does not only make sense when looking at the formal definitions, but it also makes sense intuitively: Differences between wp and wlp occur only for nontermination, but this cannot occur in loop–free programs.[6]

Weakest Liberal Preconditions of Loops. We now turn towards weakest liberal preconditions of loops. As for `diverge`, consider the following: Ac-

6 Recall that programs containing `diverge` are *not* loop–free.

C	$\mathsf{wlp}\,[\![C]\!]\,(F)$
`skip`	F
`diverge`	true
$x := E$	$F[x/E]$
$C_1 \fatsemi C_2$	$\mathsf{wlp}\,[\![C_1]\!]\,\big(\mathsf{wlp}\,[\![C_2]\!]\,(F)\big)$
`if` $(\varphi)\,\{C_1\}$ `else` $\{C_2\}$	$\big(\varphi \wedge \mathsf{wlp}\,[\![C_1]\!]\,(F)\big) \vee \big(\neg\varphi \wedge \mathsf{wlp}\,[\![C_2]\!]\,(F)\big)$
$\{C_1\} \,\square\, \{C_2\}$	$\mathsf{wlp}\,[\![C_1]\!]\,(F) \;\wedge\; \mathsf{wlp}\,[\![C_2]\!]\,(F)$
`while`$(\varphi)\{C'\}$	$\mathsf{gfp}\; X.\, \big(\neg\varphi \wedge F\big) \vee \big(\varphi \wedge \mathsf{wlp}\,[\![C']\!]\,(X)\big)$

Table 2.2: The weakest liberal precondition transformer.

cording to the definition of weakest liberal preconditions, the weakest liberal precondition of `diverge` with respect to postcondition F must be a predicate such that either `diverge` terminates in a state $\tau \models F$ (however, `diverge` never terminates, so this is never the case), or `diverge` does not terminate (this is always the case). Therefore, the weakest liberal precondition of `diverge` with respect to *any* postcondition can only be true.

Dually to weakest preconditions, we see in Table 2.2 that the weakest liberal precondition of a while loop is defined using a *greatest fixed point operator* (gfp) instead of a least one as

$$\mathsf{gfp}\; X.\, \underbrace{(\neg\varphi \wedge F) \vee (\varphi \wedge \mathsf{wlp}\,[\![C']\!]\,(X))}_{=:\,\Phi(X)},$$

i.e. the greatest fixed point of the characteristic function $\Phi(X)$.

Dually to least elements and suprema, a complete lattice also always has a greatest element and every subset also has an infimum. The greatest element of the complete lattice $(\mathcal{P}(\Sigma),\ \Longrightarrow)$ is true. The infimum of a subset $S \subseteq \mathcal{P}(\Sigma)$ is given by

$$\inf S \;=\; \bigwedge_{F \in S} F\,,$$

which is the predicate that corresponds to the intersection of all sets corresponding to the predicates in S.

One can now show that Φ is continuous and we thus know by the Kleene fixed point theorem (A.5) that Φ has a greatest fixed point, given by

$$\mathsf{gfp}\,\Phi \;=\; \inf_{n \in \mathbb{N}} \Phi^n(\mathsf{true})\,,$$

and therefore wlp for while loops is well–defined.

Dually to the situation with least fixed points, a very important fact about the fixed point iteration of continuous functions is that, if started from the *greatest* element of the underlying complete lattice, it converges *monotonically* to the *greatest fixed point*. This means in our particular case that we have a *descending* chain

$$\Phi^0(\mathsf{true}) \Longleftarrow \Phi^1(\mathsf{true}) \Longleftarrow \Phi^2(\mathsf{true}) \Longleftarrow \dots .$$

This fact also follows from the monotonicity of Φ which is implied by its continuity (see A.4).

We can now reconsider `diverge` from a gfp point of view. Recall that `diverge` is a shorthand for `while(true){skip}`. Then the characteristic functional with respect to any postcondition F is given by

$$\Phi(X) \;=\; (\mathsf{false} \wedge F) \;\vee\; (\mathsf{true} \wedge \mathsf{wp}\,[\![\mathtt{skip}]\!]\,(X)) \;=\; X\,,$$

i.e. the identity function on $\mathcal{P}(\Sigma)$. Its largest fixed point is obviously the largest element of $\mathcal{P}(\Sigma)$, namely true. Therefore,

$$\mathsf{wp}\,[\![\mathtt{diverge}]\!]\,(F) \;=\; \mathsf{wp}\,[\![\mathtt{while(true)\{skip\}}]\!]\,(F) \;=\; \mathsf{true}$$

Let us again revisit an example and prove the partial correctness of a program using the wlp calculus:

Example 2.9 (Weakest Liberal Preconditions):
Reconsider the following program from earlier in this section:

$$C_{2.9} \rhd \qquad \mathtt{while}(x \neq 0)\{x := x - 1\}$$

We would like to reason about the fact that if $C_{2.9}$ terminates, it sets x to 0. We can do so by reasoning about the weakest liberal precondition of $C_{2.9}$ with respect to postcondition $x = 0$. The characteristic function of the loop with respect to postcondition $x = 0$ is given by

$$\begin{aligned}\Phi(X) \;&=\; (x = 0 \wedge x = 0) \;\vee\; (x \neq 0 \wedge \mathsf{wp}\,[\![x := x - 1]\!]\,(X)) \\ &=\; (x = 0) \;\vee\; (x \neq 0 \wedge \mathsf{wp}\,[\![x := x - 1]\!]\,(X))\,.\end{aligned}$$

Let us perform the fixed point iteration for Φ (for wlp the fixed point iteration for the greatest fixed point goes true, $\Phi(\mathsf{true})$, $\Phi^2(\mathsf{true})$, $\Phi^3(\mathsf{true})$, … instead of false, $\Phi(\mathsf{false})$, $\Phi^2(\mathsf{false})$, $\Phi^3(\mathsf{false})$, … as for wp):

$$\begin{aligned}\Phi(\mathsf{true}) \;&=\; (x = 0) \;\vee\; (x \neq 0 \wedge \mathsf{wp}\,[\![x := x - 1]\!]\,(\mathsf{true})) \\ &=\; (x = 0) \;\vee\; (x \neq 0 \wedge \mathsf{true}) \\ &=\; (x = 0) \;\vee\; (x \neq 0)\end{aligned}$$

$= \mathsf{true}$

We see that after only one iteration we have reached a fixed point. By monotonicity of Φ, this is the greatest fixed point and we hence have

$$\mathsf{wlp}\,[\![\mathtt{while}\,(x \neq 0)\,\{x := x-1\}]\!]\,(x=0) \;=\; \mathsf{true}\,.$$

We have thus formally proven the partial correctness property that from *all* initial states the program $C_{2.9}$ sets x to 0 if it terminates.

2.3 REASONING ABOUT VALUES

Up until now, we have recapped Dijkstra's original weakest precondition calculus which enables reasoning at the level of predicates over program states. We will now see how to take this method of reasoning beyond the level of predicates to more general functions.

Recall for this purpose our notion of predicates: We have identified a predicate F with a subset of program states, i.e. $F \in \mathcal{P}(\Sigma)$. We also introduced the Iverson bracket $[F]$ which is the indicator function of F and is of type $\Sigma \to \{0,1\}$. It is obvious that, *in principle*, predicates $F \in \mathcal{P}(\Sigma)$ and their indicator functions $[F] : \Sigma \to \{0,1\}$ are the same.

As a first step to go beyond predicates, we reformulate Dijkstra's weakest precondition calculus in terms of indicator functions, i.e. functions f of type $\Sigma \to \{0,1\}$. The resulting definitions are given in Table 2.3. Let us go exemplarily over the rules for assignment, conditional, and nondeterministic choice: For the assignment, we have

$$\mathsf{wp}\,[\![x := E]\!]\,(f) \;=\; f\,[x/E]\,,$$

where $f\,[x/E]$ is defined analogously to the case for predicates as

$$f\,[x/E] \;=\; \lambda\sigma.\, f(x[x \mapsto \sigma(E)])\,.$$

This mimics the definition of wp of assignments for predicates.

For `diverge`, we have

$$\mathsf{wp}\,[\![\mathtt{diverge}]\!]\,(f) \;=\; 0\,,$$

This mimics the definition of wp of divergence for predicates since $0 = [\mathsf{false}]$.

For the conditional choice we have

$$\mathtt{if}\,(\varphi)\,\{C_1\}\,\mathtt{else}\,\{C_2\} \;=\; [\varphi]\cdot\mathsf{wp}\,[\![C_1]\!]\,(f) \;+\; [\neg\varphi]\cdot\mathsf{wp}\,[\![C_2]\!]\,(f)\,,$$

where $\cdot$ and $+$ are to be understood pointwise, i.e.

$$f_1 \cdot f_2 \;=\; \lambda\sigma.\, f_1(\sigma)\cdot f_2(\sigma) \quad\text{and}\quad f_1 + f_2 \;=\; \lambda\sigma.\, f_1(\sigma) + f_2(\sigma)\,.$$

C	$\mathsf{wp}\,[\![C]\!]\,(f)$
`skip`	f
`diverge`	0
$x := E$	$f\,[x/E]$
$C_1 \mathbin{;} C_2$	$\mathsf{wp}\,[\![C_1]\!]\,\big(\mathsf{wp}\,[\![C_2]\!]\,(f)\big)$
$\texttt{if}\,(\varphi)\,\{C_1\}\,\texttt{else}\,\{C_2\}$	$[\varphi]\cdot \mathsf{wp}\,[\![C_1]\!]\,(f) \;+\; [\neg\varphi]\cdot \mathsf{wp}\,[\![C_2]\!]\,(f)$
$\{C_1\}\,\square\,\{C_2\}$	$\min\big\{\mathsf{wp}\,[\![C_1]\!]\,(f),\, \mathsf{wp}\,[\![C_2]\!]\,(f)\big\}$
$\texttt{while}(\varphi)\{C'\}$	$\mathsf{lfp}\; X.\; [\neg\varphi]\cdot f \;+\; [\varphi]\cdot \mathsf{wp}\,[\![C']\!]\,(X)$

Table 2.3: The weakest precondition transformer acting on indicator functions. This transformer serves also as an anticipation transformer acting on more general functions of type $f\colon \Sigma \to \mathbb{R}^{\infty}_{\geq 0}$.

The definition of the conditional choice using $\cdot$ and $+$ instead of $\wedge$ and $\vee$ is meaningful since for predicates $F, G \in \mathcal{P}(\Sigma)$ we have that

$$[F \wedge G] \;=\; [F]\cdot[G] \quad \text{and} \quad [F \vee G] \;=\; [F]+[G]\,,$$

and thus $\wedge$ corresponds to $\cdot$ and $\vee$ corresponds to $+$.

For the nondeterministic choice construct we have

$$\{C_1\}\,\square\,\{C_2\} \;=\; \min\{\mathsf{wp}\,[\![C_1]\!]\,(f),\, \mathsf{wp}\,[\![C_2]\!]\,(f)\}\,,$$

where min is also to be understood pointwise, i.e.

$$\min\{f_1, f_2\} \;=\; \lambda\sigma.\; \min\{f_1(\sigma), f_2(\sigma)\}\,.$$

This is meaningful since for predicates $F, G \in \mathcal{P}(\Sigma)$ we have that

$$[F \wedge G] \;=\; \min\{[F], [G]\}\,,$$

and therefore $\wedge$ not only corresponds to $\cdot$ but also to min. The choice of $\cdot$ for the conditional choice and min for the nondeterministic is somewhat arbitrary at this point but we will say more about the role of min shortly. A very high–level intuition at this point is that we want to express by $a \cdot b$ the *logical connective* „both a and b must be true“, whereas by $\min\{a, b\}$ we want to *select* the „least true option from a and b“.

2.3.1 *Anticipated Values*

We saw how to reformulate the weakest precondition calculus to act on functions of type $f\colon \Sigma \to \{0, 1\}$. In terms of the reformulated calculus, we can

reason about whether program C will terminate in a state satisfying a predicate F by calculating

$$\mathsf{wp}\,[\![C]\!]\,([F])\ .$$

So from any state σ with $\mathsf{wp}\,[\![C]\!]\,([F])(\sigma) = 1$ the program C will terminate in a state $\tau \models F$, and from any state σ with $\mathsf{wp}\,[\![C]\!]\,([F])(\sigma) = 0$ the program C will either terminate in a state $\tau \not\models F$ or not terminate at all. This means that $\mathsf{wp}\,[\![C]\!]\,([F])$ is a function that *anticipates the truth of* F after termination of C, or in other words:

$\mathsf{wp}\,[\![C]\!]\,([F])$ is the *anticipated value* of $[F]$.

Now that we know that we can use wp to anticipate values of functions of type $\Sigma \to \{0, 1\}$, a natural question arises:

Can we anticipate values of more general functions?

For example: can we anticipate the value of program variable x after termination of C; or as another example: can we anticipate the value of $y^2 + |\sin z|$? It turns out that the answer to that question is *yes*.

2.3.2 *An Anticipated Value Calculus for Deterministic Programs*

Consider for the moment only *deterministic* programs. We would now like to reason about anticipated values of a more general class of functions, namely functions from the set of *anticipations*:

DEFINITION 2.10 (Anticipations):

A. *The set of anticipations is defined as*

$$\mathbb{A} \;=\; \left\{ f \;\middle|\; f \colon \Sigma \to \mathbb{R}^{\infty}_{\geq 0} \right\},$$

where $\mathbb{R}^{\infty}_{\geq 0}$ is the set of non–negative real numbers with an adjoined ∞ element which is larger than every real number.

B. *A complete lattice on $\mathbb{A}$ is induced by the partial order*

$$f_1 \;\preceq\; f_2 \qquad \text{iff} \qquad \forall \sigma \in \Sigma\colon \quad f_1(\sigma) \;\leq\; f_2(\sigma)\,.$$

The least element of the complete lattice $(\mathbb{A}, \preceq)$ is the function that maps every program state to 0, i.e. the function

$$\lambda\sigma\mathbf{.}\ 0\,,$$

which we (overloadingly) also denote by 0. The supremum of a subset $S \subseteq \mathbb{A}$ is constructed pointwise as

$$\sup\, S \;=\; \lambda\sigma\mathbf{.}\ \sup_{f \in S} f(\sigma)\,.$$

It turns out that for anticipating values of functions, we can just reuse the calculus from Table 2.3 but have the f's be taken from $\mathbb{A}$. In that sense, the transformer from Table 2.3 also serves as an *anticipated value transformer*. So if we want to know the value that an $f \in \mathbb{A}$ has after executing C, we just use f as the *postanticipation* and determine the *preanticipation* $\mathsf{wp}\,[\![C]\!]\,(f)$ according to Table 2.3. In that way we obtain the sought–after anticipated value of f. The completeness of the lattice $(\mathbb{A}, \preceq)$ ensures existence of least fixed points and thereby well–definedness of wp for loops. Notice that functions as

$$x \;=\; \lambda\sigma.\,\sigma(x) \quad \text{and} \quad y^2 + |\sin z| \;=\; \lambda\sigma.\,\sigma(y)^2 + |\sin \sigma(z)|$$

are both members of $\mathbb{A}$.[7] Notice furthermore that the wp calculus acting on $\mathbb{A}$ subsumes Dijkstra's original calculus since for every predicate F we have $[F] \in \mathbb{A}$ and *to all intents and purposes*, $\mathsf{wp}\,[\![C]\!]\,(F) = \mathsf{wp}\,[\![C]\!]\,([F])$. Even the order $\preceq$ on anticipations subsumes the order $\Longrightarrow$ on predicates since for predicates F_1 and F_2 we have

$$F_1 \Longrightarrow F_2 \quad \text{iff} \quad [F_1] \preceq [F_2]\,.$$

Example 2.11 (Anticipated Values of Deterministic Programs):

A. We reconsider the program $C_{2.2}$ from Example 2.2 on page 25 and instead of reasoning whether $y^2 > 2$, we will now directly anticipate the value of y^2 after execution of $C_{2.2}$. We will reuse our annotation style from earlier, i.e.

$$\begin{array}{l} /\!/\!/\ g' \\ /\!/\!/\ g \\ C \\ /\!/\!/\ f \end{array}$$

expresses the fact that $g = \mathsf{wp}\,[\![C]\!]\,(f)$ and moreover that $g' = g$. Since we want to anticipate the value of y^2, we will use the function y^2 as postanticipation and annotate $C_{2.2}$ using the rules from Table 2.3 as shown in Figure 2.9 (again: read from bottom to top).

In words, $\mathsf{wp}\,[\![C]\!]\,\left(y^2\right) = [y > 0] \cdot 4 \;+\; [y \leq 0]$ tells us that from any initial state σ with $y > 0$ we will end up in some final state with $y^2 = 4$, whereas if initially $y \leq 0$ we will end up in some final state with $y^2 = 1$.

B. Reconsider the program $C_{2.3}$ from Example 2.3:

7 We tacitly assume that x takes only positive values. Otherwise $\lambda\sigma.\,\sigma(x)$ would technically not be a member of $\mathbb{A}$. A more appropriate choice would be the function $[x > 0]\cdot x = \lambda\sigma.\,[x > 0](\sigma)\cdot\sigma(x)$ which *is* a member of $\mathbb{A}$, but we did not want to clutter the presentation above.

$C_{2.3}$ ▷

```
z := y;
while(x > 0){
    z := z + 1;
    x := x − 1
}
```

We want to reason about the value that program variable z has after the execution of $C_{2.3}$, i.e. about postanticipation z. The characteristic function of the loop with respect to postanticipation z is given by

$$\Phi(X) \;=\; [x \leq 0] \cdot z \;+\; [0 < x] \cdot \mathsf{wp}\,[\![z := z+1\,;\, x := x-1]\!]\,(X)\;.$$

The first three iterations of the fixed point iteration for Φ are:

$$\begin{aligned}
\Phi(0) &= [x \leq 0] \cdot z \;+\; [0 < x \leq 0] \cdot \lceil x \rceil \\
\Phi^2(0) &= [x \leq 1] \cdot z \;+\; [0 < x \leq 1] \cdot \lceil x \rceil \\
\Phi^3(0) &= [x \leq 2] \cdot z \;+\; [0 < x \leq 2] \cdot \lceil x \rceil
\end{aligned}$$

Detailed calculations can be found in Appendix C.2. After three iterations, we can already start seeing a pattern for $n > 1$:

$$\Phi^n(0) \;=\; [x \leq n-1] \cdot z \;+\; [0 < x \leq n-1] \cdot \lceil x \rceil$$

Again we omit proving the above pattern correct. By inspection of this pattern, we see that the preanticipation of the loop converges to

$$\begin{aligned}
&\mathsf{wp}\,[\![\texttt{while}(x > 0)\{\ldots\}]\!]\,(z) \\
&= \sup_{n \in \mathbb{N}} [x \leq n-1] \cdot z \;+\; [0 < x \leq n-1] \cdot \lceil x \rceil \\
&= z \;+\; [0 < x] \cdot \lceil x \rceil
\end{aligned}$$

For the whole program, we can finally make these annotations:

```
/// y + [0 < x] · ⌈x⌉
z := y;
/// z + [0 < x] · ⌈x⌉                    (see above)
while(x > 0){
    z := z + 1;
    x := x − 1
}
/// z
```

We have thus proven $\mathsf{wp}\,[\![C_{2.3}]\!]\,(z) = y + [0 < x] \cdot \lceil x \rceil$. This means that from all initial states $C_{2.3}$ terminates and the value of z after termination is the initial value of y plus — in case that x was initially positive — the initial value of $\lceil x \rceil$.

One issue we have not investigated so far is the anticipated value of a nonterminating program execution. Since it is not immediately clear, what the anticipated value should be, a remark on that matter is in order:

Remark 2.12 (Anticipated Values and Nontermination). Evaluation of the weakest preexpectation $\mathsf{wp}\,[\![C]\!]\,(f)$ at σ is (and indeed has to be) 0 if C does not terminate on σ. Thus, when observing e.g. $\mathsf{wp}\,[\![C]\!]\,(x)(\sigma) = 0$ alone, we cannot know offhand whether C terminates on σ in a state with $x = 0$ or whether C does not terminate on σ.

While it might seem somewhat arbitrary at first glance, we can get an intuition for that 0 by looking at the anticipated value of C with respect to $1 = [\mathsf{true}]$. Recall that $\mathsf{wp}\,[\![C]\!]\,(1)(\sigma)$ evaluates to 0 exactly if C does not terminate on σ and for reasons of continuity $\mathsf{wp}\,[\![C]\!]\,(f)(\sigma)$ has to evaluate to 0 for *any* f in case C does not terminate on σ. △

2.3.3 *Anticipated Value Calculi for Nondeterministic Programs*

We now turn towards anticipated values of nondeterministic programs. As we know, the nondeterministic choice $\{C_1\} \,\square\, \{C_2\}$ executes either C_1 or C_2 and we have no information on what is going to happen. We can therefore not speak of *the* anticipated value of a function f since there might be multiple values that a program can yield. For instance, the program

$$
\begin{array}{l}
x := 0 \,;\\
\{c := 0\} \,\square\, \{c := 1\} \,;\\
\mathtt{while}(c = 1)\{\\
\quad x := x + 1 \,;\\
\quad \{c := 0\} \,\square\, \{c := 1\}\\
\}
\end{array}
$$

may even yield any natural number for x or not terminate at all. The range of anticipated values of x is therefore infinite here.

For weakest preconditions of nondeterministic programs it made sense to choose the least true value $\min\{\mathsf{wp}\,[\![C_1]\!]\,(F), \mathsf{wp}\,[\![C_2]\!]\,(F)\}$ as the weakest precondition of $\{C_1\} \,\square\, \{C_2\}$ with respect to postcondition F. For anticipated values, this is a meaningful possible choice as well, i.e. we can define

$$\mathsf{wp}\,[\![\{C_1\} \,\square\, \{C_2\}]\!]\,(f) \;=\; \min\{\mathsf{wp}\,[\![C_1]\!]\,(f), \mathsf{wp}\,[\![C_2]\!]\,(f)\}\,,$$

```
/// [y > 0] · 4 + [y ≤ 0]
/// [y > 0] · 4 + [y ≤ 0] · 1
if (y > 0){
    /// 4
    /// (5 − 3)²
    x := 5
    /// (x − 3)²
}else{
    /// 1
    /// (2 − 3)²
    x := 2
    /// (x − 3)²
};
/// (x − 3)²
y := x − 3;
/// y²
skip
/// y²
```

Figure 2.9: Anticipated value annotations for Example 2.11 A.

where min is again meant pointwise, i.e.

$$\min\{f_1, f_2\} \;=\; \lambda\sigma.\, \min\{f_1(\sigma), f_2(\sigma)\}\ .$$

$\mathsf{wp}\,[\![\{C_1\} \,\square\, \{C_2\}]\!]\,(f)$ thus assigns to each initial state the *least* anticipated value of f. We call this the *demonic* model of nondeterminism. This model enjoys the nice property that it subsumes the weakest *precondition* calculus for nondeterministic programs and hence we will continue to use the symbol wp in the context of demonic nondeterminism.

While we just saw that min is a quite natural choice, there are use cases where choosing max instead of min is more meaningful, e.g. when reasoning about expected runtimes (see Chapter 7). In this case, we employ a so–called *angelic* model of nondeterminism, which gives us a different transformer awp for *greatest* anticipated values. This transformer is defined analogously to wp except on nondeterministic choice constructs, on which it is given as

$$\mathsf{awp}\,[\![\{C_1\} \,\square\, \{C_2\}]\!]\,(f) \;=\; \max\{\mathsf{awp}\,[\![C_1]\!]\,(f), \mathsf{awp}\,[\![C_2]\!]\,(f)\}\ .$$

Note that wp and awp obviously coincide on deterministic programs.

Both angelic and demonic nondeterminism are in some sense extremal and one could certainly think of other models. The advantage of these two models, however, is that they yield relatively easy definitions and the resulting calculi enjoy several nice properties.

Example 2.13 (Anticipated Values of Nondeterministic Programs): We will reconsider the program $C_{2.7}$ from Example 2.7 and reason about the *least* anticipated value of y^2 as shown in Figure 2.10. By these annotation, we have established $\mathsf{wp}\,[\![C]\!]\left(y^2\right) = 1 + [y > 1] \cdot 3$. This tells us that from any initial state in which y is larger than 1, C will terminate in a state where y is at least 4, and if initially $y \leq 1$ then in a state τ where y is at least 1.

Notice that even from a state in which $y \leq 1$ it is still possible that the program terminates in a state where y^2 is at least 4, namely if in the nondeterministic choice the left branch $y := 1$ is executed.

```
/// 1 + [y > 1] · 3
/// min{4, [y > 1] · 4 + [y ≤ 1]}
{
    /// 4
    /// [1 > 0] · 4 + [1 ≤ 0]
    y := 1
    /// [y > 0] · 4 + [y ≤ 0]
} □ {
    /// [y > 1] · 4 + [y ≤ 1]
    /// [y − 1 > 0] · 4 + [y − 1 ≤ 0]
    y := y − 1
    /// [y > 0] · 4 + [y ≤ 0]
};
/// [y > 0] · 4 + [y ≤ 0]                    (see Example 2.11)
if (y > 0) {x := 5} else {x := 2};
y := x − 3;
skip
/// y²
```

Figure 2.10: Anticipated value annotations for Example 2.13.